RĪGAS TEHNISKĀ UNIVERSITĀTE

MAĢISTRA DARBS

RĪGA 2021

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

Kristiāns KRONIS
maģistra profesionālo studiju programmas
students stud. apl. Nr. 151RDB207

RĪKA IZSTRĀDE DOCKER KONTEINERU TEHNOLOĢIJAS ATBALSTAM

MAĢISTRA DARBS

Zinātniskais vadītājs
asoc. prof., Dr.sc.ing.
Pāvels Rusakovs

RĪGA 2021

ANOTĀCIJA

Maģistra darba ietvaros tiek apskatītas konteineru tehnoloģijas, to priekšrocības, ieviešanas process un konteineru orķestrācija, kā arī ar to saistītie izaicinājumi.

Pirmajā nodaļā tiek apskatīta komandrindas rīka izstrāde attālinātai serveru administrēšanai caur SSH protokolu Python valodā. Tajā tiek implementēta funkcionalitāte infrastruktūras sagatavošanas automatizācijai un citu administratīvo darbību veikšanai. Tiek sniegti paraugi tā izmantošanai RPM GNU/Linux distribūciju grupas administrēšanai, par paraugu izmantojot CentOS 8 distribūciju.

Otrajā nodaļā tiek aprakstīta parauga lietotnes izstrāde konteinerizētas lietotnes un orķestratoru darbības testēšanai. Tā simulē COVID-19 inficēto izsekošanu ar GPS datu palīdzību un ir balstīta uz Ruby programmēšanas valodas, izstrādāta ar Ruby on Rails ietvaru, kā arī PostGIS ģeospatiālo relāciju datu bāzu vadības sistēmu. Tās mērķis ir kalpot par paraugu, kā horizontālā mērogošana var atļaut apstrādāt lielāku pieprasījumu skaitu. Tostarp, tā ļauj apskatīt, ar kādu slodzi varētu būt jārēķinās, izvēloties uz GPS bāzēto pieeju COVID kontaktu izsekošanai, nevis uz Bluetooth balstīto pieeju kontaktu fiksēšanai, jo īpaši ja tiek izmantotas augsta abstrakcijas līmeņa valodas un to izpildvides, kas kodu interpretē.

Trešajā nodaļā tiek aprakstīti izstrādātie slodzes testi ar K6 ietvaru, kurus izmanto, lai pārbaudītu, kā attiecīgie orķestratori ir spējīgi darboties zem slodzes. To mērķis ir simulēt inficēto cilvēku pārvietošanos un GPS datu iesūtīšanu apstrādei. Informācijas agregācijai tiek izmantots Zabbix monitoringa rīks, kurš ievāc informāciju no testētajiem serveriem.

Darba rezultātā tiek izdarīti secinājumi par to, cik viegli ir automatizēt serveru konfigurāciju un izstrādāt tai rīkus. Papildus tam, tiek sniegts ieskats Kubernetes un Docker Swarm orķestratoru īpatnībās, kā arī tiek izdarīti secinājumi par slodzes testēšanu.

Izstrādāto projektu pirmkods ir pieejams GitLab instancē: https://git.kronis.dev/rtu1

Maģistra darbā ir 133 lappuses, 66 attēli, 5 tabulas un 9 pielikumi. Tika izmantoti 38 literatūras avoti.

ANSIBLE, SALT, PYTHON, K6, DOCKER, DOCKER SWARM, KUBERNETES, K3S, RUBY, RUBY ON RAILS, POSTGIS, INFRASTRUKTŪRA KĀ KODS, SLODZES TESTĒŠANA, COVID-19 KONTAKTU IZSEKOŠANA

SATURS

IEVADS

Programmatūras izstrādes un uzturēšanas process pieprasa konstantu resursu ieguldījumu, lai novērstu dažādu risku iestāšanās iespējamību. Ir nepieciešams organizēt un plānot izmaiņu ieviešanu un plānot, kā šīs izmaiņas iederēsies kopējā risinājumu arhitektūrā, ir jāizmanto izmaiņu vadības sistēmas un, ja iespējams, šīs izmaiņas arī jādokumentē. Plānojot izmaiņas, ir jādomā gan par funkcionālajām, gan nefunkcionālajām prasībām. Tāpat, attiecībā uz pašu izmaiņu implementāciju, ir nepieciešams kodu realizēt konsistentā veidā ar pārējo koda bāzi, to dokumentēt vai nu ar pašdokumentējošā koda pieeju (angliski “self-documenting code”), vai arī ar koda komentāriem un zināšanu bāzēm. Šis kods ir arī jātestē, vai nu manuāli, vai nu ar automatizētajiem vienībtestiem, integrācijas testiem un slodzes testiem, kā arī ar citu veidu testiem. Papildus tam, nepieciešams arī domāt par laidienu pārvaldīšanu, konfigurācijas pārvaldīšanu, infrastruktūras monitoringu, kā arī pašas lietotnes analītikas nodrošināšanu, jo īpaši tīkla lietotņu izstrādes un uzturēšanas kontekstā.

Viss no minētā prasa gan laika, gan cilvēkresursu, gan skaitļošanas resursu ieguldīšanu. Ja tiek izlaists jebkurš no minētajiem procesiem, tad var izveidoties tehniskie parādi (angliski “compounding technical debt”), kas laika gaitā var sarežģīt un prasīgāku pret minētajiem resursiem padarīt jebkuru jauno izstrādes vai uzturēšanas darbu veikšanu. Tomēr, no otras puses, pārāk liels resursu ieguldījums minētajos izstrādes aspektos var programmatūras izstrādes projektu daudzos domēnos padarīt nespējīgu konkurēt tirgū vai realizēt visu plānoto funkcionalitāti ar ierobežotajiem resursiem, jo, salīdzinot ar alternatīvām, tā jaunās funkcionalitātes realizēšanas ātrums būs lēnāks, pat ja rezultējošā kvalitāte būs augstāka. Tāpēc biznesa interesēs var būt prioritizēt jaunās funkcionalitātes ieviešanu, kas, savukārt, var sarežģīt minēto procesu veikšanu, jo tiem netiks atvēlēts adekvāts daudzums laika. Tas liek domāt par rīkiem un metodēm, lai šo procesu veikšanu atvieglotu un paātrinātu, tā gan ietaupot resursus, gan nodrošinot iespēju strādāt ar standartizētiem rīkiem, tā atvieglojot gan informācijas pārneses procesu, gan arī ļaujot izmantot industrijā pieejamās darbinieku kompetences, neliekot apgūt tik lielu daudzumu uzņēmuma vai organizācijas ietvaros izveidoto rīku, tā paātrinot jauno resursu piesaistes procesa ātrumu projektam (angliski "onboarding").

Viena no šo rīku klasēm, kas var palīdzēt atvieglot darbu un samazināt risku iestāšanās iespējamību, ir konteinerizācijas rīki - tie var gan atļaut uzlabot koda izpildes vides reproducējamību, samazināt uzbrukuma vidi (angliski "attack surface") un nodalīt pašas infrastruktūrā palaistās lietotnes no serveriem un to programmatūras, kurās tie ir palaisti, minimizēt cilvēciskos faktorus, kas var būs saistīti ar nepilnīgu vai nepareizu laidienu uzlikšanu, veicināt automatizāciju uzņēmuma procesos, atvieglot testēšanu un izstrādes vižu palaišanu un konfigurāciju, kā arī uzlabot pašu lietotņu atklājamību (angliski "discoverability"), tā uzlabojot pašu izstrādātāju pieredzi darbā ar šiem risinājumiem. Taču tajā pašā laikā, daļa no šīm sistēmām, it īpaši runājot par konteineru orķestrāciju, ir sarežģītas un tāpēc ir nepieciešams domāt par to, kā šo netīšo sarežģītību (angliski "accidental complexity") mazināt un tās vietā savu uzmanību veltīt pašai domēna sarežģītībai (angliski "domain complexity"). To var realizēt ar dažādu rīku palīdzību, kas standartizētā veidā var atļaut rast risinājumus tipiskākajām problēmām to ieviešanā un uzturēšanā.

Šī maģistra darba ietvaros tiek apskatītas dažas no dotajā brīdī populārākajām konteineru un to orķestrācijas tehnoloģijām, kā arī tiek pētīta iespēja izveidot vienkāršotu serveru konfigurācijas rīku, ar uzsvaru uz serveru sagatavošanu konteinerizētu lietotņu palaišanai un konteineru orķestrācijai. Šis rīks kalpo par piemēru tam, ka šo automatizāciju ir iespējams realizēt visai vienkāršā veidā, pat ja ne vienmēr ir nepieciešamas tik lielas sistēmas kā Ansible. Lai nodemonstrētu ar šo rīku sagatavotās infrastruktūras darbību, tiek izstrādāta parauga sistēma, kura nodemonstrē tipiskās tīkla lietotnes darbību un to, kā šādu lietotni varētu izvietot uz vides konteinerizētā veidā. Tas ļauj salīdzināt arī pašu orķestratoru darbību un tiem nepieciešamos resursus, kā arī apskatīt atšķirības tiem domāto vižu deklarāciju formātos (angliski "manifest").

Pašai testa sistēmas realizācijai tika izvēlēta uz GPS bāzēta COVID kontaktu izsekošanas tematika. Neskatoties uz šādas pieejas trūkumiem privātuma ziņā, salīdzinājumā ar uz Bluetooth bāzētajām kontaktu fiksēšanas sistēmām, tāpat būtu lietderīgi izpētīt, kā šādu sistēmu izveide un lietošana varētu noslogot infrastruktūru. Šādās sistēmās būtu ievērojamāka centralizācija, kas, savukārt, liktu domāt par pašu sistēmu realizāciju un tajā izvēlētajām tehnoloģijām - ņemot vērā ierobežotos laika resursus šādu sistēmu izstrādei, pastāv iespēja, ka tam tiktu izmantotas augstākas abstrakcijas līmeņa programmēšanas valodas un ietvari,

piemēram PHP un Laravel, Ruby un Ruby on Rails, Python un Django. Tās ir raksturīgas ar dinamiskām tipu sistēmām un interpretētu kodu, kura izpilde var būt lēnāka, kā kompilētajām alternatīvām, kas var negatīvi ietekmēt sistēmas kopējo veiktspēju un resursu patēriņu - tāpēc var būt noderīgi veikt slodzes testus šādām sistēmām uz tipiskām aparatūras konfigurācijām, lai prognozētu, kādu lietotāju skaitu tās varētu atbalstīt.

Visbeidzot, serveru resursu patēriņu dati darba ietvaros tiek agregēti ar Zabbix programmatūru, kas nodemonstrē gan kā izstrādātais serveru pārvaldes risinājums ļauj automatizēt servera sagatavošanu monitoringam, gan arī ļauj uzskatāmā veidā ilustrēt šo risinājumu darbību un salīdzināt orķestratoru iespējamo ietekmi uz serverim pieejamiem resursiem un kopējo sistēmas veiktspēju, kā arī citām to īpatnībām. Nobeigumā, balstoties uz šo piefiksēto informāciju, tiek izdarīti secinājumi un rekomendācijas par to, kādus rīkus izmantot un kurās situācijās attiecīgie orķestratori varētu tikt izmantoti veiksmīgāk.

1. INFRASTRUKTŪRAS ADMINISTRĒŠANA

Viens no izaicinājumiem infrastruktūrā ir tās atjauninājumu pārvalde un konfigurācijas izmaiņu vadība. Ja infrastruktūru administrē manuāli, tad ir ievērojams cilvēciskā faktora risks - ja nepieciešamās izmaiņas tiek nodotas starp uzņēmumiem vai organizācijām, piemēram kāda produkta uzturēšanas ietvaros, tad pastāv iespēja, ka tās netiks pilnībā izprastas un/vai netiks izpildītas līdz galam. Savukārt, pat ja infrastruktūru administrē pats uzņēmums kurš izstrādā lietotni, vai arī tie paši izstrādātāji, tad tāpat pastāv risks izpildīt nepieciešamās darbības vai nu nepareizi, vai arī nepilnīgi, jo īpaši ja tiek administrēts liels skaits serveru un liels skaits vižu, kuru konfigurācijai vajadzētu būt vienādai. Līdzīgi, var rasties arī atšķirības konfigurācijā starp dažādām operētājsistēmu distribūcijām, piemēram "apt" rīka izmantošana paku pārvaldei DEB grupas GNU/Linux distribūcijās (piemēram, Debian un Ubuntu), savukārt "yum" rīka izmantošana RPM grupas GNU/Linux distribūcijās (piemēram, CentOS, Oracle Linux, Red Hat Enterprise Linux), kā arī atšķirības šo operētājsistēmu paku noklusējuma konfigurācijā.

Arī atjauninājumu automātiska uzlikšana var radīt situāciju, kurā noteiktas lietotnes darbojas konfigurācijā, kurā tās nebija testētas, ar jaunākām paku versijām - savukārt tā apiešana ar tādiem mehānismiem kā DEB grupas "apt madison" tāpat var novest pie situācijas, kur operētājsistēmas datņu sistēmā nepastāv skaidrs nodalījums starp sistēmas pakām un izstrādāto lietotni, kā arī tai nepieciešamo izpildvidi (piemēram, visas direktorijas, kurās var glabāties lietotnes serverim nepieciešamā konfigurācija, piemēram Apache Tomcat izmantotās direktorijas, kā arī izpildvide pašai lietotnei, piemēram OpenJDK). Papildus tam, pastāv iespēja, ka radīsies problēmas, ja būs nepieciešams palaist vairākas paralēlas lietotnes instances uz viena un tā paša servera - var parādīties portu konflikti, vai arī dublicēsies izmantotās datņu sistēmas mapes.

Savukārt mehānismiem, kurus var izmantot šādu situāciju novēršanai, piemēram virtuālo mašīnu piegādāšanai un uzturēšanai kā lietotnes izpildvidei, arī ir savi trūkumi. Virtuālo mašīnu lietošanas gadījumā var būt raksturīga sliktāka ātrdarbība, jo tiek uzturēts pilns operētājsistēmas kodols (angliski "kernel"), kā arī lielāks piegāžu izmērs.

Konteineru izmantošana šeit var palīdzēt, jo ļaus ieviest skaidru nodalījumu starp sistēmas programmatūru un palaižamo lietotni, nodrošināt centralizētu pieeju konfigurācijas nolasīšanai un audita izvades pārvaldei. Tāpat, tā var atvieglot lietotnei pieejamo resursu ierobežošanu, lai neatļautu situāciju, kurā pati sistēma paliek neresponsīva lielās noslodzes dēļ. Konteineru izmantošana atļaus arī norobežot pašas lietotnes no pārējās sistēmas un uzlabot to vides reproducējamību, neliekot lejupielādēt pakas vai atkarības dinamiski, pašas vides uzstādīšanas laikā. Tas nodrošinās, ka notestētās un piegādātās lietotnes kods tiešām sakritīs ar to, kurš darbosies noteiktajā vidē, tajā pašā laikā ļaujot pašai sistēmai veikt drošības un citus atjauninājumus, netraucējot konteinerizēto lietotņu darbībai, jo tie neizmantos sistēmas programmatūru izpildvidei, bet tā tiks iekļauta pašā konteinerā, tam dalot tikai operētājsistēmas kodolu un/vai atsevišķas datņu sistēmas mapes, kas pēc vajadzības varēs tikt padotas konteinerim. Konteineru orķestrācija, savukārt, atļaus realizēt ko līdzīgu servisu pārvaldei un automātiski restartēt konteinerus, kas ir apstājušies, kā arī ļaus dinamiski tos izvietot uz pieejamiem serveriem, tā atvieglojot slodzes balansēšanu un jaunu vižu izveidi.

Tomēr, arī konteineru izmantošanas gadījumā ir nepieciešams domāt par pašas infrastruktūras administrēšanu un pārvaldīšanu - pat ja darbības tā visa veikšanai ir mazāk, tāpat ir nepieciešams pēc iespējas minimizēt cilvēcisko kļūdu faktoru, tās iespēju robežās automatizējot. Lai to realizētu, industrijā ir pieejami vairāki dažādas sarežģītības rīki, taču to ieviešana bieži vien var būt saistīta ar papildus darbu, lai tos pielāgotu attiecīgā projekta vajadzībām. Tāpēc var būt vērts apsvērt alternatīva risinājuma izstrādes sarežģītību.

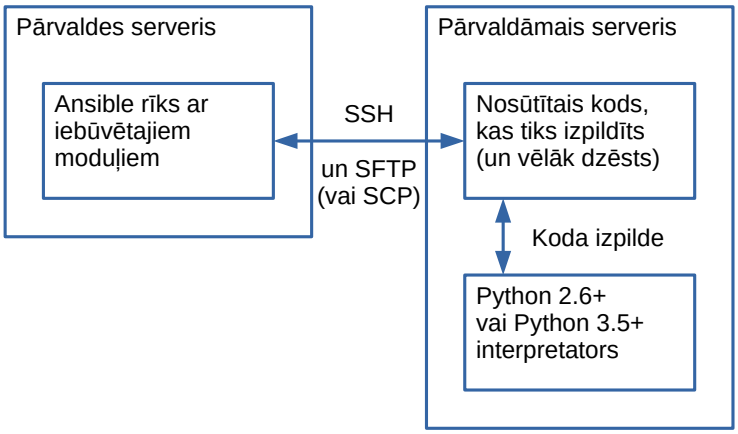
Šajā nodaļā primāri tiek aprakstītas prasības, kuras tika izvirzītas par svarīgām rīka izstrādei, lai nodrošinātu tā draudzīgumu lietotājam un uzlabotu rīka lietošanas pieredzi izstrādātājiem (angliski "developer experience" jeb DX), līdzīgi kā interfeisa ietvaros mēdz runāt par lietotāja pieredzi (angliski "user experience" jeb UX). Tāpat, tiek aprakstīts, kā katra no šīm prasībām tika realizēta un kādas pieejas un bibliotēkas tika izmantotas tā sasniegšanai. Tiek īsi aprakstīti arī industrijā izmantotie rīki un to iespējamie trūkumi.

Izstrādātā rīka pirmkods ir pieejams Git repozitorijā:
https://git.kronis.dev/rtu1/kvps5_masters_degree_astolfo_cloud_servant

1.1 Ansible un Salt rīki, to arhitektūra

Daži no populārajiem industrijā izmantotajiem serveru pārvaldes rīkiem ir Ansible un Salt - abi no tiem ir vērsti uz Linux operētājsistēmu (tostarp Alpine Linux, kas neizmanto GNU rīkus, tāpēc šeit nevar izmantot GNU/Linux nosaukumu) pārvaldi, ļaujot veikt dažādus administratīvos uzdevumus. Lai gan abi no rīkiem ir sarakstīti, izmantojot Python valodu, to pieejas serveru savienojuma nodrošināšanai un darbību izpildei atšķiras.

Ansible gadījumā, rīks ir bāzēts uz modulāras struktūras. Pašam rīkam tiek uzrakstīti moduļi, kuri ietver noteiktu funkcionalitāti, tie tiek izsaukti pateicoties YAML formātā dotam veicamo darbību aprakstam, kas tālāk rezultē pieslēguma izveidošanā ar administrējamo serveri caur SSH, datņu nosūtīšanā caur SFTP vai SCP un izpildīšanā uz attiecīgā servera. Tas nozīmē, ka uz tā arī ir nepieciešama saderīga Python izpildvide un ir ierobežotas iespējas bibliotēku izmantošanai moduļu funkcionalitātes realizēšanā (1.1. att.). Alternatīvi, Ansible piedāvā arī "raw" un "script" moduļus, kuri atļauj skriptu izpildi pa tiešo vai arī pārsūtīšanu uz attiecīgo serveri un izpildi tur, taču tie parasti netiek izmantoti visu administratīvo darbību izpildei [1].



1.1. att. Vienkāršots paraugs tam, kā Ansible izpilda komandas

Papildus tam, Ansible YAML uzdevumu formāts nodrošina moduļu izsaukšanu balstoties uz priekšnosacījumiem (piemēram, servera operētājsistēmas distribūcijas), kā arī

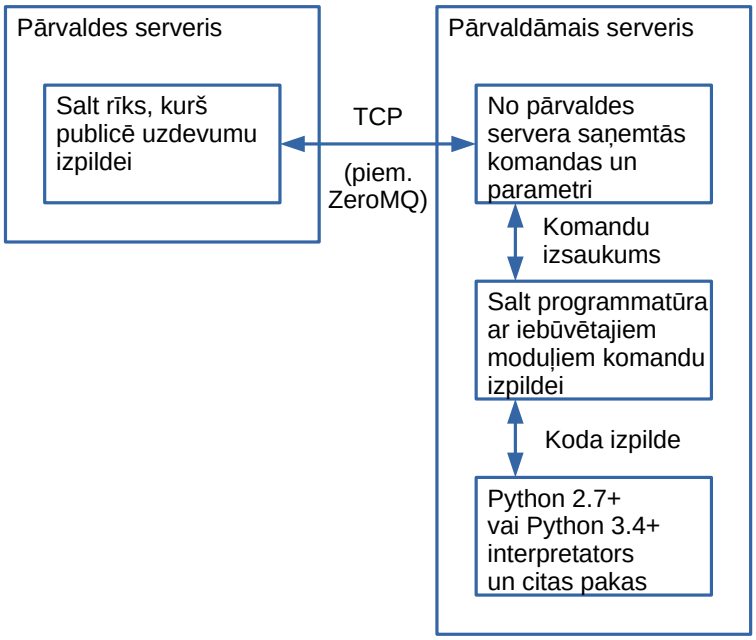
ciklu veidošanu un vides mainīgo padošanu, kā arī noteiktu uzdevumu grupēšanu un kopēju izpildi, kas ļauj realizēt sarežģītākas vižu konfigurācijas izveidi un uzturēšanu. Tajā pašā laikā, šāda arhitektūra var novest pie salīdzinoši sarežģītas paša rīka struktūras. Piemēram, Ansible gadījumā ir 22 dažādi veidi, kā padot mainīgos izpildei, ko darot ir jāņem vērā katra no šo veidu precedencēm [2] (1.2. att.).

1. command line values (for example, `-u my_user`, these are not variables)
2. role defaults (defined in role/defaults/main.yml) [1]
3. inventory file or script group vars [2]
4. inventory group_vars/all [3]
5. playbook group_vars/all [3]
6. inventory group_vars/* [3]
7. playbook group_vars/* [3]
8. inventory file or script host vars [2]
9. inventory host_vars/* [3]
10. playbook host_vars/* [3]
11. host facts / cached set_facts [4]
12. play vars
13. play vars_prompt
14. play vars_files
15. role vars (defined in role/vars/main.yml)
16. block vars (only for tasks in block)
17. task vars (only for the task)
18. include_vars
19. set_facts / registered vars
20. role (and include_role) params
21. include params
22. extra vars (for example, `-e "user=my_user"`)(always win precedence)

1.2. att. Paraugs Ansible mainīgo norādīšanas veidu precedencēm, no oficiālās dokumentācijas

Līdzīgi, Ansible nodrošina arī lielu daudzumu funkcionalitātes, kas nebūs vajadzīga visiem projektiem, piemēram uzdevumu grupēšanu blokos, vai arī notikumu apstrādes sistēmu (angliski "handlers"), kas ļaus reaģēt uz noteiktiem notikumiem izpildes gaitā, piemēram, no jauna ielādēt kāda servisa konfigurāciju un to pārstartēt pēc tam, kad izpildās uzdevums, kurš maina šī servisa definīcijas datnes. Lai gan šīs funkcionalitātes klātbūtne nenozīmē, ka tā būs tiešā veidā jāizmanto, ja pēc tās nav skaidras vajadzības, paši Ansible darbības paraugi un gatavie uzdevumu apraksti (angliski "playbooks"), ar kuriem var nākt sastapties, var būt sarežģīti un rīka pilnīgai apguvei var būt nepieciešams ievērojams laika ieguldījums.

Salt gadījumā arhitektūra ir nedaudz citāda - pretstatā Ansible, Salt gadījumā katrā no pārvaldītajiem serveriem jau būs pilnais kopums ar kodu, kas uz tiem jāizpilda, tiem no galvenā pārvaldes servera tiks pārsūtītas tikai izpildāmās komandas un to izpildei nepieciešamie parametri, kas nozīmē, ka tiks pārraidīts mazāks datu apjoms un tādēļ uzdevumu izpilde var notikt nedaudz ātrāk. Taču tādējādi rodas vajadzība pēc konfigurācijas visiem pārvaldāmajiem serveriem, lai Salt rīku varētu lietot to administrēšanai. Papildus tam, Salt izmanto nevis tikai SSH protokolu, bet gan vai nu ziņojumu rindu sistēmu (šajā gadījumā, ZeroMQ), vai arī nosūta komandas pa taisno caur TCP [3]. Tomēr, atšķirībā no SSH pieejas, šajā gadījumā paši administrējamie serveri izveido savienojumu ar galveno serveri, kas nozīmē, ka tas var atvieglot ugunsmūra konfigurāciju, jo publiski sasniedzamam tiešā veidā ir jābūt tikai galvenajam serverim.



1.3. att. Vienkāršots paraugs tam, kā Salt izpilda komandas

Kā alternatīva tomēr tiek nodrošināta starpnieku sistēma, kuru var izmantot, lai caur Salt programmatūru atbalstošiem serveriem varētu sazināties arī ar tiem, kas neatbalsta Salt standarta pieeju serveru pārvaldei - piemēram, komandas nosūtot caur HTTP API vai arī caur

SSH protokolu [4]. Līdzīgi, Salt nodrošina arī vairāk kā 20 paplašināmas sistēmas, kas atļauj to pielāgot dažādu vižu prasībām (piemēram, ja noteiktas autentifikācijas metodes netiek atbalstītas), taču tajā pat laikā var palielināt risinājuma tehnisko sarežģītību un radīt nepieciešamību pēc sevis uzrakstītās funkcionalitātes uzturēšanas [5].

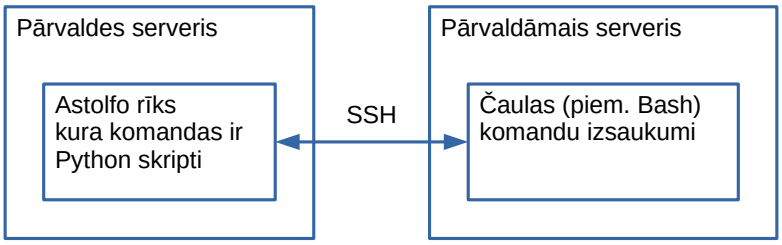
Minētie rīki var būt noderīgi, ja tos infrastruktūras pārvaldei izmanto centralizētā veidā - ja tos izvēlās par pamatu teju visu serveru pārvaldes automatizācijai un to ieviešanā iegulda adekvātus laika un cilvēkresursu līdzekļus, tā aprakstot tipisko uzdevumu veikšanu ar tiem un padarot to ieviešanu jaunos izstrādes darbos vieglāku. Tomēr tie var būt salīdzinoši sarežģīti un prasīgi pret administrējamām vidēm, kā dēļ ir lietderīgi arī izpētīt iespējas administrēt serverus ar vienkāršākām pieejām.

1.2 Alternatīva risinājuma izstrāde

Manuāli administrējot serverus, parasti tiek izmantotas čaulas (piemēram Bash), savienojumu ar serveriem izveidojot caur SSH protokolu. Šī pieeja Linux serveru administrēšanai ir visai populāra, jo prasa mazāk resursus un mazāku konfigurāciju kā grafiskās pārvaldes iespējas, piemēram VNC vai RDP protokolu izmantošana, kas, savukārt, prasīs kādu grafisko lietotāja vidi uz attiecīgā servera - piemēram X11 displeja serveri un LXDE, XFCE, GNOME vai citu darbvirsmas distribūciju un atbilstošo programmatūru (datņu pārvaldniekus, grafiskās konfigurācijas programmatūru utt.).

Tāpēc viena no iespējamām pieejām ir realizēt risinājumu, kurš pārvaldei izmanto uz attālinātā servera pieejamo čaulu, kurā izsauc noteiktu komandu izpildi un kurš reaģē uz saņemto attiecīgo komandu izvadi. Atšķirībā no Ansible, šāds rīks nebūtu atkarīgs no Python klātbūtnes uz attālinātās sistēmas un neprasītu noteiktu paku vai programmatūras instalāciju uz tās. Tas ļautu šādam risinājumam darboties uz liela daudzuma dažādu distribūciju un vajadzētu vien saderīgu čaulu un nepieciešamās utilītas, kas var tikt izsauktas uz attālinātā servera (piemēram "sed" teksta datņu apstrādei, "systemctl" servisu pārvaldei utt.). Šāds rīks arī darbotos līdzīgi tam, kā darbības veiktu cilvēki, tātad, neliktu par serveru stāvokli domāt pilnībā deklaratīvā kontekstā, kā to cenšas piedāvāt Ansible rīks, tā vietā ļaujot veikt imperatīvu darbību kopumu, līdzīgi kā to atļauj veikt Salt komandrindas funkcionalitāte.

Tāpat, tas neizslēdz iespēju izmantot Python vai citu augstāka abstrakcijas līmeņa valodu pašas uzdevumu loģikas implementācijai, taču tas tiktu darīts tikai paša rīka izpildvidē, kas būtu uz galvenā servera (1.4. att.), savukārt uz pārvaldāmo serveri tiktu nosūtītas tikai iepriekšminētās komandas.



1.4. att. Izstrādātā rīka darbības paraugs

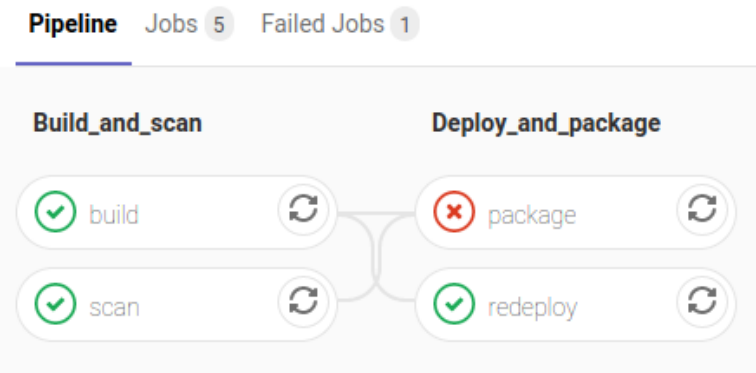
Pie iespējamiem šādas pieejas trūkumiem var minēt salīdzinoši zemāko ātrdarbību, ko varētu ieviest nepieciešamība gaidīt, kamēr komandas izpildes rezultāti tiktu pārsūtīti atpakaļ uz galveno serveri caur tīklu, kā arī iespējamo atkarību no sistēmas atbalstprogrammatūras - piemēram, ne visas operētājsistēmas kas izmanto Linux izmantos "systemd" pakotni servisu pārvaldei, piemēram, Alpine Linux izmanto OpenRC [6], kurai nederēs "systemd" domātie servisu apraksti). Šāda rīka veiktās darbības būtu arī vieglāk auditēt, jo serverim tās izskatītos pēc parasta sistēmas lietotāja darbībām, tātad, pēc noklusējuma parādītos attiecīgās čaulas vēsturē (piemēram, ".bash_history" datnē).

Maģistra darba ietvaros tika nolemts izstrādāt šādu rīku, tajā realizēt darbības, kas ļautu gan veikt dažādus serveru administratīvos uzdevumus, tostarp programmatūras atjauninājumu instalēšanu un dažādu servisu nokonfigurēšanu (piemēram, Zabbix monitoringa), gan arī sagatavot vidi konteinerizētu lietotņu palaišanai, kā arī inicializēt konteineru klasterus Docker Swarm un Kubernetes tehnoloģijās, kā arī veikt citas darbības.

1.3 Mijiedarbība ar rīku caur komandrindas interfeisu

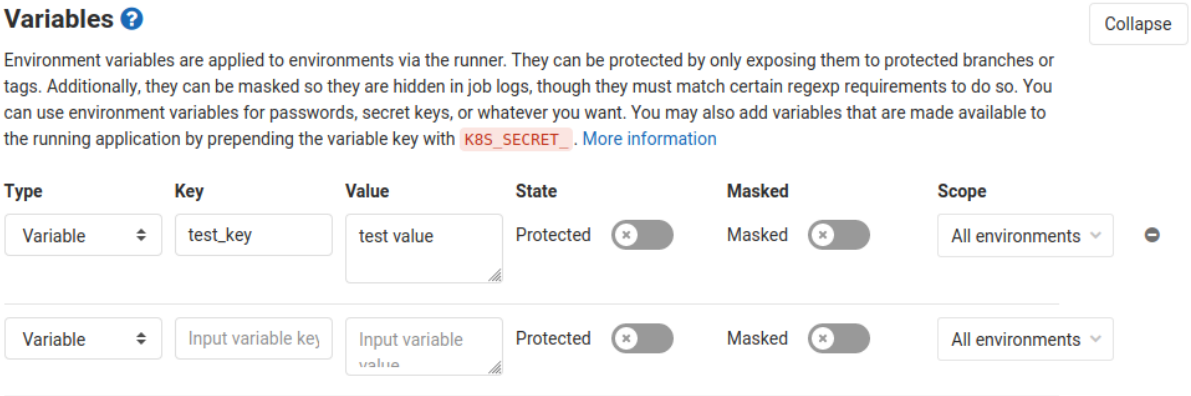
Lietotājam visdraudzīgākā varētu būt lietotne ar grafisko interfeisu, jo tajā būtu iespēja attēlot informāciju par sistēmas stāvokli grafiskā veidā, tā ļaujot viegli parādīt vispārīgu

pārskatu, atļaujot pēc vajadzības apskatīt detaļas tuvāk. Šāda pieeja arī ir sastopama vairākos nepārtrauktās integrācijas un nepārtraukto piegāžu rīkos, piemēram, GitLab CI/CD (1.5. att.).



1.5. att. Paraugs Apturi COVID mājaslapas CI/CD procesam priekš https://covid.kronis.dev izstrādes vides

Līdzīgi, šādi rīki bieži vien atļauj arī norādīt informāciju teksta laukos, tā ļaujot aizpildīt formu laukus, ne domāt par to, kā pareizi padot parametrus komandrindas lietotnei. Tajā pat laikā, tas var atļaut arī ar citu grafisko ievades elementu palīdzību norādīt papildus opcijas, kuras var tikt izsauktas (1.6. att.).



1.6. att. Paraugs GitLab CI/CD mainīgo norādīšanai

Tomēr, šādai pieejai ir arī vairāki trūkumi. Grafiskā interfeisa realizācija ievērojami var sarežģīt to, cik viegli noteiktu rīku būs integrēt kā daļu no jau esošas darbplūsmas. Tīkla lietotņu gadījumā pastāv iespēja izsaukt API piekļuves punktus (angliski "endpoints") pa tiešo, bet ja runa ir par darbvirsmas lietotni, piemēram GTK tehnoloģijā izveidotu, tad bieži vien šādas iespējas vienkārši nav, kas ievērojami traucē rīka darbības automatizācijā.

Tāpēc sākotnējai rīka izveidei tika izvēlēts realizēt komandrindas interfeisu, kurš ļautu rīka funkcionalitāti viegli izsaukt arī CI/CD kontekstā, kā arī neradītu atkarību no grafiskās vides klātbūtnes uz galvenā servera. Savukārt pretēji Ansible izvēlētajai pieejai, veicamās darbības netiek glabātas YAML formātā, tā vietā ļaujot līdzīgāk Salt realizācijai izsaukt komandu izpildi pa vienai ar komandu un to argumentu palīdzību. Līdzīgi Docker CLI, pašas rīka nodrošinātās komandas tiek organizētas grupās (1.7. att.), piemēram, tās kas ir paredzētas administrējamo serveru (un to SSH atslēgu) pievienošanai un dzēšanai ir pieejamas zem "host" apakškomandas, savukārt tās, kas ir paredzētas uzdevumu pārvaldei ir zem "task", kam atkal var būt citas apakškomandas - tāpēc uzdevuma izpildei izmantotu "task run", kā redzams paraugā zemāk.

```
(venv) kronislv@catbook:~/Documents/kvps5_masters_degree_astolfo_cloud_servant/src$ python astolfo.py --help
Usage: astolfo.py [OPTIONS] COMMAND [ARGS]...

  A CLI application to allow executing tasks on remote servers over SSH.
  Intended as a simpler alternative to Ansible and Salt.

  Most of the output that you'll be able to consume will be output as JSON
  and will be formatted in GREEN to STDOUT while in the terminal, whereas
  other output will be sent to STDERR, where errors will be red.

  Use the --help option with any of the subcommands for more info, or just
  run them without any additional arguments.

Options:
  --install-completion  Install completion for the current shell.
  --show-completion     Show completion for the current shell, to copy it or
                        customize the installation.

  --help                Show this message and exit.

Commands:
  file  Will allow you to view, add and remove files that you'll be able to...
  host  Will allow you to modify the hosts that tasks can be executed on.
  info  Will output information about the script and libraries.
  task  Will allow you to view, add and new tasks that will be executed...
(venv) kronislv@catbook:~/Documents/kvps5_masters_degree_astolfo_cloud_servant/src$
```

1.7. att. Komandu grupēšana rīkā

Ja nepieciešams paredzēt vairāku komandu secīgu izpildi un to izvades padošanu no vienas komandas otrā, tad pastāv iespēja šos izsaukumus piefiksēt Bash skriptos, vai jebkurā citā programmēšanas valodā, kas ļauj izsaukt čaulas komandas, tā neliekot apgūt specifisko rīka izmantoto formātu un tam vajadzīgos parametrus. Tam vajadzētu ļaut rīku integrēt vieglāk arī esošajās darbplūsmās, tostarp jau pastāvošajos čaulas skriptos, ja šāda nepieciešamība radīsies. Piemēram, sekojošais čaulas skripts uzinstalē Docker uz serveriem, inicializē Docker Swarm klasteri uz viena no tiem un pievieno pārējos serverus klasterim:

```
#!/bin/bash
set -e
source venv/bin/activate

# Install Docker
python astolfo.py task run docker 1.worker.catboi.net
python astolfo.py task run docker 2.worker.catboi.net
python astolfo.py task run docker db.catboi.net

# Create a Docker Swarm cluster
JOIN_TOKENS=$(python astolfo.py task run docker_swarm_cluster
1.worker.catboi.net)
python astolfo.py task run docker_swarm_worker 2.worker.catboi.net
"$JOIN_TOKENS"
python astolfo.py task run docker_swarm_worker db.catboi.net
"$JOIN_TOKENS"
```

Šādi izsaukumi gan būs atkarīgi no skriptam nepieciešamo paku klātbūtnes, kas šajā gadījumā nozīmē Python virtuālās vides nodrošināšanas pakas "venv" izmantošanu, lai neliktu rīkam nepieciešamās pakas ar "pip" rīku instalēt globāli. Paša rīka realizācija šajā gadījumā ir bāzēta uz Typer ietvaru [7], kurš ir speciāli paredzēts komandrindas lietotņu izveidei un atļauj komandas piereģistrēt ar dekoratoru palīdzību, savukārt pašu komandu izsaukumi atbilst metožu izsaukumiem ar noteikta tipa parametriem, kas gan izveido atkarību uz jaunākajām Python versijām kas nodrošina tipu sistēmu, taču papildus tam atvieglo izstrādi.

Lai realizētu augstāk esošo komandu hierarhiju, kurā rīkā izsauc uzdevuma izpildi, bija nepieciešama vairāku Typer objektu izveide un organizācija noteiktā struktūrā. No sākuma, tiek izveidots galvenais lietotnes objekts, kura definīcijā tiek norādīts arī tās apraksta ziņojums, kurš tiks izvadīts, vai nu izsaucot palīdzību, vai arī izsaucot rīku bez norādītas komandas:

```
app = typer.Typer(help="""
A CLI application to allow executing tasks on remote servers over SSH.
Intended as a simpler alternative to Ansible and Salt.\n
Most of the output that you'll be able to consume will be output as JSON
and will be formatted in GREEN to STDOUT while in the terminal, whereas
other output will be sent to STDERR, where errors will be red.\n
Use the --help option with any of the subcommands for more info, or just
run them without any additional arguments.
""")
```

Pēc tam šai lietotnei tiek pievienotas apakšlietotnes, piemēram, lietotne uzdevumu pārvaldīšanai. Tai, savukārt, var pievienot citas apakšlietotnes, šajā gadījumā, uzdevumu izsaukšanai:

```
task_app = typer.Typer(help="Will allow you to view, add and new tasks
that will be executed against the hosts. Use the --help option with any of
the subcommands for more info.")
app.add_typer(task_app, name="task")
...
task_run_app = typer.Typer(help="Will allow you to run specific tasks that
have been loaded dynamically from the tasks subdirectory.")
task_app.add_typer(task_run_app, name="run")
```

Kad ir izveidota šāda struktūra, tālāk attiecīgajām lietotnēm var pievienot komandas - metodes, kuras tiks izsauktas pēc noteikto apakšlietotņu un attiecīgās komandas ievadīšanas rīkā caur komandrindas interfeisu:

```
@task_run_app.command("ping")
def run(host: str = typer.Argument(..., help="The node to connect to.")):
    ... implementācija
```

Tas rezultē sekojošajā struktūrā: "task run ping", kurā pirmie divi ieraksti ir apakšlietotnes, savukārt pēdējais atsaucas uz Python valodā realizēto komandas implementāciju. Savukārt, ja tiks izsaukta apakšlietotne bez komandas, tad tiks parādīts tās palīdzības ziņojums, kā arī zem tās pieejamās piereģistrētās lietotnes un komandas, ar sākotnējo daļu no to palīdzības ziņojumiem (1.8. att.).

```
(venv) kronislv@catbook:~/Documents/kvps5_masters_degree_astolfo_cloud_servant/src$ python astolfo.py task run
Usage: astolfo.py task run [OPTIONS] COMMAND [ARGS]...

  Will allow you to run specific tasks that have been loaded dynamically
  from the tasks subdirectory.

Options:
  --help  Show this message and exit.

Commands:
  directory_create          Will create a directory on the host that is...
  directory_delete          Will recursively delete the directory that's...
  disk_usage                This task will check the disk usage on the...
  docker                    This task will install docker and...
  docker_swarm_cluster      This task will create a new Docker Swarm...
  docker_swarm_label        This task will join a given cluster as a
                            worker...
```

1.8. att. Apakšlietotnes izsaukšana bez komandas

Šāda pieeja ļauj viegli realizēt lietotājam draudzīgu un loģisku komandu struktūru, katrā no soļiem nodrošinot palīdzības ziņojumu izvadi, kā arī jebkurai komandai ļaujot uzreiz parādīt piereģistrētās apakškomandas vai apakšlietotnes, tostarp, arī realizēt programmatisku

jaunu komandu un lietotņu piereģistrēšanu, ja tas nepieciešams. Tas nozīmē, ka lai sāktu lietot rīku, nav obligāti nepieciešams apskatīt dokumentāciju, lai saprastu, kādas komandas ir pieejamas.

1.3.1 Palīdzības sistēma

Lietotnes izvadītā palīdzība netiek uzturēta atsevišķās datnēs, kuras būtu nodalītas no paša lietotnes koda, piemēram XML formāta datnēs, taču tā vietā šie palīdzības ziņojumi tiek uzģenerēti dinamiski, no paša koda komentāriem, vai arī Typer nodrošināto objektu lauku vērtībām. Piemēram, sekojošais koda fragments:

```
@task_run_app.command("ping")
def run(host: str = typer.Argument(..., help="The node to connect to.")):
    """
    This task will ping the selected node and check whether it can be
connected to through SSH.
    It will also check whether sudo works without password, if the account
is not root.
    """
    try:
        ... paša uzdevuma izpildes realizācija
    except Exception as e:
        ... kļūdu apstrādes realizācija
```

ļāva realizēt izvadi (1.9. att.), kura lietotājam parādīs gan komandai nepieciešamos parametrus ar informāciju par tiem, kā arī parādīs paša koda komentāra saturu, tā nodrošinot tiešu sasaisti starp koda dokumentāciju un izvadi sistēmas lietotājam, kas var kalpot par alternatīvu citām pieejām. Šajā gadījumā koda komentāriem ir ne tikai aprakstošā jēga, kas palīdzēs citiem izstrādātājiem izprast koda jēgu, bet arī sistēmas lietotājiem palīdzēs to lietot.

```
(venv) kronislv@catbook:~/Documents/kvps5_masters_degree_astolfo_cloud_servant/src$ python astolfo.py task run ping --help
Usage: astolfo.py task run ping [OPTIONS] HOST

  This task will ping the selected node and check whether it can be
  connected to through SSH. It will also check whether sudo works without
  password, if the account is not root.

Arguments:
  HOST  The node to connect to.  [required]

Options:
  --help  Show this message and exit.
(venv) kronislv@catbook:~/Documents/kvps5_masters_degree_astolfo_cloud_servant/src$
```

1.9. att. Komandas palīdzības izsaukšanas paraugs

Šāda veida funkcionalitāte kā arī koda komentāri, tika izveidoti visām publiski pieejamām komandām, lai atvieglotu izstrādātā rīka lietošanu.

1.3.2 Argumentu padošanas formāts

Viena no problēmām tradicionālo komandrindas lietotņu izmantošanā var būt datu formāts, ar kuru ir jāstrādā - it īpaši GNU/Linux operētājsistēmas distribūcijās bieži vien komandrindas lietotnes izvadi organizē kā tekstu, kuram mēdz piemērot noteiktu formatējumu. Tas sarežģī šo lietotņu izvades interpretēšanu un apstrādāšanu, jo parasti šāda komandu izvade ir domāta tam, lai lietotāji to varētu viegli salasīt un interpretēt, nevis lai tā būtu pēc iespējas vieglāk nolasāma citiem rīkiem un programmēšanas tehnoloģijām.

Piemēram, "docker ps" komanda izvada informāciju par uz sistēmas dotajā brīdī palaistajiem konteineriem, par tiem izvadot to identifikatorus, attēlu no kuriem tie ir izveidoti, komandu kas tajos ir palaista, kā arī informāciju par publicētajiem portiem un to statusu. Ierobežoto termināla emulatora vides iespēju dēļ nav iespējams tos attēlot īstas tabulas veidā, kā dēļ kas līdzīgs tiek ģenerēts tikai ar teksta palīdzību (1.10. att.).

```
[root@1 ~]# docker ps
CONTAINER ID   IMAGE                                                                COMMAND                  CREATED      STATUS      PORTS
8061b64bf869   osixia/phpldapadmin:0.9.0                                            "/container/tool/run"    3 days ago   Up 3 days   80/tcp, 443/tcp
b7ce7934d38c   osixia/openldap:1.4.0                                                "/container/tool/run"    3 days ago   Up 3 days   389/tcp, 636/tcp
c24c45e7c94c   registry.kronis.dev/rtu1/kvps5_masters_degree_covid_1984/covid1984:latest   "/bin/sh -c 'bash -c…"   3 days ago   Up 3 days   3000/tcp
21b3cc2f6ebe   registry.kronis.dev/rtu1/kvps5_masters_degree_covid_1984/covid1984:latest   "/bin/sh -c 'bash -c…"   3 days ago   Up 3 days   0.0.0.0:80->3000/tcp
7e91727fde54   portainer/portainer-ce:latest                                        "/portainer -H tcp:/…"   3 days ago   Up 3 days   8000/tcp, 9000/tcp
1ba1b06052c0   portainer/agent:latest                                               "./agent"                3 days ago   Up 3 days
[root@1 ~]#
```

1.10. att. Paraugs "docker ps" komandas izvadei

Kā redzams, izvade tiek vizuāli izlīdzināta (izņemot situācijas, kad izvade ir garāka par pieejamās rindas platumu), taču nav iespējams skaidri noteikt, kur beidzas un sākas noteiktas datu vienības, ja tiek runāts par šīs informācijas apstrādi kādas citas programmas ietvaros.

Piemēram, izvade zem komandas kolonnas ir ietverta ar "" simboliem, taču tajā pat laikā izvade zem konteinera izveides un statusa kolonnām šādi atdalīta netiek. Dažādo datu vienību atdalīšanai tiek izmantotas atstarpes, savukārt tās parādās arī pašu kolonnu teksta vidū. Tas nozīmē, ka pēc atstarpēm kā tādām filtrēt izvadi nav ieteicams, šajā gadījumā to varot mēģināt darīt tikai zinot, ka starp datu vienībām ir sagaidāmas tieši 3 atstarpes. Taču pat tad, ja mainīsies šis datu izvades formāts ar kādu rīka atjauninājumu, tad visi skripti, kuri paļaujas uz šo 3 atstarpju vai citu formatējuma īpatnību klātbūtni, vairs nebūs spējīgi veikt savu darbību. Tas nozīmē, ka šādu rīku izvadi vai nu nedrīkst izmantot kā ievadi citiem skriptiem, vai arī pašam rīkam nebūs iespējams nākotnē ieviest izmaiņas, piemēram, lai uzlabotu šīs tabulas attēlojumu.

Šo problēmu var mazināt mēģinājumi formatēt izvadi specifiskos veidos, piemēram, ar "docker ps --format "{{.ID}} {{.Status}} {{.Image}}"" komandu (1.11. att.), tā pieprasot noteiktu sev interesējošo datu atgriešanu, taču ne tuvu visa komandrindas programmatūra un rīki nodrošina šādas formatēšanas opcijas un arī pat ja tiktu ieviesti atdalošie simboli pēc kuriem izvadīto lauku saturu atšķirt, tāpat būtu jābūt drošiem, ka šie simboli pašā izvadē neparādīsies.

```
[root@1 ~]# docker ps --format "{{.ID}} {{.Status}} {{.Image}}"
8061b64bf869 Up 3 days osixia/phpldapadmin:0.9.0
b7ce7934d38c Up 3 days osixia/openldap:1.4.0
c24c45e7c94c Up 3 days registry.kronis.dev/rtu1/kvps5_masters_degree_covid_1984/covid1984:latest
21b3cc2f6ebe Up 3 days registry.kronis.dev/rtu1/kvps5_masters_degree_covid_1984/covid1984:latest
7e91727fde54 Up 3 days portainer/portainer-ce:latest
1ba1b06052c0 Up 3 days portainer/agent:latest
[root@1 ~]#
```

1.11. att. Paraugs "docker ps" komandai ar formatēšanas opcijām

Savukārt mēģinājumi ieviest atdalītājsimbolus un tos, kas parādās tekstā, papildināt ar atsoļa sekvencēm (angliski "escape sequence") tālāk sarežģītu programmatisku piekļuvi atgriezto datu laukiem, kā arī padarītu šīs izvades izmantošanu citos skriptos grūtāku. Maģistra darba ietvaros izstrādātajā sistēmā tika izmantota cita pieeja. Uz Windows operētājsistēmas tiek izmantota PowerShell čaula, kura atļauj darboties nevis tikai ar teksta veida izvadi un ievadi, bet gan ar objektiem, kuriem ir lauki ar vērtībām [8]. Lai gan PowerShell ir pieejama arī uz vairākām Linux distribūcijām, šīs tehnoloģijas izmantošana zināmā mērā ierobežotu integrācijas iespējas ar citiem rīkiem un pieprasītu arī .NET Core izpildvides instalāciju uz galvenā servera, papildus Python, tā sarežģījot arhitektūru, tomēr tās pieeja skriptu izveidei šķiet ērtāka - spēt nodot objektus no vienas komandas uz otru.

Tomēr, šajā gadījumā tika izdarīts lēmums izmantot pieeju, kas līdzīgi PowerShell darbojas ar objektiem, bet tos reprezentē teksta formātā - izmantot JSON (JavaScript Object Notation [9]) datu padošanai komandām. Šāda izvēle tika izdarīta gan tāpēc, ka ilgtermiņā ir paredzēts šo izstrādāto rīku papildināt ar tīkla interfeisu, kuram JSON datu formāts varētu noderēt, kā arī tāpēc, ka Python atbalsta objektu serializāciju un deserializāciju JSON formātā ar pickle un jsonpickle bibliotēkām [10]. Tas ļauj teksta formātā izveidot objektus ar parametriem, kuri ir nepieciešami kādas komandas izpildei, savukārt Python skripta ietvaros tos nolasīt un ar tiem darboties kā Python objektiem.

Pašā sākumā, nepieciešams definēt attiecīgās metodes objektiem, piemēram, komandai kura uzinstalē un nokonfigurē klientu Zabbix monitoringa rīkam uz attiecīgā servera, lai tas atļautu serverim pieprasīt datus par sistēmas stāvokli:

```
import jsonpickle
jsonpickle.set_preferred_backend('demjson')
class ZabbixAgentRequest(object):
    def __init__(self, server_address):
        self.server_address = server_address
    @classmethod
    def read(self, input: str):
        return jsonpickle.decode(input)
    def dump(self):
        return jsonpickle.encode(self, indent=2)
```

Šajā gadījumā dots paraugs pavisam vienkāršam objektam ar vienu lauku, taču līdzīgos objektos var tikt iekļauti vairāki lauki, vai pat citi objekti. Rezultātā, izveidojot objektu un tam izsaucot "dump" metodi, tiks izvadīts sekojošais teksts, kurš reprezentē objektu JSON formātā:

```
{
  "py/object": "tasks.task_zabbix_agent.ZabbixAgentRequest",
  "server_address": ""
}
```

Kā redzams šeit, tika pievienots "py/object" lauks, kurš piefiksē no kādas Python klases ir izveidots attiecīgais objekts. Taču lietotājam varētu būt problemātiski uzminēt, kāds tieši saturs ir nepieciešams šim laukam, vai kā vispār formatēt ievadi un tās laukus noteiktai komandai, ja tie ir nepieciešami. Tāpēc rīka ietvaros tika izveidota papildus komanda, kura atgriež tukšus argumenta objektus, kurus atliek tikai aizpildīt un padot īstajai komandai uz izpildi. Piemēram, izpildot "task arguments zabbix_agent" komandu, tiks izsaukts sekojošais kods:

```
@task_arguments_app.command("zabbix_agent")
def argument_info():
    argument = ZabbixAgentRequest("")
    typer.echo(typer.style(argument.dump(), fg=typer.colors.GREEN))
```

kurš radīs jaunu objektu ar tukšiem laukiem un to izvadīs teksta formātā, līdzīgi paraugam augstāk. Tas ļauj zināmā mērā kompensēt neērtības, kas varētu rasties gadījumā, ja JSON objektus nepieciešams izveidot manuāli, kā arī atļaut uzreiz pateikt, kādi parametri ir nepieciešami un kādā datu struktūrā tie tiek sagaidīti.

1.3.3 Izvades organizēšana

JSON objektu izmantošanas pieeja atļauj tos saņemt kā izvadi no kādas komandas un padot tos tālāk citām kā ievadi. Piemēram, izveidojot Docker Swarm klasteri, tiek iegūta atslēga, kuru tālāk ir iespējams izmantot, lai citi serveri varētu tam pievienoties un uz tiem varētu izvietot konteinerus, kas tika demonstrēts iepriekš, 1.3. nodaļas ievadā. Šajā gadījumā pati izvade pēc veiksmīgas Docker Swarm klastera izveides var izskatīties sekojoši:

```
{
  "py/object":
"tasks.task_docker_swarm_cluster.DockerSwarmCreateResponse",
  "public_ip": "176.223.139.9",
  "swarm_created": true,
  "swarm_join_token_worker": "SWMTKN-1-
1py6dffxukbi1ywylu7ppn103pk24qvpega2wgtsd8vg2jct2i-
4pk63hlxm3p5r3543d2vk0ac8 176.223.139.9:2377",
  "swarm_join_token_master": "SWMTKN-1-
1py6dffxukbi1ywylu7ppn103pk24qvpega2wgtsd8vg2jct2i-
eit6eyadhm7mjfpnq2bqhmkvr 176.223.139.9:2377"
}
```

Šo izvadi ir iespējams uzglabāt mainīgajā, piemēram iekš Bash skripta, vai arī pa taisno padot tālāk uz komandu, kura ar šīs ievades palīdzību pievienos serveri izveidotajam klasterim. Tas atļauj padot starp komandām sarežģītākus objektus un vairākas datu vienības, pašā CLI līmenī izmantojot vienu teksta tipa argumentu, no kura tālāk var nolasīt un iegūt attiecīgo Python objektu.

JSON formāts atļauj arī izvairīties no neērtās situācijas, kuru radītu YAML formāta (YAML Ain't Markup Language [11]), vai citu tam līdzīgo formātu, atkarība no jaunu rindu un tukšumzīmju izmantošanas lai radītu korektu datu reprezentāciju. Ja komandu argumentu ievadei vajadzētu izmantot YAML formātu, tad terminmāla emulatorā būtu nepieciešams ievietot jaunu rindu simbolus, vai arī visu ievadi veikt no STDIN standartievades kanāla. Savukārt JSON gadījumā datus ir iespējams ievadīt, visu objektu noformējot vienā rindā, bez jaunu rindu simboliem, savukārt lietotājam datus izvadīt lasāmākā formātā, ar daudzu rindu izvadi, katrā rindā attēlojot vienu lauku ar tā vērtību. Šī pieeja ļauj sasniegt kompromisu starp komandrindas lietojamību un izvadīto datu pārskatāmību.

Viens no JSON trūkumiem ir nespēja pievienot datus skaidrojošus komentārus, kā gadījumā būtu jāizmanto JSON5 [12], taču šis formāts netiek plaši atbalstīts.

1.4 Izvade par izpildes stāvokli

Ja rīka komandu izvadi ir nepieciešams padot tālāk citām komandām, tad šajā situācijā problēmas var radīt jebkura papildus izvade, kas nav saistīta ar komandas rezultātiem - rīks izvada ne tikai minētos rezultātu objektus, bet arī informāciju par izpildes laikā veiktajām darbībām, piemēram par to, kādas komandas tiek izpildītas caur SSH un kāda ir to izvade uz attālinātā servera (1.12. att.).


```
Īnstalling Zabbix agent on: 1.worker.catboi.net
Setting Zabbix server to: ram.servers.kronis.eu
Connecting to host: 1.worker.catboi.net
Executing: rpm -qa | grep -qw zabbix-release-3.2-1.el7.noarch || rpm -ivh http://repo.zabbix.com/zabbix/3.2/rhel/7/x86_64/zabbix-release-3.2-1.el7.noarch.rpm
Executing: rpm -qa | grep -qw zabbix-agent || yum -d1 install -y zabbix-agent
Executing: if [ ! -f /etc/zabbix/zabbix_agentd.conf.bak ]; then cp /etc/zabbix/zabbix_agentd.conf{,.bak}; else echo 'Backup file already exists!'; fi
Executing: sed -i '/Server=127.0.0.1/c\Server=ram.servers.kronis.eu' /etc/zabbix/zabbix_agentd.conf
Executing: systemctl enable zabbix-agent
Executing: service zabbix-agent restart && systemctl --no-pager status zabbix-agent
Executing: tail -n 32 /var/log/zabbix/zabbix_agentd.log; echo 'No logs exist yet!'
Stages: {
  "AddRepo": {
    "py/object": "tasks.utils.stages.Stage",
    "name": "AddRepo",
    "description": "Will add the repository for Zabbix so it can be downloaded",
    "order": 0,
    "successful": true,
    "command": "rpm -qa | grep -qw zabbix-release-3.2-1.el7.noarch || rpm -ivh http://repo.zabbix.com/zabbix/3.2/rhel/7/x86_64/zabbix-release-3.2-1.el7.noarch
pm",
    "output": [
      "Retrieving http://repo.zabbix.com/zabbix/3.2/rhel/7/x86_64/zabbix-release-3.2-1.el7.noarch.rpm",
      "warning: /var/tmp/rpm-tmp.WyBF2Z: Header V4 RSA/SHA512 Signature, key ID a14fe591: NOKEY",
```


1.12. att. Paraugs izvadei par komandām, kuras tika izpildītas

Šī izvade gan nedrīkst konfliktēt ar atgrieztajiem JSON objektiem, tāpēc visa lietotājam domātā izvade nokļūst STDERR izvades kanālā, savukārt visa izvade, kura ir paredzēta padošanai tālāk uz citām komandām nonāk STDOUT izvades kanālā. Lai palīdzētu labāk saprast, kura izvade tiek attēlota noteiktās rindiņās, tiek izmantotas iespējas kas ļauj iekrāsot izvadīto tekstu, kļūdas iekrāsojot sarkanā krāsā, savukārt komandām domāto izvadi iekrāsojot zaļā krāsā (1.13. att.).


```
    "finished": true
  }
}
{
  "py/object": "tasks.task_ping.PingResponse",
  "host_pinged": "1.worker.catboi.net",
  "host_user": "root",
  "connection_successful": true,
  "user_can_sudo": false
}
(venv) kronislv@catbook:~/Documents/kvps5_masters_degree_astolfo_cloud_servant/src$
```


1.13. att. Paraugs izvades iekrāsošanai atkarībā no tipa

Tomēr tā nav vienīgā JSON formāta izvade - tiek izvadīta arī informācija par izpildītajiem soļiem uzdevuma ietvaros, no kā sastāv katrs uzdevums. Tā ļauj sīkāk pateikt, kā veicās uzdevuma izpildē un gadījumā ja notika kāda kļūda, tad identificēt, tieši kurā solī tā

radās. Katram no soļiem var atbilst informācija par to, vai tā izpilde bija veiksmīga, komanda kura soļa ietvaros tika izpildīta uz attālinātā servera, kā arī izvade kas tika saņemta pēc tās izpildes, kas tiek definēts sekojoši:

```
class Stage(object):
    def __init__(self, name, description, order = 0, successful = False,
command = "", output = "", finished = False):
        self.name = name
        self.description = description
        self.order = order
        self.successful = successful
        self.command = command
        self.output = output
        self.finished = finished
    @classmethod
    def dump_all(self, input: List):
        return jsonpickle.encode(input, indent=2)
    def dump(self):
        return jsonpickle.encode(self, indent=2)
```

Līdzīgi kā tika izveidotas komandas argumentu izvadīšanai, tāpat katram uzdevumam var iegūt arī sagaidāmās izpildes stadijas, lai pēc tam varētu pārskatīt, kā tās ir tikušas izpildītas:

```
def get_stages():
    return build_stages([
        Stage("AddRepo", "Will add the repository for Zabbix so it can be
downloaded"),
        Stage("InstallPackage", "Will install the zabbix-agent package
through yum"),
        Stage("ConfigBackup", "Will back up the default configuration
file"),
        Stage("ServerUpdate", "Will change the Zabbix server URL in the
configuration file"),
        Stage("SystemctlEnable", "Will enable the Zabbix service through
systemctl"),
        Stage("ServiceRestart", "Will restart the service"),
        Stage("ViewLogs", "Will view the logs to make sure that everything
has been installed and launched successfully")
    ])

@task_stages_app.command("zabbix_agent")
def stage_info():
    stages = get_stages()
    typer.echo(typer.style(Stage.dump_all(stages), fg=typer.colors.GREEN))
```

Šī datu struktūra tiek aizpildīta noteiktā uzdevuma izpildes laikā, atkarībā no SSH komandu rezultātiem. Pat ja uzdevuma izpildes laikā rastos kāda kļūda, tad pēc kļūdas

apstrādes tāpat tiktu izvadīts pārskats par visiem soļiem. Piemēram, ja kādam uzdevumam izdosies servisu pārstartēt veiksmīgi, tad tas parādīsies arī soļu pārskatā pēc izpildes:

```
# We restart the service and make sure that it's working
command = "service zabbix-agent restart && systemctl --no-pager status
zabbix-agent"
error_code, output = ssh_execute(ssh_connection, command)
if not error_code:
    complete_stage(stages, "ServiceRestart", successful=True,
command=command, output=output)
    command_output.service_restart = True
else:
    complete_stage(stages, "ServiceRestart", successful=False,
command=command, output=output)
    raise Exception("Failed to restart the service!")
```

Šeit metode "complete_stage" samaina uzdevuma izpildes sākumā izveidoto soļu saraksta ierakstu ar atbilstošo tipu "ServiceRestart". Ja notiktu kļūda un šī metode netiktu izsaukta, tad pēc noklusējuma izpildi uzskatītu par neuzsāktu, tātad, neveiksmīgu.

Pēc uzdevuma izpildes STDOUT kanālā tiks izvadīts pilnais izpildes rezultātu pārskats JSON formātā. Piemēram, augstāk minētās stadijas izpildes rezultātā objekts masīvā tiks aizpildīts ar attālinātā servera atgrieztajiem datiem (1.14. att.).


```
    },
    "ServiceRestart": {
      "py/object": "tasks.utils.stages.Stage",
      "name": "ServiceRestart",
      "description": "Will restart the service",
      "order": 5,
      "successful": true,
      "command": "service zabbix-agent restart && systemctl --no-pager status zabbix-agent",
      "output": [
        "Redirecting to /bin/systemctl restart zabbix-agent.service",
        "\u25cf zabbix-agent.service - Zabbix Agent",
        "   Loaded: loaded (8;;file://1.worker.catboi.net/usr/lib/systemd/system/zabbix-agent.service\u0007/usr/lib/sys
temd/system/zabbix-agent.service8;;\u0007; enabled; vendor preset: disabled)",
        "   Active: active (running) since Fri 2021-01-01 23:13:31 EET; 19ms ago",
        "  Process: 19686 ExecStart=/usr/sbin/zabbix_agentd -c $CONFFILE (code=exited, status=0/SUCCESS)",
        " Main PID: 19688 (zabbix_agentd)",
        "    Tasks: 6 (limit: 24036)",
        "   Memory: 4.3M",
        "   CGroup: /system.slice/zabbix-agent.service",
        "           \u251c\u250019688 /usr/sbin/zabbix_agentd -c /etc/zabbix/zabbix_agentd.conf",
        "           \u251c\u250019689 /usr/sbin/zabbix_agentd: collector [idle 1 sec]",
        "           \u251c\u250019690 /usr/sbin/zabbix_agentd: listener #1 [waiting for connection]",
        "           \u251c\u250019691 /usr/sbin/zabbix_agentd: listener #2 [waiting for connection]",
        "           \u251c\u250019692 /usr/sbin/zabbix_agentd: listener #3 [waiting for connection]",
        "           \u2514\u250019693 /usr/sbin/zabbix_agentd: active checks #1 [idle 1 sec]",
        "",
        "Jan 01 23:13:31 1.worker.catboi.net systemd[1]: Starting Zabbix Agent...",
        "Jan 01 23:13:31 1.worker.catboi.net systemd[1]: zabbix-agent.service: Can't \u2026ory",
        "Jan 01 23:13:31 1.worker.catboi.net systemd[1]: Started Zabbix Agent.",
        "Hint: Some lines were ellipsized, use -l to show in full."
      ],
      "finished": true
    },
    "ViewLogs": {
      "py/object": "tasks.utils.stages.Stage",
      "name": "ViewLogs",
```


1.14. att. Paraugs stadijas izpildes informatīvajai izvadei

Šāda pieeja ļauj izmantot rīku pa taisno no komandrindas, neliekot apskatīties atsevišķas audita datnes, savukārt vajadzības gadījumā šo izvadi ir iespējams pārvirzīt uz dažādām datnēm, lai šo izvadi uzglabātu ilgstoši. Protams, pastāv iespēja arī abus izvades kanālus pārvirzīt uz vienu un to pašu datņu pilnīgākai izvades apskatīšanai, piemēram, kā tas ir ilustrēts parauga skriptos:

```
#!/bin/bash
set -e

echo "This script will setup a Docker Swarm cluster with all of the
servers joined to the single master and then Portainer will also be
deployed, in addition to worker tags." \
&& ./stats.sh &>> ./logs/full-swarm-cluster-example.log \
&& ./base-install.sh &>> ./logs/full-swarm-cluster-example.log \
&& ./swarm-cluster.sh &>> ./logs/full-swarm-cluster-example.log \
&& ./stats.sh &>> ./logs/full-swarm-cluster-example.log
```

Tātad, par spīti ierobežotajām komandrindas piedāvātajām iespējām, ir iespējams izveidot rīku, kurš piedāvā izvadi, kas gan palīdz lietotājam saprast kādas komandas tiek izpildītas un vai tās tika izpildītas veiksmīgi, kā arī atļauj izvadi nodot tālāk citām komandām, ja tā ietver tām nepieciešamo informāciju, to nododot kā sarežģītus objektus ar laukiem, nevis tikai tekstu.

Šeit gan jāpiemin, ka POSIX saderīgo operētājsistēmu divu izvades kanālu ierobežojums ir visai traucējošs, jo faktiski lietotājam domātā izvade nav uzskatāma par kļūdām, savukārt to izvadīšana STDOUT kanālā rezultētu ar to, ka visām pārējām komandām būtu jāizstrādā mehānisms, kurš ignorētu šo izvadi. Tātad, katram ziņojumam vai objektam vajadzētu ieviest pazīmi, vai tas ir paredzēts tālākai nolasīšanai. Papildus tam, vienkārši lietotājam domāti ziņojumi arī nav JSON formātā, lai nesarežģītu izvades uztveri.

Tomēr, šeit izmantotā pieeja adekvāti apmierināja vajadzību pēc lietotājam draudzīgas saskarnes izveides, tajā pat laikā netraucējot arī rīka izvadi izmantot kā ievadi citiem skriptiem. Elegantāks risinājums gan būtu papildus izvades kanālu izmantošana:

- kanāls izvadei, kas ir paredzēta lietotājam (analoģiski STDOUT)
- kanāls izvadei, kas ir paredzēta citiem rīkiem un skriptiem
- kanāls kļūdu izvadei (analoģiski STDERR)

Taču pašlaik šādu funkcionalitāti vairums operētājsistēmu nenodrošina, kas arī sarežģītu rīku izvades standartizāciju un varētu radīt citus sarežģījumus.

1.5 Rīka funkcionalitātes realizācija

Viens no izstrādes mērķiem bija realizēt sistēmu, kura būtu gan viegli lietojama, gan viegli paplašināma. Šāda sistēma nedrīkstētu no izstrādātāja prasīt ilgu laiku tās apguvei un nedrīkstētu lieki sarežģīt jaunu uzdevumu rakstīšanu vai koda pielāgošanu savām vajadzībām. To varētu neļaut sasniegt sarežģīta jauno uzdevumu piereģistrēšanas funkcionalitāte, sarežģīts uzdevumu dzīves cikls, vai arī liels skaits prasību pret jaunizveidotajiem uzdevumiem.

Lai novērstu vismaz daļu šo problēmu, tika izvēlēts izstrādāt katru uzdevumu kā vienkāršu Python skripta datni, kurā uzdevums tiktu piereģistrēts ar jau iepriekšminēto Typer nodrošināto funkcionalitāti. Savukārt visas uzdevumu datnes glabājas vienuviet, zem "tasks" direktorijas, tāpēc uzreiz kļūst skaidrs, kur iespējams pievienot jaunus, atbilstoši rīka datņu struktūrai (1.15. att.). Papildus tam, katrs skripts ir pilnībā aprakstīts vienas datnes ietvaros, tikai pēc vajadzības izsaucot papildus funkcionalitāti, piemēram, lai no jauna nebūtu jāraksta SSH komandu izpildīšanas loģika. Tas ļauj jau esošos skriptus izmantot par paraugu jaunu izstrādei.

```
├── astolfo.py
├── cli
│   └── # šeit glabājas paša rīka funkcionalitātes kods
├── files
│   └── # šeit glabājas faili lejupielādei/augšupielādei
├── infrastructure
│   └── # šeit katram serverim ir sava mape ar informāciju par to
├── requirements.txt # informācija par pip atkarībām
├── tasks
│   ├── globals.py # visiem uzdevumiem pieejamie mainīgie
│   ├── __init__.py
│   ├── task_directory_create.py
│   ├── task_directory_delete.py
│   ├── # citi uzdevumu faili, kas seko task_*.py paraugam
│   └── utils
│       ├── __init__.py
│       └── # daudzviet izmantojamā, kopējā funkcionalitāte
└── venv
    └── # nepieciešamie koda bibliotēku faili
```

1.15. att Rīka datņu struktūras paraugs

Sarežģījumus Python izmantošanas gadījumā radīja valodas pieeja jaunu koda moduļu importēšanai, jo visa rīka dalītā funkcionalitāte glabājas "cli" mapē, savukārt uz to nepieciešams atsaukties no "tasks" mapes datnēm, tātad, izdarīt atsaukšanos uz ceļiem, kas ir analoģiski pāriešanai uz augstāku mapi un pēc tam uz citu tās apakšmapi, t.i. "../cli" ceļu.

Python to pēc noklusējuma neatbalsta, kā dēļ bija nepieciešams izmainīt attiecīgā skripta kontekstu, kā dēļ visiem skriptiem ievada daļā bija izmantota "sys.path.insert" darbība:

```
# needed to be able to reference sibling packages
import sys, os
sys.path.insert(0, os.path.abspath('..'))
# needed to register this task as a CLI command
import typer
from .globals import task_run_app
from .globals import task_stages_app
from .globals import task_arguments_app
# needed for serialization of data
import jsonpickle
jsonpickle.set_preferred_backend('demjson')
# utilities to reuse existing code
from tasks.utils.stages import *
from cli.host_actions import *
from tasks.utils.ssh import *
```

Rezultātā tas ļauj atsaukties gan uz moduļiem tekošajā direktorijā, piemēram ".globals", kas atļauj importēt visus Typer lietotņu objektus, tā atļaujot dinamiski piereģistrēt jaunos uzdevumus, gan arī atsaukties uz datnēm citās mapēs, piemēram, "cli.host_actions", kas atļauj darboties ar rīkā piereģistrētajiem serveriem, piemēram, nolasīt to pieslēgšanās datus, ar ko izveidot SSH savienojumu.

Pašu skriptu ietvaros var tikt aprakstītas datu struktūras izvadei un ievadei, kurus tie izmantos, piemēram, šeit tiek parādīts piemērs uzdevumam, kurš pārbaudīs diska stāvokli uz attālinātā servera:

```
class DiskUsageResponse(object):
    def __init__(self, disk_usage = ""):
        self.disk_usage = disk_usage
    def dump(self):
        return jsonpickle.encode(self, indent=2)

def get_stages():
    return build_stages([
        Stage("GetDiskUsage", "Will get the disk usage information")
    ])

@task_stages_app.command("disk_usage")
def stage_info():
    stages = get_stages()
    typer.echo(typer.style(Stage.dump_all(stages), fg=typer.colors.GREEN))
```

Kā redzams, ir arī loģika, kura ļauj izgūt informāciju par soļiem, kurus uzdevuma izpilde ietvers, kā atsevišķa komanda. Savukārt pēc tam seko galvenā realizācijas daļa, kas realizē pašu loģiku, kas tiks izpildīta uzdevuma izsaukšanas gadījumā:

```
@task_run_app.command("disk_usage")
def run(
        host: str = typer.Argument(..., help="The node to connect to.")
    ):
    """
    This task will check the disk usage on the remote node.
    """
    stages = get_stages()
    try:
        loaded_host = load_host(host)
        typer.echo("Checking disk usage on node: " +
str(loaded_host.host), err=True)
        command_output = DiskUsageResponse()
        ssh_connection = ssh_connect(loaded_host)

        # Get the disk usage on node
        command = "df -h *"
        error_code, output = ssh_execute(ssh_connection, command)
        if not error_code:
            complete_stage(stages, "GetDiskUsage", successful=True,
command=command, output=output)
            command_output.disk_usage = output
        else:
            complete_stage(stages, "GetDiskUsage", successful=False,
command=command, output=output)
            raise Exception("Failed to check disk usage!")

        typer.echo("Stages: " + Stage.dump_all(stages), err=True)
        typer.echo(typer.style(command_output.dump(),
fg=typer.colors.GREEN))
    except Exception as e:
        typer.echo(Stage.dump_all(stages))
        typer.echo(typer.style("Failed to check disk usage on host: " +
str(host) + " with exception: " + str(e), fg=typer.colors.RED), err=True)
        sys.exit(1)
```

Kā redzams, uzdevumi var ietvert arī vienkāršu kļūdu apstrādi un reaģēt uz caur SSH izpildīto komandu izvadi, gan apskatot pašu izvades tekstu standarta kanālos, gan arī apskatot izsaukto komandu kļūdas kodu, kas veiksmīgas izpildes rezultātā būs 0.

Šeit ir redzams arī paraugs Typer nodrošinātajam formatējumam un krāsām, kas atļauj pašam lietotājam uzskatāmāku padarīt izvadi un to, vai komandas izpilde ir bijusi veiksmīga, tā kompensējot komandrindas trūkumus, salīdzinot ar grafisko izvadi piktogrammās, pat terminālu emulatoros, kas neatbalsta emocijzīmes vai simbolus, piemēram "✔" un "✗".

1.5.1 Dinamiskā uzdevumu ielāde

Tika noteikts, ka arī pašu uzdevumu ielādei ir jābūt dinamiskai - nevajadzētu uzturēt pieejamo uzdevumu sarakstu atsevišķi, jo tas sarežģītu dažādu avotu uzdevumu reģistrēšanu, gadījumā ja tie būtu ievadīti vienā datnē. Piemēram, ja būtu "tasks.json" datne ar uzdevumu sarakstu, tad dažādiem avotiem pievienojot jaunus ierakstus tam, varētu rasties konflikti to starpā - ja pašu datni aizstātu, tad vienas izmaiņas varētu pārrakstīt otras, savukārt pretējā gadījumā būtu jāizstrādā mehānismi šādas datnes atjaunošanai.

Python pats par sevi atļauj moduļu importēšanu no skriptiem datņu sistēmā, kā dēļ tika izvēlēts uzrakstīt loģiku, kura izsauktu moduļu koda importēšanu, kas, savukārt, liktu izpildīties dekoratoriem piesaistītajai loģikai - piereģistrēt datnēs esošās komandas kā pieejamas komandrindas lietotnē. Tā sasniegšanai, pašā galvenajā skripta datnē ir kods, kurš tiek izpildīts katru reizi pēc rīka palaišanas, pirms tiek izpildīta pati komandrindas loģika:

```
def initialize_tasks():
    task_run_app = typer.Typer(help="Will allow you to run specific tasks
that have been loaded dynamically from the tasks subdirectory.")
    task_stages_app = typer.Typer(help="Will allow you to list the stages
available in the tasks.")
    task_arguments_app = typer.Typer(help="Will allow you to get
information about the JSON arguments that the tasks expect.")
    from tasks import globals
    globals.task_run_app = task_run_app
    globals.task_stages_app = task_stages_app
    globals.task_arguments_app = task_arguments_app
    task_files = glob(os.path.join("tasks", "task_*.py"))
    task_filenames = [os.path.split(item)[1].replace(".py", "") for item
in task_files]
    global loaded_tasks
    for module_name in task_filenames:
        try:
            exec("from tasks." + module_name + " import *")
            loaded_tasks += 1
        except Exception as e:
            typer.echo(typer.style("Failed to load task: " +
str(module_name) + " with error: " + str(e), fg=typer.colors.RED),
err=True)
    task_app.add_typer(task_run_app, name="run")
    task_app.add_typer(task_stages_app, name="stages")
    task_app.add_typer(task_arguments_app, name="arguments")
initialize_tasks()
```

Tātad, šajā gadījumā tiek izveidotas apakšlietotnes, kas atbildīs apakškomandām un tika saglabātas "globals" modulī, lai to tālāk varētu izmantot dekoratoriem.

Pēc tam tiek izgūts operētājsistēmas datņu saraksts zem "tasks" mapes, kuri beidzās ar .py datnes paplašinājumu un kuru nosaukums sākas ar "task_", tātad, tiek izgūti skriptu datņu nosaukumi. Pēc tam no šo datņu nosaukumiem tiek iegūti atbilstošo moduļu nosaukumi, nometot datnes ceļa direktorijas daļu, kā arī nodzēšot datnes paplašinājumu. Piemēram, "tasks/task_ping.py" tiek pārveidots par "task". Pēc tam tiek izmantota Python exec komanda [13], lai dinamiski izsauktu attiecīgā koda izpildi katrai no uzdevumu datnēm, tā ļaujot to dekoratoriem piereģistrēt lietotnes komandas. Piemēram, iepriekšminētā "task" moduļa iegūšana rezultē sekojošā koda izpildē:

```
from tasks.ping import *
```

Šeit gan papildus uzmanība ir jāpievērš kļūdu apstrādes loģikai, jo situācija, kurā slikti uzrakstīts uzdevums traucē visa rīka darbību nav pieļaujama. Tāpēc tiek izmantots "try...except" bloks, šajā gadījumā izvadot informāciju par kļūdām, ja tādas radās nolasot uzdevumu. Piemēram, ja uzdevuma ietvaros ir norādīta neeksistējoša paka, lietotājs par to saņems brīdinājumu (1.15 att.).

```
(venv) kronislv@catbook:~/Documents/kvps5_masters_degree_astolfo_cloud_servant/src$ python astolfo.py task --help
Failed to load task: task_broken_example with error: No module named 'something_that_does_not_exist'
Usage: astolfo.py task [OPTIONS] COMMAND [ARGS]...

  Will allow you to view, add and new tasks that will be executed against
  the hosts. Use the --help option with any of the subcommands for more
  info.
```

1.15. att. Bojāta uzdevuma ielādes izvades paraugs

Šāda pieeja ļauj viegli realizēt dinamisku jaunā koda inicializēšanu, taču jāņem vērā, ka ar to ir saistīti zināmi drošības riski - nav ieteicams uzticēties jebkuram koda avotam, taču rīka vienkāršā realizācija atļauj manuāli pārbaudīt katra importējamā uzdevuma darbību, jo tie ir ierobežoti vienas datnes ietvaros. Šīs pieejas trūkums gan ir tas, ka pagaidām netika realizēts mehānisms papildus ietvaru un koda bibliotēku ieviešanai projektā.

Pēc noklusējuma, ar "pip" paku pārvaldes rīku tiek lejupielādētas atkarības, kas ir nepieciešamas paša rīka darbībai, piemēram "Paramiko" bibliotēka, lai nodibinātu SSH savienojumu ar attālinātu serveri, bet jaunu paku instalēšanai ir nepieciešams pārbūvēt pašu rīku, kas konteineru gadījumā nozīmēs visa konteinera izveidošanu no jauna.

1.5.2 Administratīvās darbības

Papildus paša rīka uzdevumu izpildes funkcionalitātei, tika implementēta arī funkcionalitāte, kas ļauj administrēt informāciju par pieejamiem serveriem, pārvaldīt pašus uzdevumus, kā arī datnes. Tas tika darīts, jo lai gan šos datus var pievienot un mainīt datņu sistēmā, zem attiecīgajām direktorijām: "infrastructure", "tasks" un "files", konteineru gadījumā šie dati var glabāties konteineru diskvietās (angliski "volumes"), kurām nebūs pieeja pa taisno. Līdzīgi, šīs komandas atļauj arī izvadīt palīdzības paziņojumus, līdzīgi kā pašu uzdevumu izpildes loģika.

Par katru serveri tiek uzglabāta pamata informācija: tā adrese, apraksts, operētājsistēma, lietotājvārds ar kuru mēģināt izveidot SSH pieslēgumu, kā arī SSH atslēgas datnes nosaukums, kuru izmantot autentifikācijai:

```
{
  "py/object": "cli.host_data.Host",
  "host": "1.worker.catboi.net",
  "description": "Worker that will also be the cluster master",
  "os": "CentOS 8",
  "username": "root",
  "keyfile": "id_rsa"
}
```

Šī informācija glabājas "infrastructure" mapē, katram pārvaldāmajam serverim izveidojot savu mapi, piemēram, "infrastructure/1.worker.catboi.net/", kurā šī informācija ir uzglabāta JSON formātā datnē. Tiek nodrošinātas komandas gan jaunu serveru pievienošanai, gan dzēšanai, kas atļauj šos administratīvos uzdevumus veikt arī Docker gadījumā. Tad gan ir nepieciešams pievienot attiecīgās diskvietas vai sasaisti ar datņu sistēmas direktorijām pa tiešo (angliski "bind mount"), lai šīs pievienotās datnes tiktu saglabātas:

```
docker run \
-v astolfo_tasks:/app/tasks \
-v astolfo_infrastructure:/app/infrastructure \
-v astolfo_files:/app/files \
astolfo host add 1.worker.catboi.net "Application server that will run
front-end and back-end containers" "CentOS 8" root id_rsa
```

Pagaidām šīm komandām vēl netiek izmantots JSON formāts, taču idejiski nākotnē arī šo argumentu padošanu pa tiešo iespējams aizstāt ar JSON objektu. Šajā gadījumā gan ir nedaudz uzskatāmāks palīdzības teksta saturs, jo tajā katram minētajam lielumam viegli var

norādīt noklusējuma vērtību, piemēram, ka savienojums ar serveri var tikt izveidots ar "root" lietotāju, savukārt ka atslēgas datnes nosaukums atbildīs "id_rsa" (1.16. att.).

```
kronislv@catbook:/media/kronislv/6EEA-1A2A/Nextcloud/RTU/maģistra darbs$ docker run astolfo host add --help

Usage: astolfo.py host add [OPTIONS] HOST [DESCRIPTION] OS [USERNAME]
                           [KEYFILE]

  Will add a new host to the infrastructure folder.  This is much like the
  description of the hosts in Ansible, but for performance reasons some
  information should be known beforehand (such as the OS).

Arguments:
  HOST           Name of the host to add, will name the folder after it.
                 Example: db.myapp.com  [required]

  [DESCRIPTION]  The description of the host, to display to user. Example:
                 Database server for running PostgreSQL

  OS             Vague description of the operating system that the host is
                 running. Logic can be written for hosts based on it (for
                 example, using yum on CentOS, but apt on Debian). Example:
                 CentOS 8  [required]

  [USERNAME]     The username to connect to on the remote machine through SSH.
                 The user needs to be able to execute commands with sudo with
                 no password if not root.  [default: root]

  [KEYFILE]      The name of the SSH key file in the host folder. Example:
                 id_rsa  [default: id_rsa]


Options:
  --help  Show this message and exit.
```

1.16. att. Palīdzības izvades paraugs jauna servera pievienošanai

Kā iespējams ievērot, pašas atslēgu datnes netika pievienotas ar šo komandu. Šajā gadījumā, tos nolasa no standartievades, piemēram, norādot uz šīs datnes atrašanās vietu, tās saturu izvadot ar "cat" un uzreiz to padodot tālāk uz atslēgas importa komandu:

```
cat ~/.ssh/mysite/app/id_rsa | docker run --rm -i \
-v astolfo_tasks:/app/tasks \
-v astolfo_infrastructure:/app/infrastructure \
-v astolfo_files:/app/files \
astolfo host key 1.worker.catboi.net
```

Arī šīs darbības rezultātā, pašas datnes saturs netiek izvadīts, lai mazinātu tā noplūdes iespēju, piemēram, izpildei notiekot uz CI/CD servera, kur čaulas izvade tiek piefiksēta.

```
Adding key for host: 1.worker.catboi.net
Key will be added in file: infrastructure/1.worker.catboi.net/id_rsa
Wrote 3243 characters to file...

Successfully added host key!
```

1.17. att. Atslēgas pievienošanas izvade

Kopumā, šāda pieeja atļauj realizēt līdzīgu serveru pārvaldes funkcionalitāti Ansible piedāvātajai, ļaujot šos pārvaldes uzdevumus veikt automatizētā veidā, arī iekš Docker konteineru konteksta, piemēram, šo vērtību saturu padodot no GitLab CI/CD vides mainīgo vidus, tos nekad neizvadot pa tiešo, lietotājam redzamā formātā.

Ir realizēta funkcionalitāte arī lai pārvaldītu datnes, ne tikai SSH atslēgas. Tas ļauj apskatīt uzdevumu lejupielādēto datņu saturu, vai arī pievienot datnes, kuras ar SFTP caur SSH pārsūtīt uz attālināto serveri, piemēram konteineru vides deklarāciju YAML formātā. Līdzīgi atslēgām, arī šeit pastāv iespēja no standartievades izgūt to saturu. Papildus tam tiek piedāvātas arī komandas šo datņu pārsaukšanai, dzēšanai un to satura nolasīšanai:

```
docker run --rm -i \
-v astolfo_tasks:/app/tasks \
-v astolfo_infrastructure:/app/infrastructure \
-v astolfo_files:/app/files \
astolfo file list
```

Šī komanda arī izvada datņu saturu JSON formātā, šajā gadījumā arī izmantojot atsoļa sekvences, jauno rindu apstrādei, kas gan var padarīt izvadi mazāk lasāmu komandrindas apskates kontekstā, taču ir labi piemērota, ja šos datus vajadzētu attēlot tīkla interfeisā.

```
kronislv@catbook:/media/kronislv/6EEA-1A2A/Nextcloud/RTU/maģistra darbs$ docker run astolfo file list
[
  {
    "py/object": "cli.file_data.File",
    "filename": "covid1984load-kubernetes.yml",
    "contents": "apiVersion: v1\nitems:\n- apiVersion: v1\n  kind: Pod\n  metadata:\n    creationTimestamp: null\n    labels:\n
      io.kompose.service: covid1984-load-test\n    name: covid1984-load-test\n  spec:\n    containers:\n    - env:\n      - nam
e: TEST_DURATION_SECONDS\n        value: \"1800\"\n      - name: TEST_SECONDS_BETWEEN_REQUESTS\n        value: \"1\"\n      - n
ame: TEST_URL\n        value: http://worker.catboi.net\n      - name: TEST_USERS\n        value: \"1000\"\n      image: registr
y.kronis.dev/rtu1/kvps5_masters_degree_covid_1984_load_test/covid1984_load_test\n      name: covid1984-load-test\n      resourc
es:\n        requests:\n          cpu: 100m\n          memory: \"512000000\"\n        limits:\n          cpu: 1800m\n
memory: \"7340000000\"\n    nodeSelector:\n      kubernetes.io/hostname: tester.catboi.net\n    restartPolicy: Never\n    image
PullSecrets:\n    - name: registrycredentials\n  status: {}\nkind: List\nmetadata: {}\n\n"
  },
  {
    "py/object": "cli.file_data.File",
    "filename": "covid1984-kubernetes.yml",
    "contents": "apiVersion: v1\nitems:\n- apiVersion: v1\n  kind: Service\n  metadata:\n    annotations:\n      kompose.cmd: /
snap/kompose/19/kompose-linux-amd64 convert -f covid1984.yml -o\n        covid1984-kubernetes.yml\n      kompose.version: 1.21.
0 (992df58d8)\n    creationTimestamp: null\n    labels:\n      io.kompose.service: covid1984-app\n    name: covid1984-app\n  sp
ec:\n    type: NodePort\n    ports:\n    - port: 80\n      targetPort: 3000\n      protocol: TCP\n      name: http\n      nodeP
ort: 80\n    selector:\n      io.kompose.service: covid1984-app\n  status:\n    loadBalancer: {}\n- apiVersion: v1\n  kind: Ser
vice\n  metadata:\n    annotations:\n      kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f covid1984.yml -o\n
   covid1984-kubernetes.yml\n      kompose.version: 1.21.0 (992df58d8)\n    creationTimestamp: null\n    labels:\n      io.komp
ose.service: covid1984-postgis\n    name: covid1984-postgis\n  spec:\n    ports:\n    - name: \"5432\"\n      port: 5432\n
 targetPort: 5432\n    selector:\n      io.kompose.service: covid1984-postgis\n  status:\n    loadBalancer: {}\n- apiVersion: a
pps/v1\n  kind: Deployment\n  metadata:\n    annotations:\n      kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f c
```

1.18. att. Datņu satura izvades paraugs

Izvēlētā pieeja gan padara šo funkcionalitāti derīgu galvenokārt teksta datņu pārvaldei, taču lejupielādēt un augšupielādēt arī binārā formāta datnes ir iespējams bez problēmām arī bez šīs apskates funkcionalitātes, jo tā loģika izmanto SFTP, nevis teksta ievadi čaulā.

1.5.3 Daudzviet izmantojamā funkcionalitāte

Izstrādes ietvaros tika realizēta arī funkcionalitāte ar mērķi atvieglot jaunu uzdevumu izveidi, kas nodrošina metodes bieži izpildāmām darbībām, piemēram, datņu pārsūtīšanai un saņemšanai, vai arī komandu izpildei čaulā, kad ir nodibināts SSH savienojums. Šo loģiku var iedalīt 2 grupās:

- lokāli nepieciešamā loģika paša rīka darbībai (un kas ir atrodama "src/cli" direktorijā)
- loģika, kas paredzēta attālināto serveru administrēšanai (un kas ir atrodama "src/tasks/utils" direktorijā)

Ja nepieciešams, pilno funkcionalitātes kodu ir iespējams apskatīt lietotnes pirmkodā, taču pati funkcionalitāte ir sadalīta datnēs atbilstoši to pielietojuma mērķim. Piemēram, lokāli nepieciešamās loģikas gadījumā, ir sekojošie Python koda datnes ar funkcionalitāti:

- file_actions.py - loģika, kas ir nepieciešama darbībām ar datnēm (to satura apskatīšana, jaunu datņu pievienošana vai esošo dzēšana)
- file_data.py - datnes klases definīcija, kas ietver datnes reprezentāciju un JSON pārveidošanai nepieciešamo kodu
- host_actions.py - loģika, kas ir nepieciešama pārvaldāmo serveru lokālo datu uzturēšanai, piemēram, to izveidei un dzēšanai un SSH atslēgu pievienošanai
- host_data.py - servera klases definīcija, kas ietver datu reprezentāciju un JSON pārveidošanai nepieciešamo kodu
- task_actions.py - loģika, kas realizē jaunu uzdevumu izveidi, kā arī esošo dzēšanu; tā mēģina inicializēt uzdevuma kodu to pievienojot, gadījumā ja tas neizdodas, tad uzreiz to izdzēšot
- task_data.py - paša uzdevuma klases definīcija, kas katram no tiem ļauj uzturēt savu nosaukumu un aprakstu

Kā redzams, ir nodalītas Python datnes, kuri ietver darbības un loģiku, kā arī tie, kuri ietver tikai klašu definīcijas, kas darīts ar mērķi ļaut pašiem datņu nosaukumiem pateikt, ko tie ietver - piemēram, SSH atslēgas pašas par sevi ir tikai datnes, kā dēļ tām nav paredzēta sava klases izveide.

Attiecībā uz datnēm, kuros ir datu klases, to struktūra ir līdzīga jau iepriekšminētajai uzdevumu JSON izvadei. Piemēram, host_data.py datnē ir aprakstīta Host klase, kura sevī ietver informāciju, kura ir nepieciešama, lai izpildītu komandas pret serveriem - tā tiks izmantota, lai tiem pieslēgtos un atrastu īsto SSH atslēgu, kuru izmantot:

```
from typing import List
import jsonpickle
jsonpickle.set_preferred_backend('demjson')

class Host(object):
    def __init__(
        self,
        host,
        description,
        os,
        username,
        keyfile,
    ):
        self.host = host
        self.description = description
        self.os = os
        self.username = username
        self.keyfile = keyfile
    @classmethod
    def read(self, input: str):
        return jsonpickle.decode(input)
    @classmethod
    def dump_all(self, input: List):
        return jsonpickle.encode(input, indent=2)
    def dump(self):
        return jsonpickle.encode(self, indent=2)
```

Šajā gadījumā var ievērot, ka tiek izmantota "List" klase no Python "typing" moduļa, kurš ļauj metodēm pievienot norādes uz tipiem (angliski "type hints") [14], tā atļaujot labāku sadarbību ar integrētajām izstrādes vidēm (angliski "integrated development environment" jeb IDE), tostarp padoto parametru tipu pārbaudes, pat ja pagaidām Python valoda pati šādu funkcionalitāti izpildes laikā neatļauj realizēt.

Tas arī nozīmē, ka skriptu izpildei ir nepieciešama vismaz Python 3.5+ versija, jo šī funkcionalitāte nebija pieejama vecākajās valodas versijās, piemēram Python 2.7, taču pateicoties Docker rīks tāpat var strādāt uz sistēmām, kurās ir vecā Python versija, vai Python interpretators nav pieejams vispār. Papildus tam, šeit ir arī "dump_all" metode, kura atļauj objektu saraksta serializāciju uz JSON formātu, gadījumā ja nepieciešams izvadīt visus pieejamos serverus.

Papildus datnēm, kurās ir klases dažādo darbībā izmantoto datu reprezentēšanai, ir arī datnes, kurās ir pati funkcionalitātes realizācija. Tajās ir gan metodes, kuras paredzēts izsaukt kā daļu no komandrindas loģikas un kas atgriež tekstuālo JSON objektu reprezentāciju, kā arī metodes, kas darbojas ar datu tipiem pa tiešo un atgriež objektus. Piemēram, jauna servera pievienošanas gadījumā tiks izsaukta sekojošā metode, kurai tiks padoti parametri kā simbolu virknes:

```
def action_host_add(
    host: str,
    description: str,
    os: str,
    username: str,
    keyfile: str
):
    new_host = Host(
        host, description, os, username, keyfile
    )
    save_host(new_host)
    return new_host.dump()
```

Savukārt pati metode, kas realizē attiecīgā servera saglabāšanu saņems "Host" objekta instanci un to pašu arī atgriezīs pēc veiksmīgas saglabāšanas. Šeit gan var ievērot, ka šīs komandas arī var izvadīt datus uz konsoli, taču šajā gadījumā šī izvade nonāk STDERR kanālā, tātad, netraucēs pārējo lietotnes daļu darbībai:

```
def save_host(host: Host):
    typer.echo("Creating new host: " + str(host.host), err=True)
    infrastructure_folder = os.path.join("infrastructure", host.host)
    host_file = os.path.join(infrastructure_folder, "server.json")
    if not os.path.exists(infrastructure_folder):
        typer.echo("Folder: " + str(infrastructure_folder) + " does not
exist, creating new...", err=True)
        os.makedirs(infrastructure_folder)
    typer.echo("Writing to file: " + str(host_file), err=True)
    output_file = open(host_file, "w")
    output_text = host.dump()
    written_characters = output_file.write(output_text)
    output_file.close()
    typer.echo("Wrote " + str(written_characters) + " characters to
file...", err=True)
```

Tas atļauj nodalīt pašu loģikas implementāciju no tās izsaukšanas komandrindas kontekstā. Lai gan pašas komandrindas darbības arī ir Python metodes, kuras vajadzības gadījumā var izsaukt pa taisno, gadījumā, ja tiktu izveidots tīkla interfeiss vai rīks tiktu integrēts ar citām tehnoloģijām, "action_host_add" varētu aizstāt ar citu implementāciju.

1.5.4 Dokumentācija

Svarīga ir arī ne tikai izvade no paša rīka tā lietošanas laikā, bet arī dokumentācija tam, neatkarīgi no tā, vai tā tiek pasniegta palīdzības sistēmā lejupielādējamā lietotnē, vai tā tiek pasniegta kā daļa lietotnes mājaslapas, vai arī ja tā ir pieejama Git repozitorijā kopā ar rīka kodu. Lai novērstu situāciju, kurā būtu jāuztur vairākas nodalītas informācijas vienības par vienu un to pašu rīku, šajā gadījumā dokumentācija tika realizēta turpat, Git repozitorijā, kā Markdown formāta teksta datnes - tas atļauj tos gan viegli pārmeklēt, gan tajos vajadzības gadījumā iekļaut tabulas un attēlus, kā arī uzturēt atsauces uz citām datnēm.

Šī dokumentācija ļauj jebkuram interesantam, atverot attiecīgo repozitoriju, iepazīties ar to, ko šis rīks dara, kam tas ir domāts, kā arī norāda uz datni, kurā aprakstīts tas, kā to sākt lietot (1.19. att.).

> A solution to automate the creation and testing of a variety of containerization solutions as well as run other tasks.

One of the problems that I have oftentimes stumbled upon is the initial setup of the infrastructure for running containerized workloads. Relying on managed control planes and SaaS/PaaS solutions can be both expensive and risky, since such services could be retired at any time, could create security vulnerabilities (as opposed to everything being 100% on-prem and not exposed to the outside world) and also have breaking changes with no control over when and how migrations are handled.

However, setting up Kubernetes and Docker Swarm, or even Hashicorp Nomad clusters can take time and even when following the official instructions, it can be a bit hard to walk through this process. That's why I wrote Astolfo, which is my attempt at providing the convenience of a CentOS graphical installer, but for setting up infrastructure. Of course, if you don't need the UI bits, just use the CLI directly.

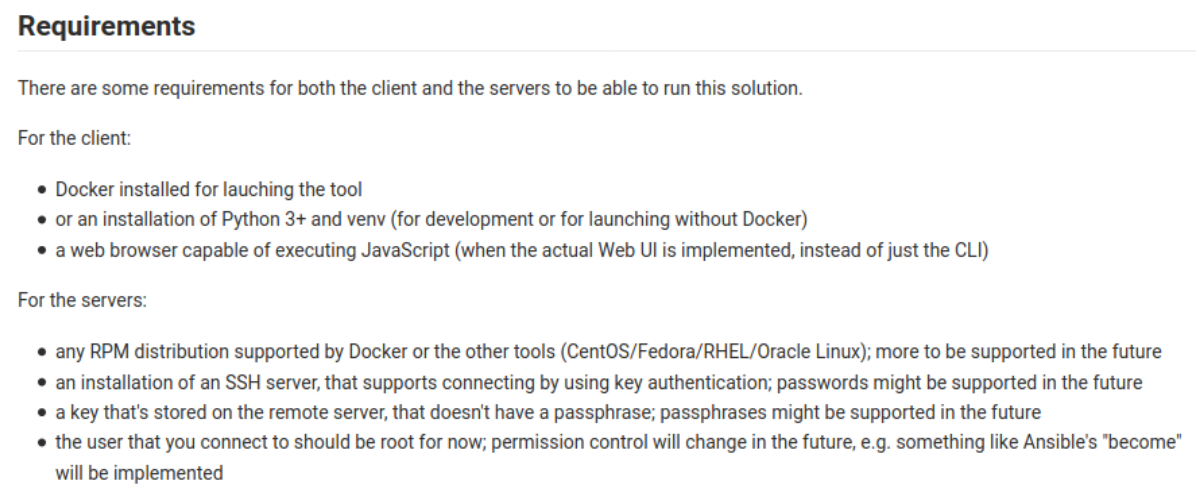
Internally, it uses Paramiko to execute commands over SSH. This approach was chosen as opposed to just using Ansible or Salt, because oftentimes Ansible won't work well with imperative commands. This small set of scripts is also easily *hackable*, in case you want to implement new functionality.

If you'd like to see an example of how to use the application, feel free to read the Quick Start guide in GUIDE.md.

1.19. att. README.md ievada paraugs

Dokumentācijas glabāšana teksta formātā arī atvieglo tās izstrādi un papildināšanu, jo nepieprasa papildus programmatūru tās atvēršanai, kā arī atļauj izmantot pierastos rīkus tās pārmeklēšanai, piemēram, ar "grep" rīku meklējot visā dokumentācijā noteiktu saturu. Līdzīgi, ja tiek izmainīts pats kods vai arī mainās lietotnes nodrošinātā funkcionalitāte, tad arī šo dokumentāciju atjaunot kļūs vieglāk un mazināsies iespēja, ka atsevišķā "Wiki" saitā, vai kādā citā ārējā sistēmā uzglabātā dokumentācija tiks izlaista un netiks atjaunota.

Papildus tam, šis pats "README.md" datne sniedz ieskatu tajā, kas ir nepieciešams, lai palaistu attiecīgo rīku gan lokāli, gan uz attālinātajiem serveriem, piemēram, par to, ka pašlaik tiek atbalstītas tikai RPM Linux distribūcijas (1.20. att.).



Requirements

There are some requirements for both the client and the servers to be able to run this solution.

For the client:

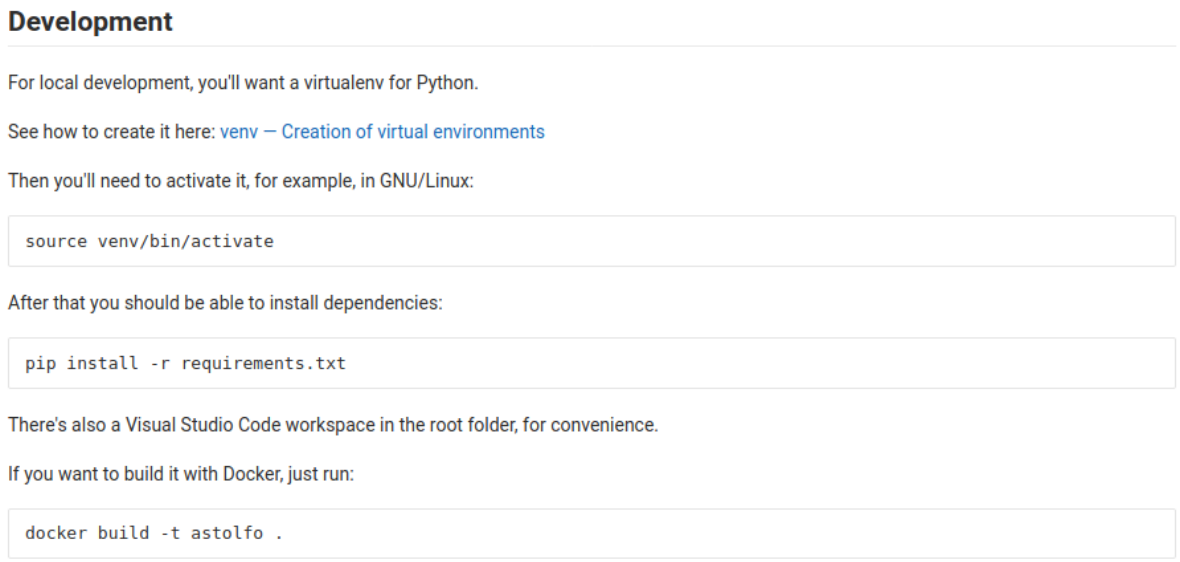
- Docker installed for lauching the tool
- or an installation of Python 3+ and venv (for development or for launching without Docker)
- a web browser capable of executing JavaScript (when the actual Web UI is implemented, instead of just the CLI)

For the servers:

- any RPM distribution supported by Docker or the other tools (CentOS/Fedora/RHEL/Oracle Linux); more to be supported in the future
- an installation of an SSH server, that supports connecting by using key authentication; passwords might be supported in the future
- a key that's stored on the remote server, that doesn't have a passphrase; passphrases might be supported in the future
- the user that you connect to should be root for now; permission control will change in the future, e.g. something like Ansible's "become" will be implemented

1.20. att. README.md informācija par prasībām izpildvidei

Papildus tam, šeit ir arī informācija, kas nepieciešama, lai varētu papildināt rīka funkcionalitāti ar jauniem uzdevumiem vai to palaistu pa tiešo, bez konteineriem (1.21. att.).



Development

For local development, you'll want a virtualenv for Python.

See how to create it here: venv — Creation of virtual environments

Then you'll need to activate it, for example, in GNU/Linux:

```
source venv/bin/activate
```

After that you should be able to install dependencies:

```
pip install -r requirements.txt
```

There's also a Visual Studio Code workspace in the root folder, for convenience.

If you want to build it with Docker, just run:

```
docker build -t astolfo .
```

1.21. att. README.md informācija par rīka izstrādi

Šīs dokumentācijas pilno saturu iespējams apskatīties GitLab instancē vai arī 1., 2. un 3. pielikumos, kuri arī ietver lietošanas gidu un lietošanas paraugus visām komandām.

1.6 Izstrādātā rīka funkcionalitāte

Maģistra darba ietvaros tika realizēta rīka funkcionalitāte ar mērķi parādīt gan parastu servera administrācijas darbību veikšanu, tostarp paku instalāciju un servisu konfigurēšanu, gan arī ar konteinerizāciju saistītas darbības - Docker sagatavošanu lietošanai, kā ar Docker Swarm un Kubernetes K3s distribūcijas klasteru izveidi, serveru pievienošanu tiem, kā arī grafiskā Portainer rīka palaišanu.

Šai funkcionalitātei vajadzētu gan kalpot par praktisku piemēru rīka lietošanai un noderībai, gan arī atļaut uz esošo uzdevumu bāzes izstrādāt jaunus. Pilnais uzdevumu pirmkods ir pieejams Git repozitorijā, savukārt šeit tiks sniegts vispārīgs ieskats daļā no piedāvātās funkcionalitātes. Pašlaik rīkā ir implementēti 27 dažādi uzdevumi.

1.6.1 Informācijas izguve par serveri

Dotajā brīdī rīkā ir realizētas dažas informatīva rakstura komandas, kuras nemaina sistēmas stāvokli, bet iegūst informāciju par to. Tās ir: "task_ping", "task_os_version", "task_disk_usage" un "task_installed_packages", kuras pārbauda savienojumu ar serveri, iegūst informāciju par tā operētājsistēmu, kā arī pārbauda brīvās vietas daudzumu uz tā diskiem un gan saskaita, gan atgriež sarakstu ar instalētajām pakām uz servera.

Pirmā komanda vienkārši pārbauda, vai ir iespējams izveidot SSH savienojumu ar attālināto serveri un vai lietotājs var izpildīt "sudo" komandu bez paroles priviliģēto darbību izpildei, kas var būt nepieciešamas servisu pārvaldei. Nākotnē var būt nepieciešams implementēt ko līdzīgu Ansible nodrošinātajai "become" komandai [15], kas nosaka, vai ir nepieciešams izsaukt "sudo su" ekvivalentu, administratīvo darbību veiksmīgai izpildei. Tās izpildes rezultātā varam iegūt objektu, kurš reprezentēs darbību rezultātu (1.22. att.).

```
kronislv@catbook:~$ docker run astolfo task run ping 1.worker.catboi.net 2>/dev/null
{
  "py/object": "tasks.task_ping.PingResponse",
  "host_pinged": "1.worker.catboi.net",
  "host_user": "root",
  "connection_successful": true,
  "user_can_sudo": false
}
```

1.22. att. Paraugs "ping" komandas izpildei

Otrā komanda arī izpilda vienkāršas teksta izvades darbības lai noskaidrotu kāda operētājsistēma ir uz attālinātā servera, kā arī lai pārbaudītu tās kodola versiju. Tajā tiek izmantota "cat" un "uname" komandu kombinācija, kas iegūst tās rezultātus (1.23. att.).

```
kronislv@catbook:~$ docker run astolfo task run os_version 1.worker.catboi.net 2>/dev/null
{
  "py/object": "tasks.task_os_version.OSVersionResponse",
  "version_info": [
    "CentOS Linux release 8.3.2011",
    "NAME=\"CentOS Linux\"",
    "VERSION=\"8\"",
    "ID=\"centos\"",
    "ID_LIKE=\"rhel fedora\"",
    "VERSION_ID=\"8\"",
    "PLATFORM_ID=\"platform:el8\"",
    "PRETTY_NAME=\"CentOS Linux 8\"",
    "ANSI_COLOR=\"0;31\"",
    "CPE_NAME=\"cpe:/o:centos:centos:8\"",
    "HOME_URL=\"https://centos.org/\"",
    "BUG_REPORT_URL=\"https://bugs.centos.org/\"",
    "CENTOS_MANTISBT_PROJECT=\"CentOS-8\"",
    "CENTOS_MANTISBT_PROJECT_VERSION=\"8\"",
    "CentOS Linux release 8.3.2011",
    "CentOS Linux release 8.3.2011"
  ],
  "kernel_info": [
    "Linux 1.worker.catboi.net 4.18.0-80.7.1.el8_0.x86_64 #1 SMP Sat Aug 3 15:14:00 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux"
  ]
}
```

1.23. att. Paraugs "os_version" komandas izpildei

Trešā komanda izsauc "df" rīku, kurš ļauj pārbaudīt, cik uz servera ir brīvās vietas pieejamās disku daļās (angliski "partition"), šajā gadījumā izvadi formatējam cilvēkiem salasāmā formātā, ne tikai izvadot lielumus baitos (1.24. att.).

```
kronislv@catbook:~$ docker run astolfo task run disk_usage 1.worker.catboi.net 2>/dev/null
{
  "py/object": "tasks.task_disk_usage.DiskUsageResponse",
  "disk_usage": [
    "Filesystem      Size  Used Avail Use% Mounted on",
    "/dev/sda1        40G  2.9G   35G   8% /"
  ]
}
```

1.24. att. Paraugs "disk_usage" komandas izpildei

Pēdējā komanda izmanto "yum" paku pārvaldnieku, lai noskaidrotu, cik daudzas pakas ir instalētas sistēmā un atgriež arī pilno paku sarakstu. Tāpēc ka šī izvade ir visai gara, tā šeit netiek piedāvāta, tomēr tā var būt noderīga, lai salīdzinātu sistēmas stāvokli pirms un pēc noteiktas programmatūras instalācijas.

Piemēram, ja mūs interesē iespēja uzzināt, cik daudzas pakas ir nepieciešams uzinstalēt Docker Swarm un Kubernetes K3s distribūcijas orķestratoru palaišanai, tad atliek vien izpildīt šīs komandas pirms un pēc attiecīgo rīku instalācijas. Šīs komandas var tālāk

kombinēt arī ar diska patēriņa pārbaudes komandām, kas ļauj precīzāk pateikt, ne tikai, cik pakas ir nepieciešams instalēt, bet arī, cik daudz vietas uz diska tās aizņem (1.1. tabula).

1.1. tabula

Paraugs instalēto paku skaitam un diska vietas patēriņam orķestratoriem

Scenārijs	Instalēto paku skaits	Diska vietas patēriņš
Sistēma bez orķestratoriem	430	1.9 GB
Docker Swarm orķestrators	447	2.4 GB
Kubernetes K3s orķestrators	448	2.7 GB

Tas ļauj secināt, ka Kubernetes pat distribūcijās, kas ir orientētas uz mazāku resursu patēriņu var aizņemt vairāk vietas uz diska, taču tajā pat laikā šīs atšķirības ir visai minimālas. Šeit Docker Swarm arī ir zināmas priekšrocības, jo Swarm orķestrators jau ir iekļauts, uz servera instalējot Docker.

1.6.2 Paku instalācija un konfigurācija

Rīks atļauj arī specifiskas programmatūras instalāciju un konfigurāciju. Nākotnē iespējams šo loģiku papildināt ar noteiktu paku saraksta padošanu un automātisku visu to uzinstalēšanu, līdzīgi Ansible nodrošinātajam "yum" modulim, taču šeit uzsvars tika likts uz pakām, kuras ir saistītas ar servisiem, kuriem var būt nepieciešama papildus konfigurācija. Šeit tiek piedāvāti divi paku instalācijas un konfigurācijas paraugi: "task_fail2ban" un "task_zabbix_agent", kas piedāvā uzinstalēt fail2ban servisu un Zabbix klienta programmatūru.

Fail2ban ir programmatūra, kura ļauj automātiski piefiksēt neveiksmīgos pieslēgšanās mēģinājumus sistēmai caur SSH un izveidot ugunsmūra likumus, kuri attiecīgo pieprasījumu IP adreses uz laiku bloķēs [16], ja secīgi noteiktā laikā intervālā tiks izdarīti pārāk daudzi secīgi mēģinājumi. Šādu rīku ir lietderīgi izmantot, ja serverim nepieciešams atļaut ne tikai piekļuvi ar SSH atslēgām, bet arī ar parolēm, kuras ļaunprātīgi darboņi bieži vien mēģina uzlauzt publiski pieejamiem serveriem (1.25. att.). Tas, protams, neizslēdz iespēju, ka paroles tāpat var uzlauzt, ja tās nav izvēlētas pietiekoši garas, tomēr var ļaut mazināt ar to saistītos riskus.

```
Jan  2 19:12:32 ram sshd[27904]: Failed password for root from 49.88.112.110 port 52697 ssh2
Jan  2 19:12:37 ram sshd[27904]: Failed password for root from 49.88.112.110 port 52697 ssh2
Jan  2 19:12:38 ram sshd[27906]: Failed password for root from 67.205.153.4 port 46826 ssh2
Jan  2 19:13:55 ram sshd[27947]: Failed password for root from 51.161.104.169 port 34821 ssh2
Jan  2 19:14:14 ram sshd[27970]: Failed password for root from 151.69.170.146 port 42301 ssh2
Jan  2 19:14:47 ram sshd[28001]: Failed password for root from 61.177.172.61 port 51654 ssh2
Jan  2 19:14:51 ram sshd[28005]: Failed password for root from 217.160.3.229 port 36082 ssh2
Jan  2 19:14:52 ram sshd[28001]: Failed password for root from 61.177.172.61 port 51654 ssh2
Jan  2 19:14:52 ram sshd[28003]: Failed password for root from 81.69.47.35 port 20810 ssh2
Jan  2 19:14:55 ram sshd[28001]: Failed password for root from 61.177.172.61 port 51654 ssh2
Jan  2 19:14:59 ram sshd[28001]: Failed password for root from 61.177.172.61 port 51654 ssh2
Jan  2 19:15:02 ram sshd[28009]: Failed password for root from 51.178.87.42 port 53452 ssh2
Jan  2 19:15:06 ram sshd[28029]: Failed password for root from 49.88.112.110 port 60466 ssh2
Jan  2 19:15:16 ram sshd[28037]: Failed password for root from 80.11.253.8 port 58440 ssh2
Jan  2 19:15:29 ram sshd[28055]: Failed password for root from 67.205.153.4 port 56916 ssh2
kronislv@ram:~$ sudo grep "Failed password" /var/log/auth.log | wc -l
34528
```

1.25. att. Paraugs mēģinājumiem nesankcionēti piekļūt serverim

Attiecīgā uzdevuma ietvaros tiek lejupielādēta šī programmatūra un tiek palaists šis serviss, pēc kā arī tiek pārbaudīts tā statuss un apskatītas uzģenerētās audita datnes, ja tādas ir. Kā iepriekš, pašas komandas izvade STDOUT kanālā ir visai minimāla (1.26. att.), jo tā ļauj tikai pārliecināties par izpildes rezultātiem.

```
kronislv@catbook:~$ docker run astolfo task run fail2ban 1.worker.catboi.net 2>/dev/null
{
  "py/object": "tasks.task_fail2ban.Fail2BanResponse",
  "package_installed": true,
  "systemctl_enable": true,
  "service_restart": true
}
```

1.26. att. Paraugs "fail2ban" uzdevuma izpildei

Pilnā izpilde ietver arī izvadi no audita datnēm, jo iekšēji uzdevuma rezultātos ir vairākas stadijas, kuru ietvaros tiek izpildītas vairākas komandas (1.27. att.).

```
Executing: rpm -qa | grep -qw fail2ban || yum -d1 install -y fail2ban
Executing: systemctl enable fail2ban
Executing: service fail2ban restart && systemctl --no-pager status fail2ban
Executing: tail -n 32 /var/log/fail2ban.log; echo 'No logs exist yet!'
```

1.27. att. Paraugs fail2ban instalācijas procesā izpildītajām komandām

Kā redzams, šīs darbības varētu izpildīt arī manuāli, pa tiešo uz servera, taču rīka izmantošana ļauj ar vienas komandas palīdzību gan veikt šī rīka instalāciju, gan arī pārbaudīt, vai tā ir bijusi veiksmīga uz vairākiem serveriem secīgi, kā daļu no kādas skripta, kā arī pieglabāt izvadi no tā datnē, vai to pārvadīt uz citām komandām, ja nepieciešams.

Papildus šādiem rīkiem varētu interesēt arī iespēja automatizēt servera monitoringa konfigurāciju - atļauj noteiktiem rīkiem, piemēram, Nagios vai Zabbix pieslēgties serverim un no tā ievākt informāciju par sistēmas stāvokli, resursu patēriņu un citiem lielumiem. Darba ietvaros tika realizēts paraugs Zabbix aģenta programmatūras konfigurācijai, jo tas tika izmantots arī izstrādātā konteinerizētā rīka monitoringam, lai salīdzinātu orķestratoru darbību.

Zabbix gadījumā situācija ir sarežģītāka kā ar fail2ban, jo ir nepieciešams mainīt arī noklusējuma konfigurāciju, lai norādītu, kur atrodas Zabbix serveris, kam atļaut datu ievākšanu. Paša skripta ietvaros, servera adrese tiek saņemta kā daļa JSON argumenta, ko pēc tam konvertē Python objektā, pēc kā attiecīgo adresi iekļauj kā daļu no izpildāmās čaulas komandas:

```
command_input = ZabbixAgentRequest.read(input)
...
server_address = command_input.server_address
...
command = "sed -i '/Server=127.0.0.1/c\Server=" + str(server_address) + "'
/etc/zabbix/zabbix_agentd.conf"
error_code, output = ssh_execute(ssh_connection, command)
if not error_code:
    complete_stage(stages, "ServerUpdate", successful=True,
command=command, output=output)
    command_output.server_update = True
else:
    complete_stage(stages, "ServerUpdate", successful=False,
command=command, output=output)
    raise Exception("Failed to set the Server parameter in config!")
```

Šajā gadījumā, šī loģika kopā ar pārējiem soļiem skripta ietvaros rezultē salīdzinoši garākā izvadē (1.28. att.), šim procesam automatizētā formātā esot daudz ērtāk izpildāmam, nekā to cenšoties darīt katram serverim manuāli, labojot konfigurācijas teksta datnes ar teksta redaktoriem, piemēram "vi" vai "nano".

```
Executing: rpm -qa | grep -qw zabbix-release-3.2-1.el7.noarch || rpm -ivh http://repo.zabbix.com/zabbix/3.2/rhel/7/x86_64/zabbix-release-3.2-1.el7.noarch.rpm
Executing: rpm -qa | grep -qw zabbix-agent || yum -d1 install -y zabbix-agent
Executing: if [ ! -f /etc/zabbix/zabbix_agentd.conf.bak ]; then cp /etc/zabbix/zabbix_agentd.conf{,.bak}; else echo 'Backup file already exists!'; fi
Executing: sed -i '/Server=127.0.0.1/c\Server=ram.servers.kronis.eu' /etc/zabbix/zabbix_agentd.conf
Executing: systemctl enable zabbix-agent
Executing: service zabbix-agent restart && systemctl --no-pager status zabbix-agent
Executing: tail -n 32 /var/log/zabbix/zabbix_agentd.log; echo 'No logs exist yet!'
```

1.28. att. Paraugs komandām, kuras tiek izpildītas instalējot Zabbix aģentu

Zabbix gadījumā šos serverus nepieciešams piereģistrēt arī uz centrālā servera, taču tas tika darīts manuāli, caur tīkla interfeisu, kā dēļ šeit netiek sīkāk apskatīts.

1.6.3 Docker Swarm administrēšanas darbības

Papildus sistēmas programmatūras administrēšanai tika realizēti arī uzdevumi tieši konteineru tehnoloģiju uzinstalēšanai un sagatavošanai uz servera, lai varētu palaist konteinerizētas lietotnes. No sākuma tika realizēta funkcionalitāte uzdevumam "task_docker", kas ļauj uzinstalēt un sagatavot Docker rīku, ņemot vērā, ka pēc noklusējuma uz CentOS 8 un citām RPM distribūciju grupas GNU/Linux distribūcijām tas nav spējīgs darboties ar tekošo publiski pieejamo stabilo versiju (19.03).

Šī uzdevuma ietvaros tiek veiktas vairākas darbības, lai uz sistēmas uzstādītu Docker uzņēmuma "yum" paku repozitoriju, no tā lejupielādētu un uzinstalētu nepieciešamās pakas. Pēc tam tiek nodrošināts, ka pats serviss ir iespējots, kā arī tiek salabota minētā ugunsmūra kļūda, kura var traucēt iekšējā DNS izmantošanu, šajā gadījumā izmainot ugunsmūra konfigurāciju tās apiešanai (1.29. att.) - ugunsmūra atslēgšana kā problēmas risinājums nebūtu pieņemama, jo radītu ievērojamus drošības riskus.

```
Executing: rpm -qa | grep -qw yum-utils || yum -d1 install -y yum-utils && yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
Executing: rpm -qa | grep -qw docker-ce || yum -d1 install -y docker-ce docker-ce-cli containerd.io
Executing: systemctl enable docker
Executing: firewall-cmd --zone=public --add-masquerade --permanent && firewall-cmd --reload
Did not need to add a masquerade to the firewall, because it appears that the firewall is not enabled!
Executing: service docker restart && systemctl --no-pager status docker
Executing: docker run --rm hello-world
Executing: journalctl --no-pager -u docker | tail -n 32; echo 'No logs exist yet!'
```

1.29. att. Paraugs "docker" uzdevuma izpildītajām komandām

Pēc šo darbību veikšanas, tiek pārlādēta ugunsmūra konfigurācija, lai jaunie likumi stātos spēkā, kā arī tiek pārstartēts pats "docker" serviss un pārbaudīts, vai ir iespējams palaist vienkāršu konteineru, pēc kā, līdzīgi iepriekšējiem uzdevumiem, tiek izvadīta informācija no sistēmas audita datnēm.

Papildus tam ir realizētas arī vairākas komandas, kas atļauj izveidot Docker Swarm klasterus, kā arī pievienot serverus tiem: "task_docker_swarm_cluster", "task_docker_swarm_leave", "task_docker_swarm_master", "task_docker_swarm_worker". Šo uzdevumu izpilde arī ir salīdzinoši gara, taču šeit ievērības cienīgs ir tas, ka klastera izveides komandas izvadi var padot tālāk kā ievadi pievienošanās komandām, jo šajā izvadē būs vērtības, kas nepieciešamas, lai varētu pievienoties - marķieri (angliski "token"):

```
JOIN_TOKENS=$(python astolfo.py task run docker_swarm_cluster
1.worker.catboi.net)
python astolfo.py task run docker_swarm_worker 2.worker.catboi.net
"$JOIN_TOKENS"
python astolfo.py task run docker_swarm_worker db.catboi.net
"$JOIN_TOKENS"
python astolfo.py task run docker_swarm_worker tester.catboi.net
"$JOIN_TOKENS"
```

Minētajā izvadē būs ietverti gan nepieciešamie marķieri, lai atļautu pievienot serveri kā uzdevumu izpildītāju, gan arī kā klastera pārvaldītāju (1.30. att.).

```
kronislv@catbook:/media/kronislv/6EEA-1A2A/Nextcloud/RTU/maģistra darbs$ docker run astolfo task run docker_swarm_cluster 1.worker.catboi.net 2>/dev/null
{
  "py/object": "tasks.task_docker_swarm_cluster.DockerSwarmCreateResponse",
  "public_ip": "176.223.139.9",
  "swarm_created": false,
  "swarm_join_token_worker": "SWMTKN-1-3nukkbjfbrm9tnrg9932j1z3ht3jdarbt2in9x6gf1qau67nwl-8zjrvwa4tiubd764j06dfmrx0 176.223.139.9:2377",
  "swarm_join_token_master": "SWMTKN-1-3nukkbjfbrm9tnrg9932j1z3ht3jdarbt2in9x6gf1qau67nwl-cs2o2o8gcjusxptkue0f0pmm0 176.223.139.9:2377"
}
```

1.30. att. Paraugs klastera izveides izvadei

To, kuru no marķieriem izmantot, izlems komanda, kurā šī ievade tiks padota:

- task_docker_swarm_worker - pievienos serveri klasterim kā izpildītāju
- task_docker_swarm_master - pievienos serveri klasterim kā vadītāju

Papildus tam šeit tiek nodrošināta arī komanda, kas ļaus attiecīgajam serverim likt pamest klasteri: "task_docker_swarm_leave", ko var izmantot arī kā pirmo soli jauna klastera izveidē, ja serveris jau ir kādam pievienots un pie tā vairs nav vajadzīgs.

Tiek realizēti arī papildus uzdevumi, piemēram, "task_docker_swarm_label" ļaus serveriem pievienot marķējumus (angliski "labels"), ko pēcāk varēs izmantot konteineru izvietošanai uz specifiskiem serveriem. Piemēram, testējot maģistra darbā izveidoto sistēmu, lietotnes konteineri tiek izvietoti uz serveriem ar marķieriem "type=worker", kas tiek iegūti ar sekojošajām komandām:

```
# Add labels for deploying COVID1984 workload
python astolfo.py task run docker_swarm_label 1.worker.catboi.net
'{"py/object": "tasks.task_docker_swarm_label.DockerSwarmLabelRequest",
"target_host": "1.worker.catboi.net", "target_label": "type=worker"}'
python astolfo.py task run docker_swarm_label 1.worker.catboi.net
'{"py/object": "tasks.task_docker_swarm_label.DockerSwarmLabelRequest",
"target_host": "2.worker.catboi.net", "target_label": "type=worker"}'
```

Visbeidzot, tiek realizēts arī uzdevums "Portainer" programmatūras [17] palaišanai - to var izmantot gan Docker Swarm, gan tikai Docker, gan arī Kubernetes pārvaldīšanai, kā arī to palaist uz jebkuras vides, kas atbalsta konteineru palaišanu, tā neliekot to instalēt atsevišķi.

Tomēr šīs pieejas trūkums ir tas, ka Docker Swarm un Kubernetes vajag atšķirīgas vižu deklarācijas, kā arī palaišanas komandas attiecīgajiem orķestratoriem ir atšķirīgas. Tāpēc tika izveidoti uzdevumi abiem orķestratoriem, kas šīs programmatūras palaišanu atvieglo.

Portainer un Swarm kombinācijas gadījumā pati realizācija ir ļoti vienkārša, jo pašā Portainer mājaslapā tiek piedāvāts deklarācijas datne, kuru pietiek lejupielādēt un palaist iekš Docker Swarm (1.31. att.).

```
Executing: mkdir -p /docker/portainer
Executing: curl -L https://downloads.portainer.io/portainer-agent-stack.yml -o /docker/portainer/portainer-agent-stack.yml
Executing: docker stack deploy -c /docker/portainer/portainer-agent-stack.yml portainer
```

1.30. att. Portainer palaišanai nepieciešamās komandas uz Docker Swarm

Pēc tam, kad rīks būs palaists, tas kļūs pieejams servera 9000 portā. Parasti interesētu arī iespēja palaist to aiz cita tīkla servera, kurš varētu nodrošināt piekļuvi tam (angliski "reverse proxy"), kā arī realizētu automātisku SSL/TLS sertifikātu nodrošināšanu savienojuma drošības uzlabošanai, piemēram, Caddy tīkla serveri [18], vai jebkuru citu tīkla serveri kombinācijā ar Certbot rīku [19], kurš atļautu automātisko sertifikātu bezmaksas atjaunināšanu. Testa nolūkiem gan šāda konfigurācija pagaidām netika realizēta, to atstājot kā uzdevumu nākotnei, jo tikpat labi var nākties apsvērt arī Traefik servera [20] izmantošanas iespējas, jo K3s to nodrošina kā ingresa punktu, kas ļautu nākotnē arī salīdzināt abu orķestratoru tehnoloģiju ar šādu ingresam domātu serveri.

1.6.4 Kubernetes administrēšanas darbības

Kubernetes ir cits industrijā populārs konteineru orķestrators, taču daudzās no populārajām tā distribūcijām, piemēram Rancher Kubernetes Engine, var būt raksturīgi ar salīdzinoši lielu resursu patēriņu [21]. Tas var rezultēt sliktākā serveru veiktspējā, jo īpaši ja tiem ir ierobežoti resursi, piemēram, virtuālo privāto serveru izmantošanas gadījumā. Tāpēc tika izvēlēts uzmanību vērst tieši uz Rancher izstrādāto K3s distribūciju, kura ir tikusi sertificēta [22] Kubernetes atbalstam, tajā pašā laikā cenšoties vienkāršot risinājuma tehnisko sarežģītību un resursu patēriņu. Piemēram, tajā "etcd" distributētās datu glabātuves vietā tiek izmantota SQLite uz datnēm bāzētā datu bāze, kā arī šī distribūcija nodrošina vienkāršāku instalācijas procesu, tā atvieglojot darba uzsākšanu.

Līdzīgi Docker Swarm gadījumam arī šeit tiek realizētas komandas, kas ļauj sagatavot tehnoloģiju lietošanai un inicializēt klasterus: "kubernetes_k3s_cluster", "kubernetes_k3s_leave", "kubernetes_k3s_worker".

Arī šajā gadījumā tiek izmantots instalācijas skripts no K3s mājaslapas, kurš lejupielādēs jaunāko rīka versiju attiecīgajai operētājsistēmai, no kuras tas ir ticis izsaukts (1.31. att.).

```
Executing: curl -sfL https://get.k3s.io | sh -s - --docker && systemctl enable k3s --now
Executing: sed -i 's|server \\|server --kube-apiserver-arg service-node-port-range=1-65535 \\|g' /etc/systemd/system/k3s.service && systemctl daemon-reload && service k3s restart
Executing: cat /var/lib/rancher/k3s/server/node-token
Executing: cat /etc/rancher/k3s/k3s.yaml
```

1.31. att. Paraugs K3s klastera izveidošanai

Šeit gan problēmas radīs distribūcijas noklusējuma konfigurācija, kura "NodePort" režīmā padara pieejamus tikai dažus no portiem, kas nozīmēs ka Portainer nebūs iespējams palaist tādā pašā konfigurācijā, kā tas tika darīts Docker Swarm gadījumā, tāpēc testēšanas nolūkiem šī konfigurācija tiek mainīta, lai atļautu izmantot visus sistēmas portus, pieņemot, ka portu konflikti netiks radīti testēšanas ietvaros:

```
command = "sed -i 's|server \\\\|server --kube-apiserver-arg service-node-
port-range=1-65535 \\\\|g' /etc/systemd/system/k3s.service && systemctl
daemon-reload && service k3s restart"
error_code, output = ssh_execute(ssh_connection, command)
if not error_code:
    complete_stage(stages, "FixNodePortRange", successful=True,
command=command, output=output)
    command_output.node_ports_fixed = True
else:
    complete_stage(stages, "FixNodePortRange", successful=False,
command=command, output=output)
    raise Exception("Failed to fix the port range for NdoePort!")
```

Python pusē gan šai darbībai vajadzēja izmantot dubultās atsoļa sekvences, kas tālāk tika padotas "sed" komandai teksta aizvietošanai, K3s servera systemd servisa definīcijā. Citādi šis uzdevums darbojas līdzīgi Swarm paredzētajam un atgriež marķierus, kas ļauj šo izvadi padot pievienošanās komandām, piemēram:

```
JOIN_TOKENS=$(python astolfo.py task run docker_swarm_cluster
1.worker.catboi.net)
python astolfo.py task run docker_swarm_worker 2.worker.catboi.net
"$JOIN_TOKENS"
python astolfo.py task run docker_swarm_worker db.catboi.net
"$JOIN_TOKENS"
python astolfo.py task run docker_swarm_worker tester.catboi.net
"$JOIN_TOKENS"
```

Tāpat kā iepriekš šeit ir arī komandas marķējumu piešķiršanai noteiktiem serveriem, kas arī šeit vēlāk tiek izmantoti lietotņu palaišanas izplānošanai uz noteiktiem serveriem, kā arī šeit tiek nodrošināta Portainer rīka palaišana: "kubernetes_k3s_label", "kubernetes_k3s_portainer". Pirmās komandas realizācija ir izveidota tā, lai paši parametri būtu teju tādi paši kā Docker Swarm gadījumā:

```
# Add labels for deploying COVID1984 workload
python astolfo.py task run docker_swarm_label 1.worker.catboi.net
'{"py/object": "tasks.task_docker_swarm_label.DockerSwarmLabelRequest",
"target_host": "1.worker.catboi.net", "target_label": "type=worker"}'
python astolfo.py task run docker_swarm_label 1.worker.catboi.net
'{"py/object": "tasks.task_docker_swarm_label.DockerSwarmLabelRequest",
"target_host": "2.worker.catboi.net", "target_label": "type=worker"}'
```

Savukārt attiecībā uz Portainer palaišanu, šeit tiek izmantota modificēta vides deklarācija, kurā ir izmainīti noklusējuma porti, uz kuras rīks tiks padarīts pieejams, attiecīgā deklarācijas daļa izskatās sekojoši:

```
spec:
  type: NodePort
  ports:
    - port: 9000
      targetPort: 9000
      protocol: TCP
      name: http
      # We want to use the same port as on Docker Swarm
      # though this only works with changed service-node-port-range
      # nodePort: 30777
      nodePort: 9000
```

Tas tiek panākts, pielietojot realizēto funkcionalitāti, kura atļauj datnes zem "files" direktorijas pārsūtīt uz attālināto sistēmu, izmantojot SFTP, kas tiek Python kodā sasniegts sekojoši:

```
# Then we copy the stack file from the local device
try:
    local_file = load_file('portainer-kubernetes.yml')
    remote_file = "/docker/portainer"
    sftp_send(loaded_host, local_file, remote_file)
    complete_stage(stages, "UploadStack", successful=True, command="",
output="")
    command_output.upload_stack = True
except Exception as e:
    complete_stage(stages, "UploadStack", successful=False, command="",
output="")
    raise e
```

Pēc tam atliek izveidot Portainer domātu nosaukumvietu un uz attālinātā servera, izmantojot K3s uzinstalēto "kubectl" rīku, ir iespējams palaist deklarācijā ietverto lietotnes vidi uz nosaukumvietas, tā to nodalot no noklusējuma un atļaujot tai netraucēt, piemēram, paslēpjot to no palaisto lietotņu saraksta, līdzīgi kā tas tiek darīts ar sistēmas komponentēm:

```
# Deploy stack
command = "kubectl create namespace portainer && kubectl apply -n
portainer -f /docker/portainer/portainer-kubernetes.yml"
error_code, output = ssh_execute(ssh_connection, command)
if not error_code:
    complete_stage(stages, "DeployStack", successful=True,
command=command, output=output)
    command_output.deploy_stack = True
else:
    complete_stage(stages, "DeployStack", successful=False,
command=command, output=output)
    raise Exception("Failed to deploy the stack file!")
```

Taču atšķirībā no Docker Swarm gadījuma Kubernetes pārvaldei tiek realizētas arī papildus komandas. Kubernetes gadījumā ir nepieciešams privāto reģistru pieslēgšanās informāciju pieglabāt ar iebūvēto noslēpumu sistēmu, nevis to piefiksēt Portainer interfeisā, kas bija iespējams izmantojot Docker Swarm. To pēc tam vajag pievienot pašai vides deklarācijai, savukārt tā izveidošanu realizē sekojošā komanda: "kubernetes_k3s_secret".

Tāpēc ka Kubernetes ir dažādi noslēpumu tipi, atkarībā no ievadītajām vērtībām, pašas komandas formāts tiek uzģenerēts dinamiskā veidā:

```
command_input = KubernetesSecretRequest.read(input)
secret_type = command_input.secret_type
secret_name = command_input.name
secret_values = command_input.values
secret_value_string = ""
for secret_value in secret_values:
    secret_value_string += " --" + secret_value
typer.echo("Will set new secret: " + str(secret_name) + " of type: " +
str(secret_type), err=True)
```

Piemēram, sekojošā komanda tiek izmantota, lai pievienotu pieslēgšanās datus privātajam Docker reģistram, ko nodrošina GitLab Registry [23]:

```
python astolfo.py task run kubernetes_k3s_secret 1.worker.catboi.net
'{"py/object": "tasks.task_kubernetes_k3s_secret.KubernetesSecretRequest",
"secret_type": "docker-registry", "name": "registrycredentials", "values":
["docker-server=https://registry.kronis.dev", "docker-username=KronisLV",
"docker-password=mPT4G5wAePZ5ge44i9FC", "docker-
email=kristians@kronis.dev"]}'
```

Komandas izpildes rezultātā tiks izpildīta sekojošā noslēpuma izveides komanda uz attālinātā servera:

```
kubectl create secret docker-registry registrycredentials  --docker-
server=https://registry.kronis.dev --docker-username=KronisLV --docker-
password=mPT4G5wAePZ5ge44i9FC --docker-email=kristians@kronis.dev
```

Pēc tam būs iespējams uz to atsaukties, vižu deklarācijām pievienojot "imagePullSecrets" lauku ar atsauci uz attiecīgo nosaukumu, kurš noslēpumam tika izveidots:

```
spec:
containers:
  image:
registry.kronis.dev/rtu1/kvps5_masters_degree_covid_1984/covid1984:latest
imagePullSecrets:
- name: registrycredentials
```

Nepieciešamība pēc šāda veida konfigurācijas sarežģī darbu ar Kubernetes, taču šis uzdevums zināmā mērā atļauj mazināt Portainer integrācijas trūkumus, jo privātos reģistrus pats rīks atļauj konfigurēt tikai izmantojot Docker Swarm. Šeit noslēpumā var tikt izmantoti uzģenerētie privātie piekļuves marķieri (angliski "token"), ko pēc tam var atsaukt.

Papildus šīm konfigurācijas iespējām, Kubernetes ir arī vairāki lokālie izstrādes rīki, piemēram Lens [24], kuru var palaist lokāli un pieslēgties attālinātajam klasterim, līdzīgi kā izmantojot iebūvētās komandas un "kubectl" rīku. Lai realizētu šo pieslēgšanos, ir nepieciešama attiecīgā informācija par klasteri, ko glabāt "kubeconfig" datnē uz servera. Ir izveidots arī uzdevums, kurš šo datni atļauj lejupielādēt kā datni uz lokālo sistēmu caur SFTP: "kubernetes_k3s_kubeconfig".

Tas strādā līdzīgi iepriekšminētajai funkcionalitātei, kas nosūta datni caur SFTP uz attālināto sistēmu, tikai šoreiz process notiek pretējā virzienā, datnei tiekot saglabātai lokālajā "files" mapē:

```
try:
    remote_file = "/etc/rancher/k3s/k3s.yaml"
    local_file = "kubeconfig.yml"
    sftp_download(loaded_host, local_file, remote_file)
    complete_stage(stages, "DownloadKubeconfig", successful=True,
command="", output="")
    command_output.downloaded_kubeconfig = True
except Exception as e:
    complete_stage(stages, "DownloadKubeconfig", successful=False,
command="", output="")
    raise e
```

1.6.5 Čaulas darbību izsaukšana

Papildus specifiskajām Docker Swarm un Kubernetes darbībām, tiek realizēta arī loģika, kas atļauj izsaukt čaulas komandas pa tiešo, tā ļaujot izpildīt loģiku, kas vēl nav realizēta kā atsevišķi uzdevumi. Šajā gadījumā gan jārēķinās, ka katras papildus komandas izsaukšanai vajadzēs vēl vienu rīka komandas izsaukšanu, kas nozīmēs jaunu SSH savienojuma izveidi un tam pagaidām nebūs optimāla veiktspēja, pat ja tas atļaus ar rīku ātri izsaukt čaulas komandas, kurām vēl nav paredzēti atsevišķi uzdevumi.

Tas ir labs veids, kā realizēt mijiedarbību ar programmatūru uz attālinātā servera ar pašu rīku, piemēram, ja mēs gribam izgūt Kubernetes pieejamos serverus, kuri ir pievienoti klasterim, ir iespēja izsaukt sekojošo komandu:

```
docker run --rm astolfo task run shell_execute 1.worker.catboi.net
'{"py/object": "tasks.task_shell_execute.ShellExecuteRequest",
"shell_command": "kubectl get nodes"}' 2>/dev/null
```

Pats kods ir garāks kā SSH izsaukšana pa tiešo, taču tajā pat laikā tas seko datu formātam, ko izmanto pārējais rīks (1.32. att.), kas būs noderīgi gadījumā, ja nākotnē rīkam tiks izstrādāts lokāli palaižams tīkla interfeiss.


```
{
  "py/object": "tasks.task_shell_execute.ShellExecuteResponse",
  "shell_command": "kubectl get nodes",
  "command_output": [
    "NAME                 STATUS   ROLES    AGE   VERSION",
    "db.catboi.net        Ready    <none>   30m   v1.19.5+k3s2",
    "tester.catboi.net    Ready    <none>   29m   v1.19.5+k3s2",
    "1.worker.catboi.net  Ready    master   31m   v1.19.5+k3s2",
    "2.worker.catboi.net  Ready    <none>   31m   v1.19.5+k3s2"
  ],
  "command_executed": true
}
```


1.32. att. Paraugs čaulas komandas izpildei ar rīku

Līdzīgi var vienu pēc otras arī ķēdēt vairākas komandas, taču pagaidām nav standartizēta veida kā apstrādāt šo komandu izvadi, jo šie dati paši par sevi netiek tālāk filtrēti vai organizēti objektos ar laukiem, pagaidām visa izvade nonāk zem "command_output" lauka:

```
docker run --rm astolfo task run shell_execute 1.worker.catboi.net
'{"py/object": "tasks.task_shell_execute.ShellExecuteRequest",
"shell_command": "echo nodes:; kubectl get nodes; echo deployments:;
kubectl get deployments"}' 2>/dev/null
```

Tāpat šādā ievadē vajadzētu aizstāt "" simbolus ar atsoļa sekvencēm, kas sarežģītu lietošanu.

1.6.6 Datņu sistēmas darbības

Rīks atļauj arī pārvaldīt direktorijas uz attālinātā servera, ar komandām to izveidošanai un rekursīvai dzēšanai: "directory_create" un "directory_delete". Šīm komandām vienmēr tiek pievienots arī "--verbose" parametrs, kurš nozīmē, ka tiks izvadīti visi dzēstie vai izveidotie ceļu elementi:

```
command = "rm --verbose -rf " + str(directory_path)
command = "mkdir --verbose -p " + str(directory_path)
```

Piemēram, izveidojot dziļāku ceļa struktūru, būs pieejama izvade par katru direktoriju, kas tiek izveidota (1.33. att.), savukārt ja direktorijā ir vairākas datnes, tad dzēšot pašu direktoriju tiks uzskaitīti visi no tiem (1.34. att.).


```
Executing: mkdir --verbose -p /test/directory/path/that/contains/many/folders
Stages: {
  "CreateDirectory": {
    "py/object": "tasks.utils.stages.Stage",
    "name": "CreateDirectory",
    "description": "Will create the necessary directory structure",
    "order": 0,
    "successful": true,
    "command": "mkdir --verbose -p /test/directory/path/that/contains/many/folders",
    "output": [
      "mkdir: created directory '/test'",
      "mkdir: created directory '/test/directory'",
      "mkdir: created directory '/test/directory/path'",
      "mkdir: created directory '/test/directory/path/that'",
      "mkdir: created directory '/test/directory/path/that/contains'",
      "mkdir: created directory '/test/directory/path/that/contains/many'",
      "mkdir: created directory '/test/directory/path/that/contains/many/folders'"
    ],
    "finished": true
  }
```


1.33. att. Dziļākas datņu struktūras izveide


```
Executing: rm --verbose -rf /test
Stages: {
  "DeleteDirectory": {
    "py/object": "tasks.utils.stages.Stage",
    "name": "DeleteDirectory",
    "description": "Will delete the given directory",
    "order": 0,
    "successful": true,
    "command": "rm --verbose -rf /test",
    "output": [
      "removed '/test/directory/path/that/contains/many/folders/test_file_2'",
      "removed '/test/directory/path/that/contains/many/folders/test_file_1'",
      "removed directory '/test/directory/path/that/contains/many/folders'",
      "removed directory '/test/directory/path/that/contains/many'",
      "removed directory '/test/directory/path/that/contains'",
      "removed directory '/test/directory/path/that'",
      "removed directory '/test/directory/path'",
      "removed directory '/test/directory'",
      "removed directory '/test'"
    ],
    "finished": true
  }
}
```


1.34. att. Direktorijas dzēšanas izvades paraugs

Rīks realizē arī iespēju lejupielādēt vai augšupielādēt datnes no "files" direktorijas, kurā var uzglabāt Docker Swarm un Kubernetes nepieciešamos vižu deklarāciju datnes vai arī citas datnes, kuras nepieciešams nogādāt uz attālināto serveri. To realizē sekojošie uzdevumi: "file_download", "file_send".

Pastāv arī "file_delete" uzdevums, kurš ļauj izdzēst vienu datni, neriskējot izpildīt rekursīvās darbības, tā mazinot iespēju izdzēst par daudz datu.

1.7 Izstrādātā rīka konteinerizācija

Pats rīks arī tika konteinerizēts, lai spētu darboties uz datoriem, kuros nav pieejams Python interpretators, tostarp atļautu darbību arī uz ierīcēm ar Windows operētājsistēmu, pat ja paša rīka izstrādes un testēšanas laikā tika izmantota tikai GNU/Linux - konteinerizācija šajā gadījumā ļauj arī izvairīties no iespējamām problēmām, kas varētu rasties citas operētājsistēmas platformas specifikas dēļ (piemēram, datņu rindiņu beigu simbolu dēļ).

Rīka konteinerizācijai tika izveidots "Dockerfile" - datne, kuru padot Docker rīkam, no kura tiek izveidots OCI saderīga formāta attēls, kuru pēc tam var palaist ar Docker, Podman, vai kādu no citiem saderīgajiem rīkiem. Attiecīgajā datnē ir aprakstītas darbības, kuras ir nepieciešamas konteinera izveidei, kā arī iekļauj informāciju par to, uz kura pamata attēla to bāzēt:

```
FROM python:3.9.0-slim-buster

WORKDIR /app

COPY ./src/requirements.txt /app
RUN echo "Installing dependencies: " && pip install -r requirements.txt

COPY ./src /app

ENTRYPOINT ["python", "astolfo.py"]
```

Šajā gadījumā komandas nodrošina darbu "/app" direktorijā, tā to skaidri nodalot no pārējās sistēmas datnēm, kas būs konteinerā, kā arī sākotnēji pārkopējot tikai "requirements.txt" datni, kurā ir aprakstītas visas nepieciešamās pakas, kuras vajag rīka būvēšanai ar atbilstošajām to versijām:

```
bcrypt==3.2.0
cffi==1.14.4
click==7.1.2
colorama==0.4.4
cryptography==3.2.1
demjson==2.2.4
importlib-metadata==3.1.1
jsonpickle==1.4.2
paramiko==2.7.2
pycparser==2.20
PyNaCl==1.4.0
python-dotenv==0.15.0
shellingham==1.3.2
six==1.15.0
typer==0.3.2
zipp==3.4.0
```

Pateicoties šim saturam, pēc tam pašas lietotnes atkarības ar "pip" rīku tiek lejupielādētas no Interneta un sakonfigurētas, pēc kā atliek vien pārkopēt pašus skriptus no pirmkoda direktorijas "src". Šāda komandu secība atļauj veikt ātrākas konteineru būvēšanas darbības, jo gadījumā, ja nemainīsies "requirements.txt" saturs, varēs atjaunot tikai pēdējo Docker attēla slāni (1.35 att.), t.i. pietiks to izmantot par pamatu un iekopēt jaunās datnes.

1.35. att. Docker attēla slāņu inspekcija ar "dive" rīku

Papildus tam šeit tiek izmantota arī Docker "ENTRYPOINT" komanda, kura nozīmē, ka rīkā vienmēr pēc noklusējuma tiks izsaukta "python astolfo.py" komanda, visu pārējo ievadi padodot kā parametrus komandai tālāk. Piemēram, izpildot sekojošo komandu, patiesībā tiks izsaukta "python astolfo.py info" komanda:

```
docker run --rm
registry.kronis.dev/rtu1/kvps5_masters_degree_astolfo_cloud_servant/
astolfo_cloud_servant info
```

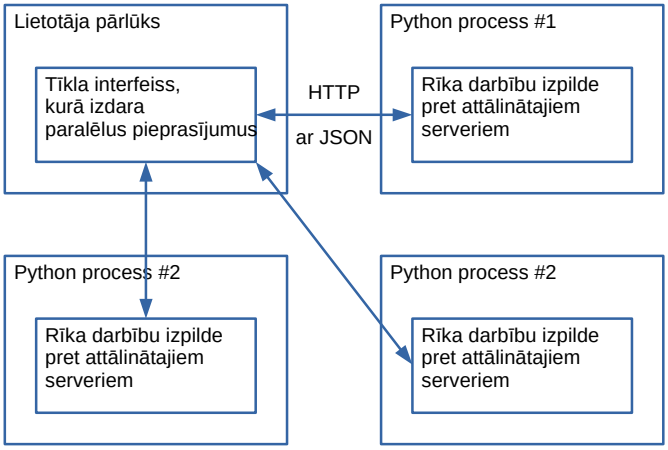
Tāpēc, ka pats GitLab repozitorijs ir publisks, arī pats reģistrs ir pieejams publiski, kā dēļ šo attēlu var arī pamēģināt izpildīt lokāli, nebūvējot rīku no pirmkoda. Šī rīka versijas tiek pārbūvētas uz GitLab CI/CD serveriem pie katru jauno izmaiņu nonākšanas Git repozitorijā, pateicoties nepārtrauktās integrācijas ieviešanai (4. pielikums).

1.8 Rīka izstrādes kopsavilkums

Izstrādes rezultātā veiksmīgi tika iegūts rīks, kuru ir iespējams izmantot kā alternatīvu Ansible vai Salt rīkiem serveru pārvaldei un kurš ar dažu komandu palīdzību atļauj uz serveriem uzinstalēt visu, kas nepieciešams konteineru tehnoloģiju pielietošanai un klasteru inicializācijai ar Docker Swarm un Kubernetes (K3s distribūcijas) orķestratoru palīdzību.

Pats rīks ar nepārtrauktās integrācijas rīku palīdzību arī veiksmīgi tiek atjaunināts un ievietots publiski pieejamā konteineru reģistrā, iespējams, nākotnē tas tiks pārvietots arī uz Docker Hub un papildus tam kodu būs iespējams publicēt GitHub, to licencējot ar LGPL [25] vai citu līdzīgu licenci, tā ļaujot rīka izmantošanu un funkcionalitātes pielāgošanu bez maksas jebkuram interesantam, sekojot brīvās programmatūras principiem. Izstrādes ietvaros tika arī sniegti vienkārši piemēri Bash čaulas skriptiem, kuri ļauj rīku izmantot, lai inicializētu abu minēto orķestratoru konteineru klasterus, apkopojot visas tam nepieciešamās komandas (tie ir pieejami "script_examples" mapē Git repozitorijā, kā arī 5. un 6. pielikumā).

Pašlaik gan rīkam ir daži trūkumi, piemēram, tas pats par sevi neļauj darbības paralelizēt, taču to būtu iespējams panākt, tā izsaukumus paralelizējot, piemēram, ja tiks izstrādāts tīkla interfeiss, tas pats var realizēt šo paralelizāciju (1.36. att.).



1.36. Iespējamās rīka paralelizācijas paraugs

Šāda JSON API un nebloķējošā tīkla servera izvēle, kā arī tīkla interfeisa izstrāde gan pagaidām ir ārpus darba tvēruma, ņemot vērā tā apjoma ierobežojumus.

2. LIETOTŅU KONTEINERIZĀCIJA

Lai praktiski pārbaudītu to, vai ar rīku sagatavotā infrastruktūra ir funkcionēt spējīga un lai salīdzinātu orķestratoru darbības specifiku, tika nolemts uz tās palaist testa lietotnes, kas atklās šīs atšķirības un darbības gaitu. Taču, runājot par pašu lietotņu sagatavošanu palaišanai konteineros - konteinerizāciju, ir nepieciešams pieminēt aspektus, kas jāņem vērā, lai nodrošinātu šādu lietotņu pareizu un konsistentu darbību.

Praksē nepietiek paņemt jebkuru eksistējošo lietotni, no tās uzbūvēt konteineri un sagaidīt, ka attiecīgās lietotnes funkcionalitāte konteinerizētā veidā nebūs traucēta, vai pat ka to varēs palaist vairākās paralēlās instancēs. Šeit var noderēt gan zināšanas par pašu konteineru dzīves ciklu, to tehnoloģiju nodrošinātajām iespējām attiecībā uz vides mainīgo padošanu, noslēpumu pārvaldīšanu, tīklošanu un datņu sistēmas dalīšanu, gan arī par attiecīgo programmēšanas valodu un ietvaru praksēm, kur var būt nepieciešamas konfigurācijas izmaiņas vēlamās darbības nodrošināšanai.

Maģistra darba ietvaros tika realizēta tīkla lietotne, kas atļauj vairāku tās instanču paralēlu palaišanu, kas arī kalpo par piemēru dažām no mākoņrisinājumu izstrādes praksēm un ļauj tās nodemonstrēt ar praktiskiem piemēriem. Par tīkla lietotnes tematiku tika izvēlēta alternatīva pieeja COVID kontaktu fiksēšanai - kontaktu izsekošana ar GPS palīdzību, ar ko saistītos datus var agregēt un apstrādāt centrālajā serverī. Šāda izvēle ļauj gan realizēt tipisku tīkla lietotni, kas apstrādā HTTP pieprasījumus un kurai ir salīdzinoši viegli veikt slodzes testēšanu, simulējot vairākus paralēlos lietotāju pieprasījumus, kā arī ļauj nodemonstrēt paralelizācijas nozīmi lielāka pieprasījumu skaita apkalpošanā. Papildus tam tas ļauj spriest, vai interpretēto valodu piedāvātā veiktspēja ir adekvāta šādu sistēmu izstrādē, it īpaši situācijās, kurās izstrādes darbiem pieejamais laiks ir ierobežots, kā dēļ zemāka abstrakcijas līmeņa valodas izmantošana varētu nebūt labi piemērota problēmas risināšanai.

Izstrādātās parauga lietotnes pirmkods ir pieejams Git repozitorijā:
https://git.kronis.dev/rtu1/kvps5_masters_degree_covid_1984

2.1 Mākoņrisinājumu izstrādes prakses

Izstrādājot risinājumus darbībai mākoņdatošanas vidē, ir jārēķinās ar to, ka varētu tikt palaistas vairākas lietotnes instances, kas nozīmē, ka nevar rēķināties ar pašas lietotnes vides datņu sistēmas izmantošanu ilgstošai datu uzglabāšanai. Līdzīgi, ja tiek ģenerēti dati, ar kuriem tiks validēti vai pārbaudīti ienākošie pieprasījumi, piemēram JWT marķieri, kā alternatīva sesiju sīkdatņu (angliski "cookies") uzturēšanai HTTP pieprasījumu kontekstā, tad šie dati nedrīkst tikt uzģenerēti un atrasties atmiņā tikai uz viena no instancēm, jo tad tos nevarēs izmantot citas un darbība būs nekorekta.

Konteineru izmantošanas gadījumā attiecīgā konteinera datņu sistēma kļūst nepieejama pēc tā izpildes beigām un jauna konteinera palaišanas gadījumā tam par pamatu tiks izmantots konteinera sākotnējais attēls - lai nodrošinātu šo datu ilgstošu uzglabāšanu, ir nepieciešams izmantot citus mehānismus, piemēram, Docker disk vietas (angliski "volumes") [26], sasaisti ar paša servera datņu sistēmu (angliski "bind mount") vai arī tīkla datņu sistēmas, piemēram NFS, kas var atļaut datu uzglabāšanu un piekļuvi tiem pat vairāku serveru izmantošanas gadījumā. Tajā pašā laikā ir jāņem vērā datņu sistēmas ierobežojumi, jo īpaši ja ar to darbojas vairāki serveri, kas var novest pie nepareizas datu ierakstīšanas vai datu rakstīšanas darbību konfliktiem, vai arī iespējas, ka datne ir nobloķēta, jo tajā jau notiek rakstīšanas darbība, kā dēļ attiecīgajām lietotnēm būs lielāka nepieciešamība uzmanību pievērst šādu kļūdu apstrādei.

Līdzīgi arī audita izvadi šādos gadījumos nav ieteicams izvadīt datnēs, jo tad būtu nepieciešams vai nu darboties ar iepriekšminētajām datu rakstīšanas atomiskuma problēmām un to ietekmes mazināšanu, vai arī censties augšupielādēt vairākas datnes, kuras pēc tam mēģināt agregēt kopējā notikumu apskatā, kas varētu radīt sarežģījumus ar šo datu apkopošanu atbilstoši to ierakstīšanas laikam. Tā vietā industrijā tiek izmantota speciāla programmatūra šādu laika sēriju datu agregēšanai, kas vēlāk atļauj tos apstrādāt tālāk, kā arī veikt meklēšanu pēc noteiktiem kritērijiem vai satura, piemēram Graylog [27] un Elastic Stack [28]. Pastāv iespēja arī izmantot pašu operētājsistēmu nodrošināto auditēšanas funkcionalitāti, piemēram, Docker atļauj izmantot arī syslog programmatūru [29], lai audita

datus piefiksētu tajā, bet atsevišķu komponenšu instalācijas vai liela daudzuma konfigurācijas.

Tāpēc konteinerizēto lietotņu kontekstā bieži vien audita izvadi nogādā uz standartizvades kanāliem: STDOUT un STDERR, kuru saturu pēc tam konteineru tehnoloģijas atļauj nogādāt tālāk. Pastāv arī komandas, kas ļauj šo audita izvadi apskatīt reālajā laikā, gan individuālu servisu, gan arī atsevišķu konteineru kontekstā, kas, savukārt, var atvieglot to izstrādi un atkļūdošanu.

Tā dēļ izstrādātajā risinājumā, kurš ir bāzēts uz Ruby tehnoloģijas, audita izvades darbībām arī tiek izmantota standartizvade, pēc noklusējuma lietojot iebūvēto auditēšanas funkcionalitāti:

```
def perform(scheduled_frequency)
    begin
        ...
    rescue => exception
        Rails.logger.info "Exception occurred in heatmap data generation!"
        Rails.logger.info exception.backtrace
        @@task_busy = false
    ensure
        ...
    end
end
```

Savukārt visam Ruby on Rails ietvaram tika papildināta auditēšanas konfigurācija, lai šo izvadi nogādātu uz STDOUT izvades kanālu:

```
config.log_formatter = ::Logger::Formatter.new
logger           = ActiveSupport::Logger.new(STDOUT)
logger.formatter = config.log_formatter
config.logger    = ActiveSupport::TaggedLogging.new(logger)
```

Tas nozīmē, ka šīs izmaiņas ieviest pašā lietotnē ir salīdzinoši viegli un šo pašu konfigurāciju var izmantot gan izstrādes laikā, gan arī palaižot lietotni uz vides, produkcijā, neliekot uzturēt atsevišķus konfigurācijas paraugus dažādajām vidēm.

Mākoņrisinājumu gadījumā gan tomēr var rasties konfigurācijas atšķirības starp dažādajām vidēm, ja tiek runāts par testēšanas vidēm un produkcijas vidēm, kuras pieslēgsies citām datu bāzēm vai citām ārējām sistēmām ar atšķirīgiem kontiem, pat ja pati infrastruktūras un sistēmas topoloģija abos gadījumos būs pēc iespējas līdzīgāka un pat ja vidēs tiks izpildīts viens un tas pats kods, kas būs radies no tā paša konteinera. Lai neliktu

konfigurāciju obligāti nolasīt no datnes, kā gadījumā būtu šīs konfigurācijas datnes jāpiegādā uz visiem serveriem, kuros tie ir nepieciešami, vai arī tie būtu jānogādā iekš konteineriem, konteineru tehnoloģijas atļauj konfigurācijas nolasīšanu no vides mainīgajiem.

Izstrādātās lietotnes ietvaros arī tās darbībai nepieciešamā konfigurācija tika nolasīta no vides mainīgajiem, gan Ruby on Rails lietotnes gadījumā, gan arī uz K6 bāzētās slodzes testēšanas lietotnes gadījumā, tā neliekot domāt par to, kādas tehnoloģijas attiecīgās lietotnes izmanto, bet nodrošinot konsistentu konfigurācijas padošanas formātu tām visām. Noklusējuma vērtību ieviešana gan ir atkarīga no attiecīgās tehnoloģijas:

```
production:
  <<: *default
  host: <%= ENV.fetch("POSTGRES_HOST") { "localhost" } %>
  database: <%= ENV.fetch("POSTGRES_DB") { "covid1984" } %>
  username: <%= ENV.fetch("POSTGRES_USER") { "covid1984" } %>
  password: <%= ENV.fetch("POSTGRES_PASSWORD") { "covid1984" } %>
```

Šeit gan jāpiemin, ka produkcijas vidēm un biznesa mērķu sasniegšanai domātu lietotņu realizācijā nav ieteicams sensitīvos datus, t.i. privātās atslēgas saturu un dažādu kontu lietotājvārdus un paroles padot caur vides mainīgajiem, jo šie dati netiek slēpti vai šifrēti, jebkurš ar pieeju attiecīgajam konteineram varētu tos nolasīt (2.1. att.).

```
kronislv@catbook:~/Documents/kvps5_masters_degree_covid_1984$ docker exec covid1984_postgis printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=ef08ed3a95b2
POSTGRES_PASSWORD=covid1984
POSTGRES_USER=covid1984
POSTGRES_DB=covid1984
LANG=en_US.utf8
PG_MAJOR=11
PG_VERSION=11.10
PG_SHA256=13e6d2f80662fe463bc7718cdf0de6a9ec67fc78afcc7a3ae66b9ea19bb97899
PGDATA=/var/lib/postgresql/data
POSTGIS_VERSION=3.0.3
POSTGIS_SHA256=9aae25d46dc8b124f6e8a35886edcf9bd23a3ab049090edd8335b9c7324cae74
HOME=/root
kronislv@catbook:~/Documents/kvps5_masters_degree_covid_1984$
```

2.1. att. Vides mainīgo izvadīšana no lokāli pacelta Docker konteinera

Šādās situācijās ieteicams izmantot alternatīvos rīkus, kuri ļauj šos datus uzglabāt drošākā veidā un tos pieprasīt pēc vajadzības, piemēram, Hashicorp Vault [30], taču šīs parauga lietotnes ietvaros konfigurācija tā netika sarežģīta. Var izmantot arī pašu konteineru tehnoloģiju nodrošinātās iespējas noslēpumu uzglabāšanai, kas tika nodemonstrēts ar "imagePullSecrets" norādīšanu 1.6.4. nodaļā.

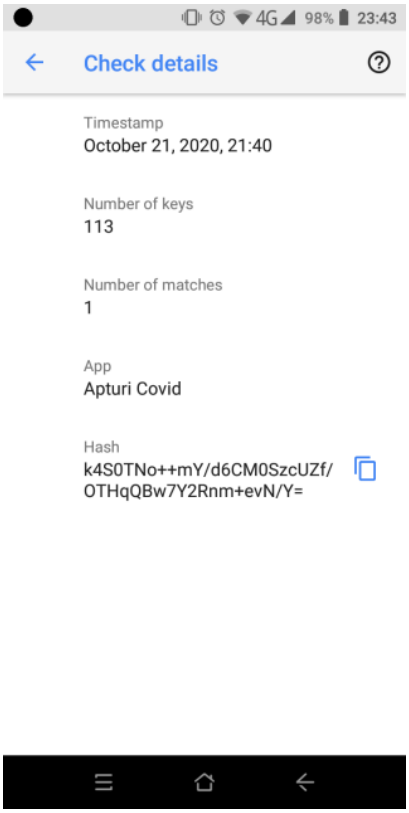
2.2 Uz GPS bāzēts risinājums COVID inficēto izsekošanai

Pašai lietotnei tika izvēlēta ar COVID saistīta tematika. Pašlaik esošie risinājumi, piemēram Apturi Covid [31], kuri izmanto Google un Apple izstrādāto funkcionalitāti kontaktu fiksēšanai [32], izmanto galvenokārt decentralizētu pieeju, kas ļauj individuālajām ierīcēm piefiksēt kontaktus ar citu ierīču šifrētajām identifikācijas atslēgām, savukārt gadījumā kurā kāds tiek inficēts ar vīrusu, ir iespējams šīs ierīces ģenerētās atslēgas publicēt, tā lietotnes lietotājiem paziņojot par to, ka tie ir nonākuši kontaktā ar kādu, kurš tagad ir inficēts (2.2. att.).

2.2. att. Paraugs Apturi Covid paziņojumam par kontaktu

Tomēr šāda pieeja nenodrošina iespēju skaidri identificēt, kad un kur ir noticis šis kontakts, kas var būt problemātiski gadījumā, ja lietotne ir uzinstalēta tikai vienam no cilvēkiem kādā grupā (piemēram, paziņu vai kolēģu starpā), kas neļauj par iespējamo nepieciešamību pašizolēties paziņot citiem, jo nav informācijas par to, vai kontakta brīdī tie ir vai nav bijuši tuvumā.

Ierīce pati varētu piefiksēt kaut vai laika zīmogu katras atslēgas kontakta piefiksēšanai, taču lai nodrošinātu privātumu, arī pašiem viedtālruņa lietotājiem šādi dati nav tiešā veidā pieejami, piemēram, Android parāda tikai kontaktu skaitu (2.3. att.).



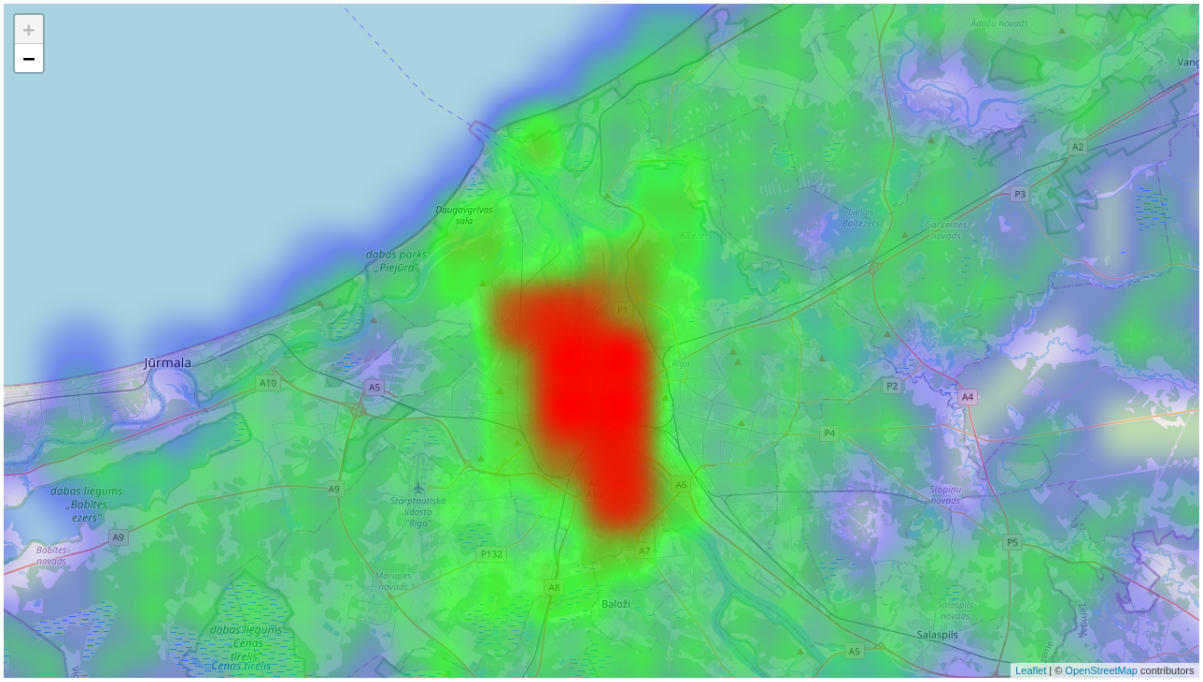
2.3. att. Atslēgu skaita attēlojums Android operētājsistēmā

Trūkums šai pieejai ir tāds, ka ar izvēlēto realizāciju lietotne tiešā veidā palīdz tikai tiem, kuri izmanto to savos tālruņos un kuru ierīces to vispār atbalsta: pārāk vecās ierīcēs, kā arī dažu ražotāju ierīcēs šī funkcionalitāte pagaidām nav pieejama [31]. Pārējie no tās labumu gūst tikai netiešā veidā, t.i. SPKC darbam tiekot atvieglotam un samazinoties iespējai nonākt kontaktā ar kontaktpersonām, tām pašizolējoties.

Tomēr, pēc CSB datiem par iedzīvotāju skaitu [33] sanāk, ka Latvijā ir aptuveni 1'907'675 iedzīvotāji, savukārt saskaņā ar SPKC doto informāciju [34], lietotne ir tikusi lejupielādēta 155'000 reižu. Pat pieņemot, ka lietotni lieto 100% no tiem, kuri ir to

lejupielādējuši, tas nozīmē, ka tā ir pieejama aptuveni 8,1% Latvijas iedzīvotājam - mazāk kā 1 no katriem 10, kas traucē tai efektīvi darboties un sasniegt savus mērķus. Tomēr, ja lietotnes instalācija un lietošana ir pilnībā brīvprātīga, ar šādiem realizācijas trūkumiem ir jārēķinās.

Tāpēc maģistra darba ietvaros sīkāk nepētot privātuma aspektus, tika nolemts papētīt tehniskos izaicinājumus alternatīvam risinājumam, kurš izmantotu GPS datus, kurus apstrādātu centralizētā veidā. Šāds risinājums atļautu piefiksētos atrašanās vietas datus agregēt un pēc tam ģenerēt "karstuma kartes" ar izvēlētu precizitāti, piemēram, tā parādot, kuros pilsētas rajonos ir statistiski lielāka iespēja nonākt kontaktā ar kādu inficēto (2.4. att.).

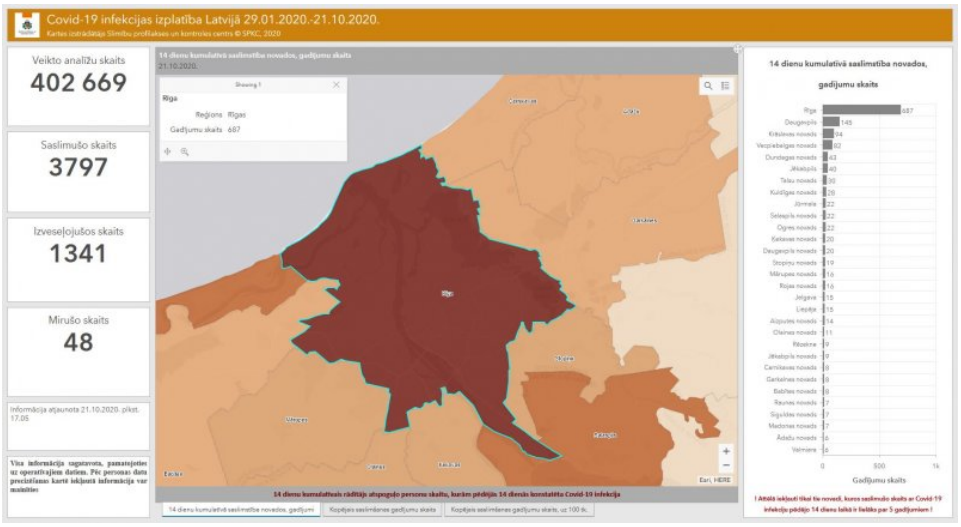


Data last renewed: 2021-01-03 15:30:41 (every 30 seconds). Displaying 1013 data points for a total of 2502 logged locations.

2.4. att. Paraugs uzģenerētajiem testa datiem

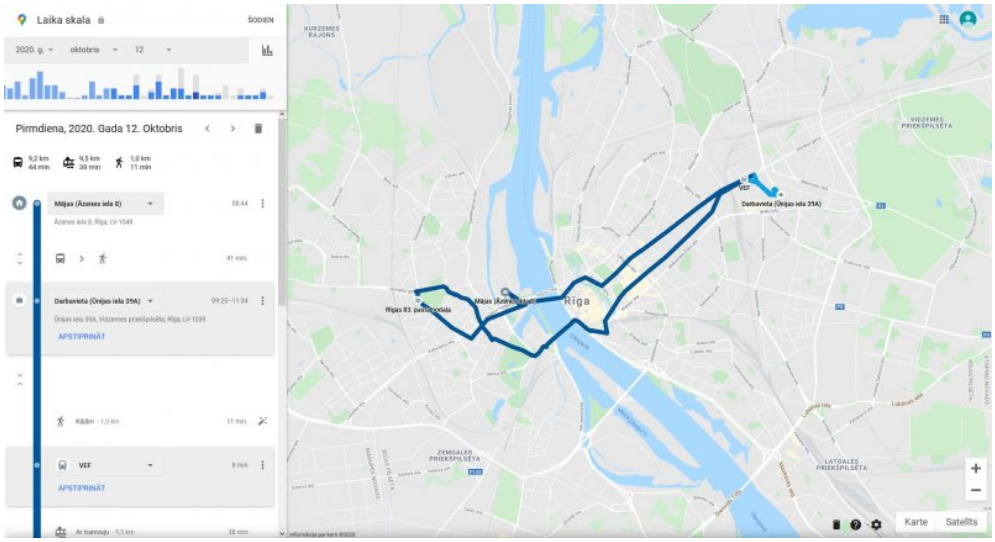
Tas varētu katram iedzīvotājam palīdzēt lēmumu pieņemšanas procesā, piemēram, izvēloties uz kuru veikalu pilsētā doties, lai iepirktu pārtiku vai citas nepieciešamās preces, tajā pat laikā nesniedzot informāciju par specifisko inficēto personu identitāti vai to dzīves vietu, pieņemot, ka statistika tiek ģenerēta par atrašanās vietu ar pietiekami lielu iedzīvotāju blīvumu. Tas gan varētu radīt sarežģījumus gadījumā, ja šāda veida informācija tiktu ģenerēta par valsts lauku apgabaliem, kuros ir zems iedzīvotāju blīvums.

Dotajā brīdī SPKC jau ir padarījis pieejamas kartes, kas parāda šāda veida informāciju, tikai nevis par specifiskām vietām, bet gan par Latvijas reģioniem kopumā, balstoties uz piefiksēto statistiku (2.5. att.) [35].



2.5. att. Paraugs SPKC publicētajām kartēm

Līdzīgi, liela daļa viedtālruņu operētājsistēmu jau ietver funkcionalitāti, kura fonā apkopo datus par lietotāju pārvietošanos un atrašanās vietas datiem noteiktā laikā, piemēram, Google Timeline (2.6. att).

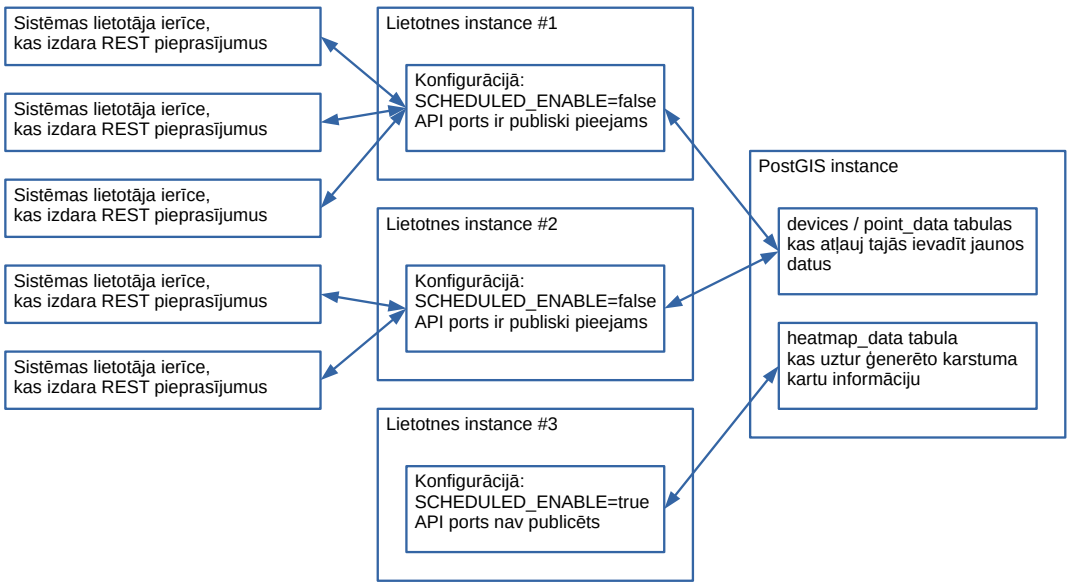


2.6. att. Paraugs Google Timeline datiem, dodoties uz sapulci darbā

Piezīme: students, darbojoties SIA "Autentica" uzņēmumā [36], bija viens no aptuveni 90 speciālistiem, kuri piedalījās Apturi Covid izstrādē, taču šajā darbā parādītā un izstrādātā lietotne ir nesaistīta ar šo projektu un kalpo par vienkāršotu paraugu uz GPS bāzētai kontaktu izsekošanas sistēmai, ilustrējot gan konteinerizācijas priekšrocības tīkla lietotņu izstrādē, gan izceļot GPS datu izmantošanas un šādas centralizācijas īpatnības.

2.3 Modulārā monolītiskā struktūra

Pašai lietotnes izstrādei tika izvēlēta monolītiska struktūra, visu funkcionalitāti iekļaujot vienā konteinerā, taču tam atļaujot šo funkcionalitāti vai nu ieslēgt vai izslēgt atkarībā pēc vajadzības, ar funkcionalitātes karogu (angliski "feature flags") palīdzību. Tas ļauj izvēlēties vienu konteinera instanci, kurai ļaut fonā apstrādāt plānotos procesus, piemēram, karstuma karšu pārģenerāciju, savukārt pārējām instancēm šo funkcionalitāti atslēgt lai nerastos konflikti, bet atļaut apkalpot mainīgo skaitu HTTP pieprasījumu (2.7. att.).



2.7. att. Paraugs modulārajai lietotnes struktūrai

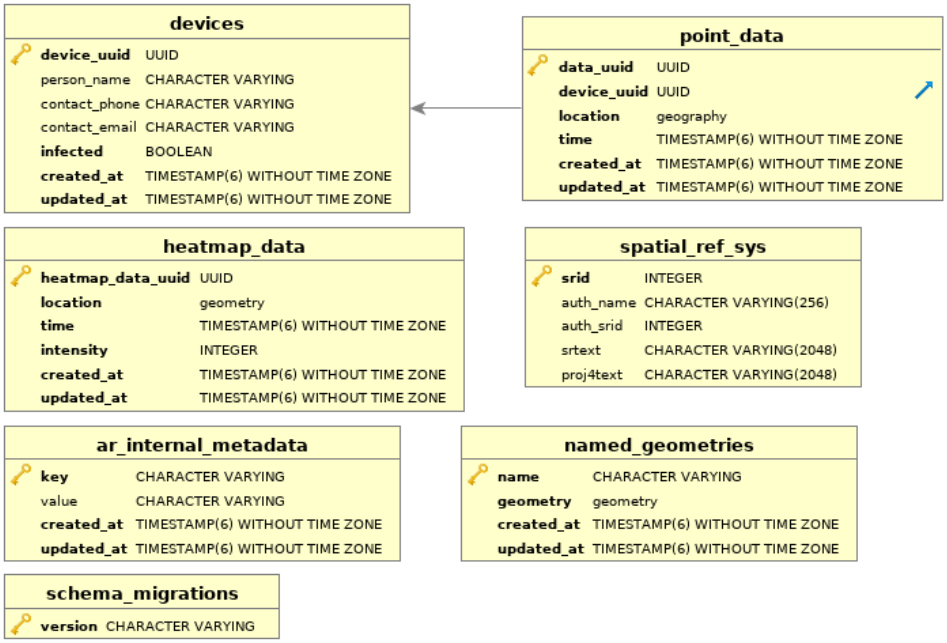
Tas ļauj vienkāršot jaunu lietotnes versiju izveidi un tās konfigurācijas palaišanu, izmantojot vairākus konteinerus ar dažādiem mērķiem, tajā pat laikā neradot infrastruktūrā to sarežģītību, kas rastos mikro servisu arhitektūras dēļ, kur katrs no servisiem būtu jāpārvalda

atsevišķi. REST interfeiss ietver sevī informāciju par ierīcēm, kurām savukārt var būt piereģistrēti punktu dati, kas reprezentē ierīces atrašanos noteiktā ģeogrāfiskajā punktā kādā datumā (2.8. att.), testa nolūkiem atļaujam arī datu nolasīšanu. Līdzīgi ir pieejami arī uzģenerētie karstuma kartes punktu dati, kurus iespējams izmantot kartes attēlošanai, izmantojot leaflet.js bibliotēku.

```
            Prefix Verb   URI Pattern                                       Controller#Action
              root GET    /                                                 map#index
           version GET    /version/:id(.:format)                            version#show
device_point_data GET    /devices/:device_id/point_data(.:format)          point_data#index
                  POST   /devices/:device_id/point_data(.:format)          point_data#create
           devices GET    /devices(.:format)                                devices#index
                  POST   /devices(.:format)                                devices#create
      heatmap_data GET    /heatmap_data(.:format)                           heatmap_data#index
```

2.8. att. Paraugs pieejamiem HTTP ceļiem lietotnē

Pašā DB struktūras gadījumā tā ir visai vienkārša un izmanto atsauci starp ierīcēm un šiem punktu datiem, bet citādi starp tabulām nav nepieciešama tieša datu sasaiste. Papildus tam, Ruby on Rails pievieno arī tabulu DB migrāciju izsekošanai "schema_migrations", kā arī ir tabula "named_geometries", kurā ir ievadītas Latvijas robežas, lai nodrošinātu, ka visi ievadītie dati tiešām būs Latvijas teritorijā, kā arī ir vēl dažas sistēmas tabulas darbam ar ģeospatiālo informāciju (2.9. att).



2.9. att. DB shēmas diagramma

Šeit var pieminēt arī to, ka pašā lietotnē ir ieviesta kontrole uz to, lai jaunus datus noteiktai ierīcei varētu pieminēt, tikai zinot iepriekšējā ieraksta identifikatoru, kā arī pašu ierīces identifikatoru:

```
# POST /devices/:device_id/data
@point_data = PointData.new(user_params)
@point_data.data_uuid = SecureRandom.uuid

parent_device = Device.find_by(device_uuid: @point_data.device_uuid)
if parent_device == nil
    render json: { error: 'No device exists for UUID.' }, status: 400
end

if @point_data.previous_data_uuid.empty? # is new
    previous_data = PointData.where(device_uuid: @point_data.device_uuid)
    if !previous_data.empty?
        render json: { error: 'No existing previous data UUID provided,
necessary because not first record.' }, status: 400
        return
    end
else
    previous_data = PointData.find_by(device_uuid:
@point_data.device_uuid, data_uuid: @point_data.previous_data_uuid)
    if previous_data == nil
        render json: { error: 'No existing previous data UUID
provided.' }, status: 400
        return
    end
end

factory = RGeo::Cartesian.factory
new_point = factory.point(@point_data.location_x, @point_data.location_y)
@point_data.location = new_point

in_latvia = PointData.check_point_in_latvia(@point_data.location_x,
@point_data.location_y)
if !in_latvia
    render json: { error: 'Point is not in Latvia.' }, status: 400
    return
end

if @point_data.save!
    render json: @point_data
else
    render json: { error: 'Unable to create new device.' }, status: 400
end
```

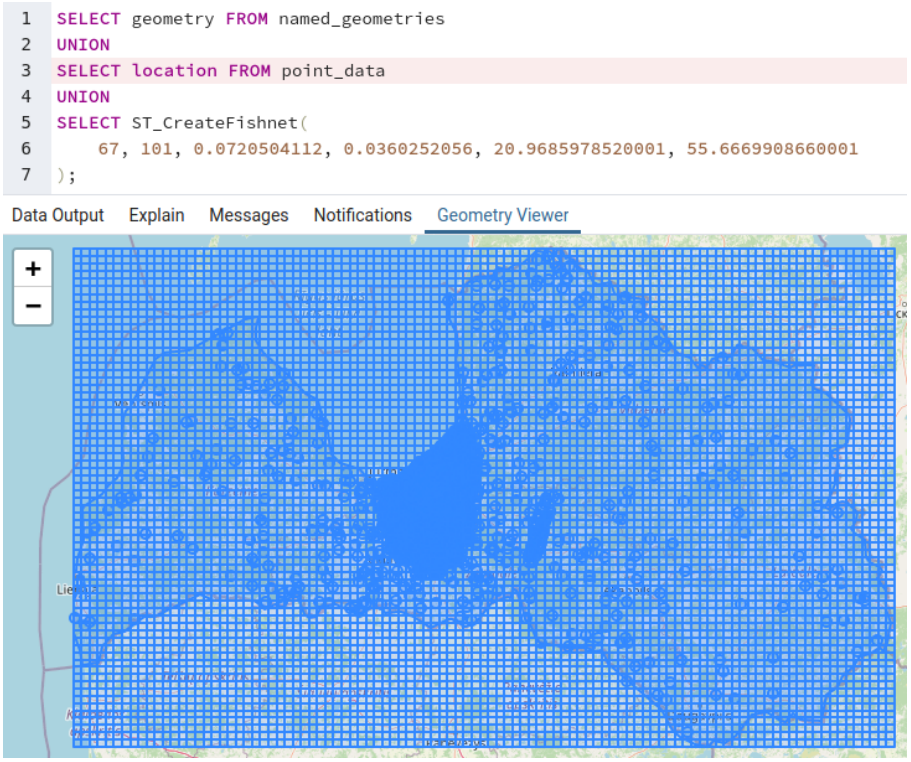
Savukārt tāpēc, ka šie identifikatori ir nevis skaitļi, kurus ir salīdzinoši viegli uzminēt, bet gan UUID datu tips [37], tas nozīmē ka šīs vērtības noviltot būs grūtāk, kas nedaudz uzlabo sistēmas drošību, līdzīgi tam, kā strādā blok ķēdes tehnoloģijas.

2.4 Datu apstrāde datu bāzu vadības sistēmā

Pati lietotne veic datu ierakstīšanu datu bāzē, taču datu karšu ģenerācija notiek, izsaucot datu bāzes vadības sistēmā saglabāto procedūru (angliski "stored procedure") ar lietotnes konfigurācijā norādītiem parametriem, ar noteiktu biežumu:

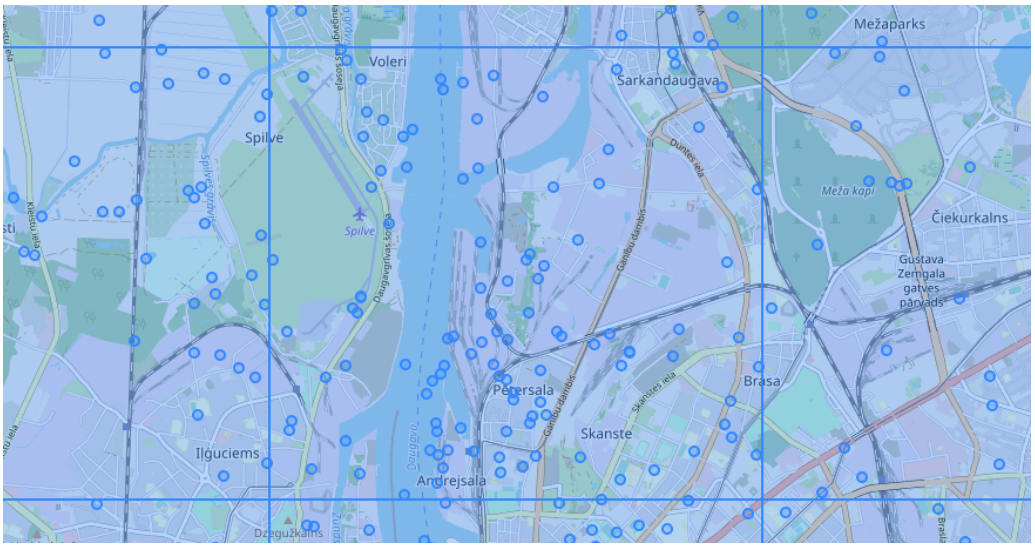
```
def update_heatmap
  Rails.logger.info "Updating heatmap data..."
  HeatmapData.regenerate_heatmap(
    HEATMAP_CELL_RESOLUTION_KM,
    HEATMAP_MINUTES_AGO,
    HEATMAP_UNIQUE_BATCHES_TO_KEEP
  )
  Rails.logger.info "Finished updating heatmap!"
end
```

Šīs darbības ietvaros, visa Latvija tiek sadalīta dažus kilometrus lielos apgabalos, kuros tiks atlasīti izvēlētajā laika periodā piefiksētie atrašanās vietu dati, kas kalpos par pamatu to apkopojuma ierakstu izveidei "heatmap_data" tabulā (2.10. att.). Tas nodemonstrē paraugu, darbību izpildēm DBVS (angliski "in-database processing"), ko arī industrijā mēdz izmantot.



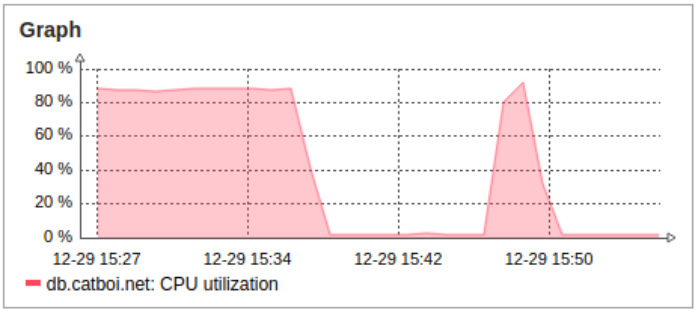
2.10. att. Paraugs funkcijai, kas sadala teritoriju apgabalos

Tas atļauj katram apgabalam iegūt skaitli, kurš raksturo tā intensitāti, atkarībā no tā, cik personas tajā ir bijušas (2.11. att.), liedzot lietotājiem piekļuvi šiem datiem tiešā veidā, bet reprezentējot to, cik drošs ir attiecīgais apgabals, lai gan šo apgabalu iedalījums neseko administratīvajām robežām (ja netiek izmantoti papildus dati, piemēram, OpenStreetMap, kuros šīs robežas būtu).



2.11. att. Paraugs nejauši ģenerētajiem datiem par ierīcēm

Pati saglabātās procedūras realizācija ir pieejama 7. pielikumā, kuru apskatot kļūs redzams, ka tās realizācijā netiek izmantots neviens cikls. Tomēr, testējot sistēmu, tika ievērots, ka tieši šī funkcionalitāte ļoti noslogo DBVS, piemēram, mēģinot šos datus pārģenerēt pēc slodzes testiem (2.12. att.).

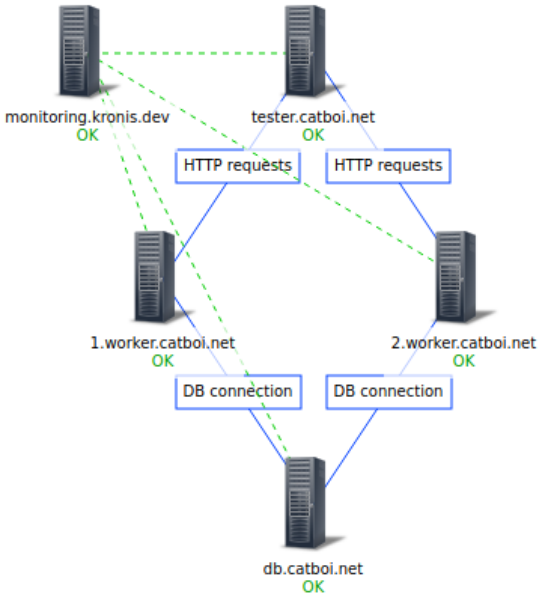


2.12. att. Paraugs DB noslodzei, pārģenerējot datus (labajā pusē)

Tas liek domāt, ka praksē visticamāk vajadzētu atsevišķu DBVS instanci šai darbībai.

2.5 Docker Swarm vides deklarācija

Lietotnes palaišanai uz Docker Swarm orķestratora tika izstrādāta vides definīcija Docker Compose formātā, kura apraksta 3 lietotnes instanču (1 no kurām izpildīs plānotos procesus) izvietošanu uz pieejamiem īrētajiem privātajiem serveriem (angliski "virtual private server" jeb VPS) (2.13. att.), kā arī pašas PostGIS instances palaišanu konteinerizētā veidā, izmantojot Docker piedāvātās diska vietas datu uzglabāšanai. Šī deklarācija ir pieejama 8. pielikumā.



2.13. att. Paraugs izmantotajai infrastruktūrai

Šī pieeja atļauj ar vienu aprakstu efektīvi izmantot pieejamos resursus, lietotnes izvietojot uz visiem serveriem. Problēmas gan radīja Docker Compose formāts, jo konteineriem netika atļauts izmantot 100% procesora resursu, lai sistēmu nepadarītu neatsaucīgu uz SSH pieslēgumiem. Tādā gadījumā ir jāzin, cik procesora kodoli ir pieejami un cik atvēlēt uzdevumiem - Compose formāts nepiedāvā procentuālas vērtības norādīšanu:

```
deploy:
  resources:
    limits:
      cpus: "0.90"
```

2.6 Kompose rīks deklarāciju konvertēšanai un Kubernetes vides deklarācija

Lai nodrošinātu pēc iespējas tuvāku vides deklarāciju, Docker Swarm paredzētais Docker Compose formāta datne tika pa tiešo konvertēts priekš izmantošanas Kubernetes. Tas tika panākts ar Kompose rīka [38] palīdzību (2.14. att.).

```
kronislv@catbook:~/Downloads/definitions$ kompose convert -f covid1984.yml -o covid1984kubernetes.yml
WARN Restart policy 'unless-stopped' in service covid1984_app_scheduled is not supported, convert it to 'always'
INFO Service name in docker-compose has been changed from "covid1984_app_scheduled" to "covid1984-app-scheduled"
WARN Restart policy 'unless-stopped' in service covid1984_app is not supported, convert it to 'always'
INFO Service name in docker-compose has been changed from "covid1984_app" to "covid1984-app"
WARN Restart policy 'unless-stopped' in service covid1984_postgis is not supported, convert it to 'always'
INFO Service name in docker-compose has been changed from "covid1984_postgis" to "covid1984-postgis"
```

2.14. att. Kompose izmantošanas paraugs

Rezultātā tika iegūta vides deklarācija, kas ir pieejama 8. pielikumā pēc labojumu veikšanas tai. Kā minēts iepriekš, nācās pievienot "imagePullSecrets" noslēpumu, lai atļautu piekļuvi konteineru attēliem, kas ir pieejami privātajā konteineru reģistrā. Papildus tam, pēc noklusējuma tiek izvadītas atmiņas vērtības zinātniskajā formātā, kas var salauzt citus rīku (piemēram, Portainer rīks nav spējīgs tās nolasīt un salūzt tīkla interfeiss):

```
resources:
  limits:
    cpu: 900m
    memory: "3145728e3"
```

Papildus tam, šie limiti patiesībā atsauksies uz resursiem, kurus Kubernetes orķestrators rezervēs, gadījumā, ja nav norādīts "requests" elements, tāpēc jāsecina, ka šajā gadījumā Kompose rīks var dažas vērtības konvertēt nepareizi. Izlabotais fragments no deklarācijas izskatās sekojoši (skaitliskās vērtības tika noapaļotas):

```
resources:
  requests:
    cpu: 100m
    memory: "512000000"
  limits:
    cpu: 900m
    memory: "3145000000"
```

Taču pat jā citādi šī konversija bija veiksmīga un ievērojami ietaupīja laiku, Docker Swarm deklarācijā ir 70 rindiņas, savukārt priekš Kubernetes nepieciešamas 234, kas padara šīs deklarācijas daudz grūtāk manuāli uzrakstāmas, kā arī daudz grūtāk pārskatāmas. To gan var skaidrot ar Kubernetes datu formāta sarežģītību, taču tas var traucēt šī rīka izmantošanai gadījumos, kad ir ierobežots daudzums laika konfigurācijai un nav piemēru.

3. SLODZES TESTĒŠANA

Maģistra darba izstrādes ietvaros tika izveidoti arī slodzes testi, kuri simulē sistēmas lietotājus, kas pārvietotos pa Latvijas teritoriju un iesūtītu savus atrašanās vietas datus regulārā intervālā, simulējot izstrādātas lietotnes darbību. Šo testu mērķis ir pārbaudīt, cik lielu pieprasījumu skaitu ir iespējams apstrādāt ar vienu un vairākām tīkla lietotnes instancēm lai izvērtētu, vai horizontālā mērogošana var palīdzēt efektīvi mazināt interpretēto valodu izvēles radītos veiktspējas riskus. Papildus tam šāda veida testēšana parādītu, vai ar iepriekšminēto rīku sagatavotā infrastruktūra ir spējīga veiksmīgi darboties un vai gan Docker Swarm, gan Kubernetes orķestratori ir tikuši veiksmīgi uzinstalēti uz serveriem, kā arī vai nav palikušas iepriekšminētās DNS problēmas saistībā ar Docker Swarm darbību.

Izstrādāto slodzes testu pirmkods ir pieejams Git repozitorijā: https://git.kronis.dev/rtu1/kvps5_masters_degree_covid_1984_load_test

3.1 k6 rīka izmantošana testēšanai

Testu realizācijai tika izvēlēts k6 rīks [39], kurš pašu testu saturu ļauj rakstīt JavaScript valodā, savukārt izpildes vide ir rakstīta Google uzņēmuma izstrādātajā Go programmēšanas valodā, pašu JavaScript dzini iekļaujot tajā. Tas ļauj šim rīkam realizēt salīdzinoši labu veiktspēju, tajā pašā laikā atļaujot izmantot augstāka abstrakcijas līmeņa programmēšanas valodu, tā ievērojami saīsinot testēšanas skriptu izstrādes laiku.

Līdzīgi augstākminētajam COVID kontaktu izsekošanas risinājumam arī šī lietotne savu konfigurāciju saņem no vides mainīgajiem un arī tiek palaista uz serveriem konteinerizētā formātā. Tas ļauj to viegli izvietot vai nu vairākās paralēlās instancēs vai arī uz vairākiem serveriem, taču šoreiz tika izmantots viens serveris, kuram ir vairāk skaitļošanas resursi kā tiem, uz kuriem ir pati Ruby on Rails lietotne. Arī paša slodzes testu servera resursu patēriņš tiek uzraudzīts, lai nodrošinātu, ka tā resursu trūkuma dēļ testēšanas rezultāti netiek negatīvi ietekmēti.

Šajā gadījumā pati konfigurācija ir aprakstīta visai īsi, piedāvājot arī noklusējuma vērtības, ko var izmantot lietotnes testēšanai gadījumā, ja tā ir tikusi pacelta lokāli. Pašā konfigurācijā šoreiz tiek norādītas 4 testēšanas stadijas, kas progresīvi palielina testēšanas

slodzi ar vairāk un vairāk virtuālajiem lietotājiem, ar mērķi ļaut izpētīt, vai resursu patēriņš abu orķestratoru izmantošanas gadījumā arī pieaug līdzīgi:

```
function getOrDefault(value, defaultValue) {
    return value === undefined ? defaultValue : value;
}

let kilometerDistance = 0.0090063014;

export let options = {
    debug: getOrDefault(__ENV.DEBUG, false),
    stages: [
        { duration: parseInt(getOrDefault(__ENV.TEST_DURATION_SECONDS, 15)
* 0.25) + 's', target: Math.max(1, parseInt(getOrDefault(__ENV.TEST_USERS,
1) * 0.25)) },
        { duration: parseInt(getOrDefault(__ENV.TEST_DURATION_SECONDS, 15)
* 0.25) + 's', target: Math.max(1, parseInt(getOrDefault(__ENV.TEST_USERS,
1) * 0.50)) },
        { duration: parseInt(getOrDefault(__ENV.TEST_DURATION_SECONDS, 15)
* 0.25) + 's', target: Math.max(1, parseInt(getOrDefault(__ENV.TEST_USERS,
1) * 0.75)) },
        { duration: parseInt(getOrDefault(__ENV.TEST_DURATION_SECONDS, 15)
* 0.25) + 's', target: Math.max(1, parseInt(getOrDefault(__ENV.TEST_USERS,
1) * 1.00)) },
      ],
    url: getOrDefault(__ENV.TEST_URL, 'http://localhost:3000'),
    centerLocation: {
        x: getOrDefault(__ENV.TEST_CENTER_LOCATION_X, 24.10),
        y: getOrDefault(__ENV.TEST_CENTER_LOCATION_Y, 56.96)
    },
    maximumDistance: getOrDefault(__ENV.TEST_MAXIMUM_DISTANCE,
kilometerDistance * 1000),
    secondsBetweenRequests:
getOrDefault(__ENV.TEST_SECONDS_BETWEEN_REQUESTS, 1),
    minimumDistancePerMinute:
getOrDefault(__ENV.TEST_MINIMUM_DISTANCE_PER_MINUTE, 60 / 60 *
kilometerDistance), // 60 km/h
    maximumDistancePerMinute:
getOrDefault(__ENV.TEST_MAXIMUM_DISTANCE_PER_MINUTE, 90 / 60 *
kilometerDistance) // 90 km/h
};
```

Augstāk esošais kods sevī arī ietver loģiku, kura būs nepieciešama GPS datu ģenerēšanai, nodrošinot simulētā lietotāja pārvietošanos ar noteiktu ātrumu. Pašu uzdevumu izpildi ir iespējams palaist ar vienu komandu, attiecīgajam rīkam liekot izpildīt noteiktā skriptā definētos uzdevumus:

```
k6 run load-test.js
```

Tas ļauj to viegli izsaukt, arī ja tas netiek lietots konteinerizētā veidā, lokālai testēšanai.

3.2 GPS datu ģenerēšana un pārvietošanās simulēšana

Testu ietvaros tiek uzģenerēti GPS pozīcijas dati kā sākuma pozīciju izvēloties punktu, kurš ir nejaušā attālumā un ir izvēlēts nejaušā virzienā no Rīgas, ko izmanto par centrālo punktu. Šo koordinātu iegūšanai tiek izmantotas vienkāršas darbības ar 2D vektoriem - nejauša vektora izveide, tā normalizācija, kā arī tā komponenšu pareizināšana ar nejauši izvēlētu skaitli dotajās robežās, kurš reprezentē attālumu no centrālā punkta.

Daļai no uzģenerētajiem datiem tiek samazināts attālums līdz centrālajam punktam, lai uzskatāmi parādītu kartes datu attēlojumu gadījumos, kur noteiktā punktā ir lielāks piereģistrēto datu punktu blīvums, šajā gadījumā, ap galvaspilsētu:

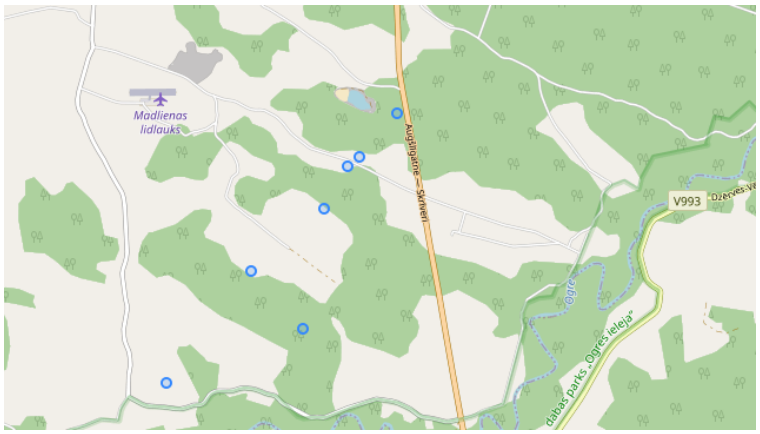
```
function initializeStartingPosition() {
    // make most of the requests center around the initial point (capital)
    let distanceFromStart = Math.random() * options.maximumDistance;
    let randomValue = Math.random();
    if (randomValue < 40) {
        distanceFromStart *= 0.5;
    } else if (randomValue < 30) {
        distanceFromStart *= 0.5;
    }

    // initialze the starting position of the requst
    let locationStart = new Vector(options.centerLocation.x,
options.centerLocation.y);
    let randomOffset = new Vector(0,
0).random().normalize().multiply(distanceFromStart);
    locationStart.add(randomOffset);

    return locationStart;
}
```

Protams, šāda pieeja nozīmē, ka daļa no šiem punktiem atradīsies ārpus Latvijas sauszemes teritorijas, kā dēļ ja tiek par šādiem datiem saņemta validācijas kļūda, tad šī inicializācijas loģika tiks izsaukta vēlreiz, izvēloties jaunu atrašanās vietu, kas varētu būt derīga:

```
if (registeredData.hasOwnProperty('error') && registeredData.error ==
"Point is not in Latvia.") {
    data.locationNext = new Vector(0,
0).add(initializeStartingPosition());
}
```

Šādu kļūdainu datu ievades mēģinājumi tomēr ir pieļaujami, jo tie varētu notikt gadījumos, kuros cilvēki neatrodas valsts teritorijā, kas nozīmētu, ka šie dati kartē nav jāiekļauj.

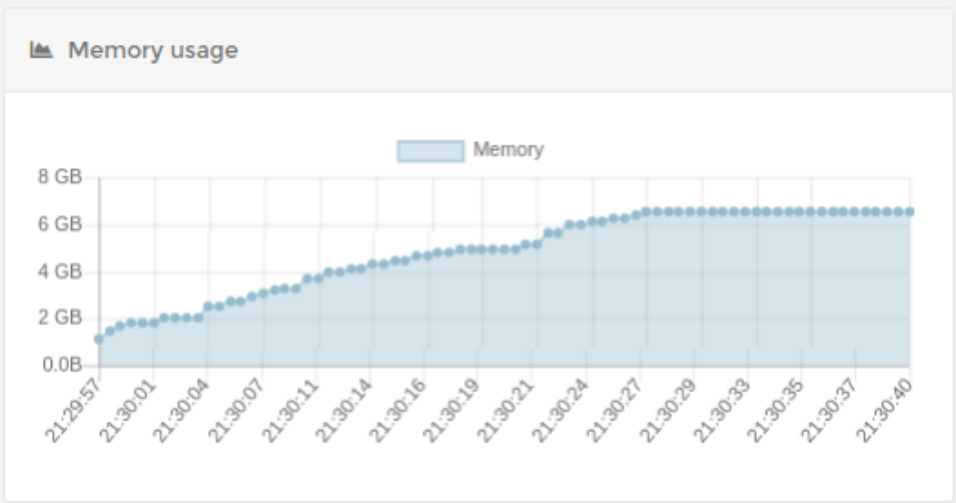
Pati datu validācijas loģika gan ir ļoti vienkārša, kas, lai gan nozīmē to ka testu izpilde būs salīdzinoši ātra un nebūs pārāk prasīga pret aparatūru, tajā pašā laikā novedīs pie situācijām, kur individuālie datu punkti nesekos tam, kā pārvietotos cilvēki - liela daļu to neatradīsies uz ceļiem, lai gan kustības ātrums attiecīgo ierakstu atjaunināšanai var būt izvēlēts tāds, ar kuru pārvietotos automašīna (3.1. att.). Tas gan netraucē veikt testēšanu, jo sistēmas testēšanas nolūkos svarīgāks ir punktu izvietojuma blīvums, ne to izvietojuma specifika.



3.1. att. Paraugs nejauši ģenerētajiem atrašanās vietas datiem

Par spīti šīm optimizācijām, tomēr izskatās, ka k6 rīks ir visai prasīgs pret atmiņas resursiem, kas ierobežo maksimālo virtuālo lietotāju skaitu, kurus ir iespējams norādīt testu izpildei. Tas atļauj testēšanu veikt ar mazām aizturēm starp individuālajiem atrašanās vietu saglabāšanas mēģinājumiem, taču neatļauj veikt testēšanu ar lielu skaitu paralēlo virtuālo lietotāju, jo katram lietotājam ir nepieciešama sava konfigurācijas kopija ar sākuma izpildes vietu, kustības ātrumu, unikālo identifikatoru, kā arī kur uzglabāt tam piešķirto ierīces identifikatoru.

Tas noved pie tā, ka ar vairāk kā 1500 lietotāju tiek izmantota visa serverim pieejamā atmiņa, kas var novest pie procesa izpildes pārtraukšanas (3.2.. att.).



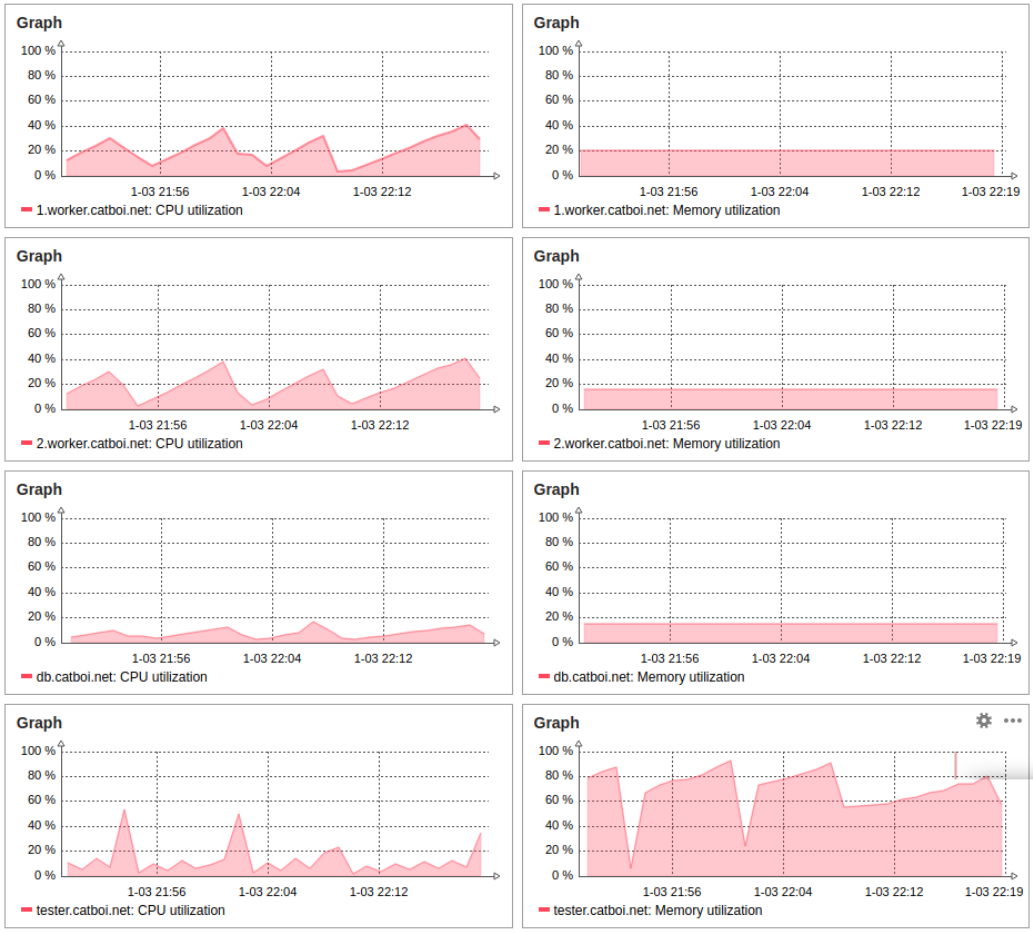
3.2. att. Testēšanas konteinera atmiņas patēriņš ar 2500 lietotājiem

Šos ierobežojumus primāri rada finansiāli faktori, jo minētie serveri tiek īrēti no Time4VPS pakalpojumu nodrošinātāja, kas tika izvēlēts dēļ tā, ka nodrošina uz KVM bāzētu virtualizāciju un izmanto serveru procesorus - šajā gadījumā Intel Xeon Gold 6132, kas savā darbībā varētu sniegt labāku priekšstatu par mākoņrisinājumu darbināšanu, kā tā simulācija virtuālajās mašīnās lokāli, izmantojot personīgo datoru procesorus.

Papildus tam šajā gadījumā ir vieglāk pārinstalēt operētājsistēmu un to no jauna sagatavot klasteru testēšanai, KVM arī nodrošinot garantētu piekļuvi noteiktam resursu daudzumam, kas varētu būt problemātiskāk ar personīgajiem datoriem, kur nepietiekama procesora dzesēšana var novest pie takts frekvences izmaiņām, kas negatīvi ietekmētu testēšanas rezultātu precizitāti.

3.3 Zabbix izmantošana datu agregēšanai

Papildus šādai serveru izvēlei tika izmantota arī Zabbix programmatūra uz atsevišķa servera, lai periodiski pārbaudītu resursu patēriņu - diska rakstīšanas un lasīšanas darbību radīto noslodzi, noslodzi uz procesoru kā arī operatīvās atmiņas patēriņu, kā arī citas līdzīgās vērtības. Zabbix šajā gadījumā arī atļauj izveidot pārskatus, kuros iespējams reālajā laikā attēlot grafikus ar šiem raksturlielumiem, kā arī šos datus apskatīt ilgāku laika intervālu kontekstā, ja nepieciešams (3.3. att.).



3.3.. att. Paraugs Zabbix pārskatam par serveru stāvokli testēšanas laikā

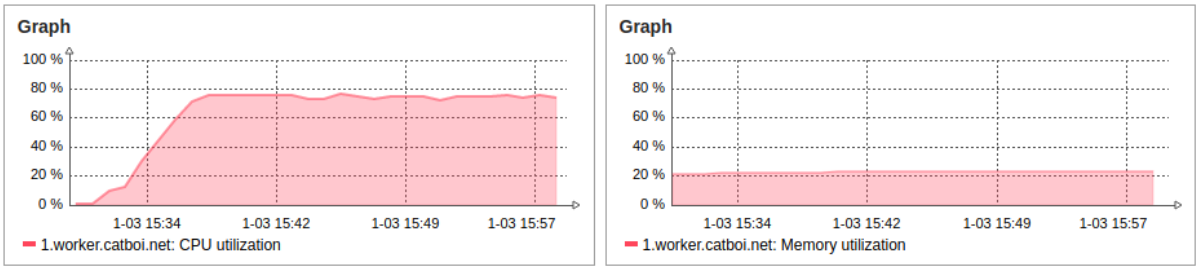
Papildus tam, Zabbix rīks var atļaut nodrošināt ziņojumu izsūtīšanu gadījumā, ja kāds no parametriem pārāk ilgi ir ārpus pieļaujamām robežām (piemēram, procesora noslodze pārsniedz 80% 30 minūtes pēc kārtas), tā infrastruktūras administratoram ļaujot paziņot par problēmām, kas var būt radušās.

Šīs izstrādes ietvaros tika izvēlēts Zabbix arī tā iemesla dēļ, ka rīks uzrauga visu serveri kopumā, kas ļaus pārliecināties par reālo resursu patēriņa stāvokli noteiktiem orķestratoriem, ne tikai individuālo konteineru radīto resursu patēriņu - tas var būt svarīgi, ja pats orķestrators patērē salīdzinoši lielu resursu daudzumu, tā traucējot darboties pašām lietotnēm uz servera.

3.4 Horizontālās mērogošanas ietekme uz lietotnes veiktspēju

Testi tika izpildīti ar 1000 virtuālajiem lietotājiem, kas lokācijas datus nosūta ar 1 sekundes aizkavi starp tiem. Praksē, visticamāk, šādai sistēmai pietiktu ar datu nosūtīšanu ik pa minūtei vai pat ik pēc 5 minūtēm, taču šādi var simulēt līdzīgu situāciju lielākam lietotāju skaitam - piemēram, 1000 virtuālie lietotāji ar 1 sekundes aizkavi atbilst aptuveni 60'000 lietotājiem ar 60 sekunžu aizkavi, jo vienīgā atšķirība starp dažādiem lietotājiem testa ietvaros ir sākotnējā jaunās ierīces piereģistrēšana, kas testa izpildes laikā nerada papildus slodzi. Šajā gadījumā testi ilga 30 minūtes, kā ietvaros pakāpeniski tika palielināta slodze, līdz tika sasniegts pilnais virtuālo lietotāju skaits. Abos gadījumos tika izmantots Docker Swarm orķestrators, jo tam ir raksturīgs salīdzinoši mazs resursu patēriņš.

No sākuma šī testēšana tika veikta pret klasteri, kurā lietotne ir pacelta bez paralelizācijas - kurā ir tikai viena tās instance, pret kuru tiek izpildīti visi pieprasījumi. Šajā gadījumā lietotnes servera procesoram bija ievērojama noslodze (3.4. att.), pat ja atmiņas patēriņš palika salīdzinoši zems.

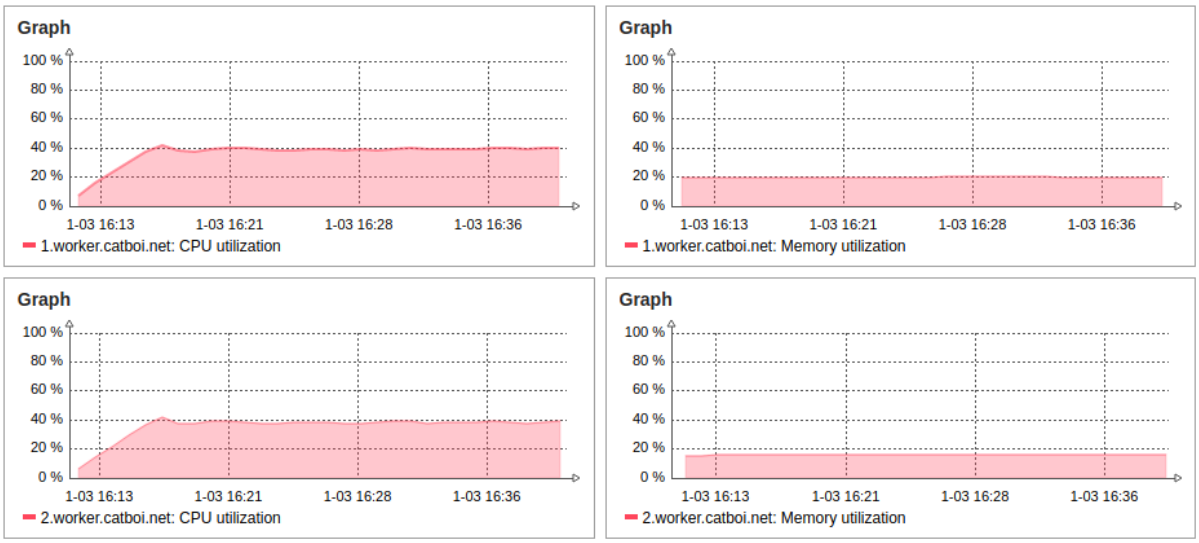


3.4. att Paraugs testēšanai pret vienu instanci

Var ievērot, ka noslodze nesasniedza 100%, kas nenotika dēļ ieviestajiem konteineru resursu ierobežojumiem, kas gan Docker Swarm gadījumā, gan Kubernetes gadījumā atļauj izvairīties no situācijas, kurā šīs noslodzes dēļ serveris būtu neresponsīvs. Taču kā minēts 2.5 nodaļā, nav iespējams šādus limitus norādīt kā procentuālu vērtību, kas nozīmē, ka šādas

kontroles ieviešanai ir nepieciešams zināt vairāk par infrastruktūru, uz kuras konteineri tiek izvietoti, kas ir pretstatā infrastruktūras abstrahēšanas principiem.

Tomēr, palielinot paralelizāciju uz divām instancēm, kļūst skaidri redzams, ka slodze uz katru individuālo serveri samazinās (3.5. att.), jo pieprasījumi var tikt dalīti starp tiem, pateicoties round-robin pieejai DNS ierakstu izveidei. Tomēr arī šajā gadījumā ir ievērojams, ka noslodze ir vērsta, galvenokārt, uz pašu procesoru, nevis operatīvās atmiņas patēriņu.



3.5. att. Paraugs testēšanai ar divām instancēm

To ir iespējams skaidrot ar faktu, ka tekošajā lietotnes realizācijā nav nepieciešamība ilgstoši uzglabāt datus operatīvajā atmiņā - katra pieprasījuma dati tiek apstrādāti nekavējoties, šajā gadījumā ilgstoši neliekot atmiņā glabāt HTTP sesiju informāciju, kas paralēlu lietotnes instanču gadījumā arī nebūtu pieļaujami, jo tam vajadzētu izmantot ārējo datu glabātuvi (vai nu DBVS, vai arī "atslēga-vērtība" pāru glabātuvi, piemēram Redis).

Pat veicot testēšanu ar līdzīgu slodzi, ko abi serveri ir spējīgi apstrādāt bez servera resursu izsīkuma, starp abām no šīm konfigurācijām ir ievērojamas atšķirības attiecībā uz to, cik bieži radās kļūdas (3.1. tabula).

3.1. tabula

Paraugs apstrādātajiem pieteikumiem dažādos paralelizācijas scenārijos

Scenārijs	Izdarītie pieprasījumi	Pieprasījumi, kas netika apstrādāti	Izdevušies pieprasījumi, %
Viena instance	322831	34080	89,4%
Divas instances	315729	16911	94,3%

To var skaidrot ar to, ka papildus paša servera resursiem, katrai lietotnes instancei ir ierobežots arī pieejamo paralēli izmantojamo resursu daudzums - piemēram, operētājsistēmas pavedieni, kurus var izmantot HTTP pieprasījumu apstrādei, kā arī DB savienojumi, kuri ir pieejami, pieņemot, ka tiek izmantota savienojumu grupa (angliski "thread pool"), lai katram pieprasījumam neliktu radīt jaunu savienojumu, kam būtu negatīva ietekme uz DB noslogojumu. Tas nozīmē, ka nepietiek vien ar paralēlu instanču pacelšanu, bet ir jādomā par attiecīgās lietotnes integrāciju ar tai pieslēgto programmatūru - vai katrai no instancēm ir atvēlēts adekvāts pavedienu un DB savienojumu skaits, kā arī vai pašā DB konfigurācijā norādītais maksimālais atļautais savienojumu skaits netiks pārsniegts.

To apstiprināja arī izvade no paša k6 rīka, kas ļāva sīkāk apskatīt dažādus izpildes raksturlielumus gan vienas instances slodzes testēšanai (3.6. att), gan divu instanču slodzes testēšanai (3.7. att.).

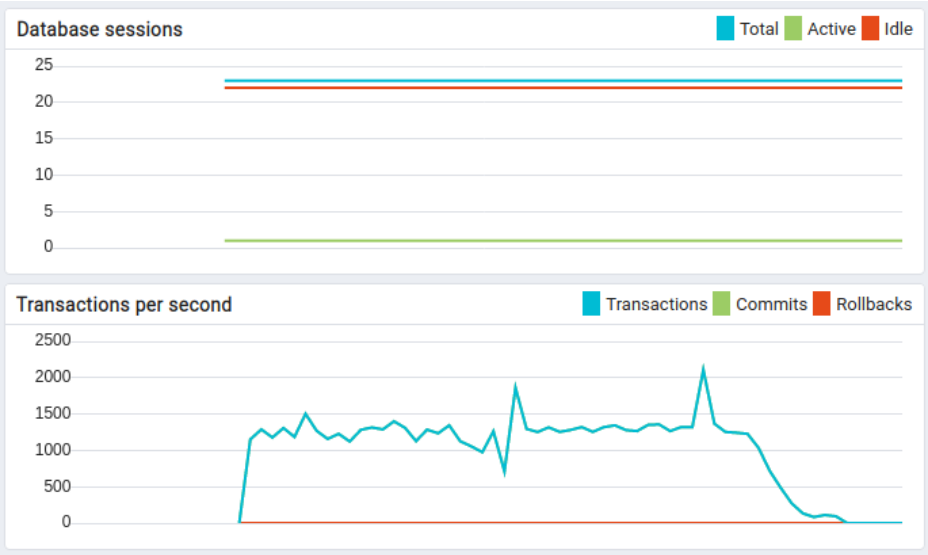
```
data_received..............: 140 MB 77 kB/s
data_sent..................: 106 MB 58 kB/s
http_req_blocked...........: avg=11.84ms   min=0s       med=4.47µs    max=15.65s    p(90)=6.74µs    p(95)=7.97µs
http_req_connecting........: avg=11.84ms   min=0s       med=0s        max=15.65s    p(90)=0s        p(95)=0s
http_req_duration..........: avg=269.7ms   min=0s       med=240.71ms  max=1m0s      p(90)=356.39ms  p(95)=396.54ms
http_req_receiving.........: avg=58.31µs   min=0s       med=54.13µs   max=26.7ms    p(90)=87.6µs    p(95)=101.5µs
http_req_sending...........: avg=31.27µs   min=0s       med=28.08µs   max=28.27ms   p(90)=44.17µs   p(95)=51.74µs
http_req_tls_handshaking...: avg=0s        min=0s       med=0s        max=0s        p(90)=0s        p(95)=0s
http_req_waiting...........: avg=269.61ms  min=0s       med=240.61ms  max=1m0s      p(90)=356.3ms   p(95)=396.46ms
http_reqs..................: 323831 177.669774/s
iteration_duration.........: avg=2.8s      min=500.8ms  med=1.25s     max=1m1s      p(90)=1.72s     p(95)=30.53s
iterations.................: 322831 177.121124/s
location_x.................: avg=24.036967 min=19.507332 med=24.03205  max=28.616512 p(90)=26.430993 p(95)=26.954629
location_y.................: avg=56.932713 min=52.432564 med=57.032741 max=61.459043 p(90)=59.374102 p(95)=59.952803
vus........................: 19     min=0    max=999
vus_max....................: 1000   min=1000 max=1000
```

3.6. att. k6 rīka izvade vienas instances testēšanai

```
data_received..............: 144 MB 79 kB/s
data_sent..................: 110 MB 60 kB/s
http_req_blocked...........: avg=20.23ms   min=0s       med=5.12µs    max=15.65s    p(90)=7.53µs    p(95)=8.91µs
http_req_connecting........: avg=20.23ms   min=0s       med=0s        max=15.65s    p(90)=0s        p(95)=0s
http_req_duration..........: avg=286.69ms  min=0s       med=207.2ms   max=1m0s      p(90)=300.14ms  p(95)=326.57ms
http_req_receiving.........: avg=61.55µs   min=0s       med=55.63µs   max=27.44ms   p(90)=88.44µs   p(95)=102.62µs
http_req_sending...........: avg=34.42µs   min=0s       med=30.24µs   max=34.25ms   p(90)=46.35µs   p(95)=54.12µs
http_req_tls_handshaking...: avg=0s        min=0s       med=0s        max=0s        p(90)=0s        p(95)=0s
http_req_waiting...........: avg=286.6ms   min=0s       med=207.1ms   max=1m0s      p(90)=300.04ms  p(95)=326.48ms
http_reqs..................: 315729 173.22127/s
iteration_duration.........: avg=2.87s     min=506ms    med=1.23s     max=1m1s      p(90)=1.67s     p(95)=30.57s
iterations.................: 314729 172.672631/s
location_x.................: avg=24.272297 min=19.569807 med=24.327056 max=28.533018 p(90)=26.672043 p(95)=27.349534
location_y.................: avg=56.911212 min=52.302545 med=56.947723 max=61.395223 p(90)=59.236021 p(95)=59.949615
vus........................: 8      min=0    max=999
vus_max....................: 1000   min=1000 max=1000
```

3.7. att. k6 rīka izvade divu instanču testēšanai

Šajos rezultātos tiek parādīts tas, ka individuālo pieprasījumu apstrādes laiks 95% gadījumu nepārsniedz 30 sekundes, taču atsevišķos gadījumos tas var aizņemt pat 1 minūti, pie kā noteiktām ierīcēm arī varētu iestāties savienojuma noilgums un tas tikt pārtraukts. Savukārt sīkāk apskatot DB noslogojumu, ir redzams, ka vairums DB savienojumu lielu daļu laika pavada dīkstāvē (3.8. att.), pat ja katru sekundi tiek izveidoti vairāki tūkstoši jaunu transakciju, kas nozīmē, ka visticamāk vaina ir meklējama pašā lietotņu konfigurācijā.



3.8. att. Informācijas izvade par DB noslodzi, testēšanas lielākās slodzes izdarīšanas laikā

To iespējams pārbaudīt, vai nu katrai instancei piešķirot lielāku pavedienu skaitu ar attiecīgā ietvara nodrošināto "RAILS_MAX_THREADS" parametru vai arī vienkārši palaižot lielāku skaitu paralēlo instanču. Otrā pieeja gan rezultētu mazāk efektīvā resursu izlietojumā lieko DB sesiju dēļ, taču tāpat ļautu pārbaudīt, vai vaina ir meklējama pašu lietotņu konfigurācijā.

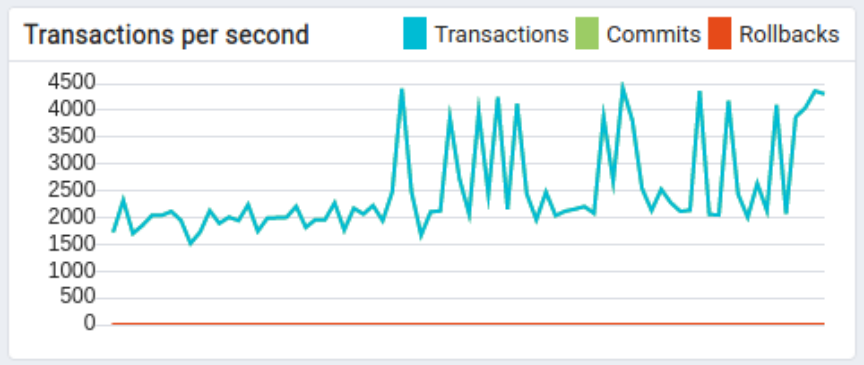
Šajā gadījumā tomēr tika pacelts pavedienu skaits ar minēto parametru un tika novērots, ka paceļot pavedienu skaitu no 20 līdz 40, izvade pēc testēšanas beigām (3.9. att.) apstiprina minējumu par visu pieejamo pavedienu nobloķēšanos - ar lielāku pavedienu skaitu var tikt izdarīti nevis 322'831 pieprasījumi, bet jau 528'843 pieprasījumi. Līdzīgi arī tagad ir izvadītas mazāk kļūdas par savienojuma noilgumu:

```
level=warning msg="Request Failed" error="Post
\"http://worker.catboi.net/devices/47d3fcda-d433-4520-9891-8cd200af0b3b/
point_data\": context deadline exceeded"
```

```
data_received..............: 250 MB 137 kB/s
data_sent..................: 192 MB 105 kB/s
http_req_blocked...........: avg=7.11ms    min=0s        med=4.22µs    max=15.65s    p(90)=6.31µs    p(95)=7.65µs
http_req_connecting........: avg=7.11ms    min=0s        med=0s        max=15.65s    p(90)=0s        p(95)=0s
http_req_duration..........: avg=282.77ms  min=0s        med=243.09ms  max=1m0s      p(90)=427.82ms  p(95)=466.47ms
http_req_receiving.........: avg=62.21µs   min=0s        med=52.92µs   max=24.78ms   p(90)=84.91µs   p(95)=100.2µs
http_req_sending...........: avg=35.84µs   min=0s        med=27µs      max=32.77ms   p(90)=44.16µs   p(95)=54.2µs
http_req_tls_handshaking...: avg=0s        min=0s        med=0s        max=0s        p(90)=0s        p(95)=0s
http_req_waiting...........: avg=282.67ms  min=0s        med=242.99ms  max=1m0s      p(90)=427.72ms  p(95)=466.36ms
http_reqs..................: 529843 290.5517/s
iteration_duration.........: avg=1.7s      min=505.81ms  med=1.23s     max=1m1s      p(90)=1.69s     p(95)=1.79s
iterations.................: 528843 290.003327/s
location_x.................: avg=23.973028 min=19.172845 med=24.061079 max=28.6439   p(90)=26.610504 p(95)=27.112112
location_y.................: avg=56.990219 min=52.332181 med=56.918228 max=61.417551 p(90)=59.508813 p(95)=60.036905
vus........................: 6      min=0     max=999
vus_max....................: 1000   min=1000  max=1000
```

3.9. att. Paraugs darbībai ar vairāk HTTP pavedieniem

Kā redzams, arī datu bāzu vadības sistēmā, tas atļauj nodrošināt vairāk aktīvo transakciju un lielāku darbību skaita izpildi testu laikā (3.10. att.), kas nozīmē, ka šajā gadījumā tiešām visas sistēmas darbību varēja bloķēt tieši HTTP pavedienu skaits.

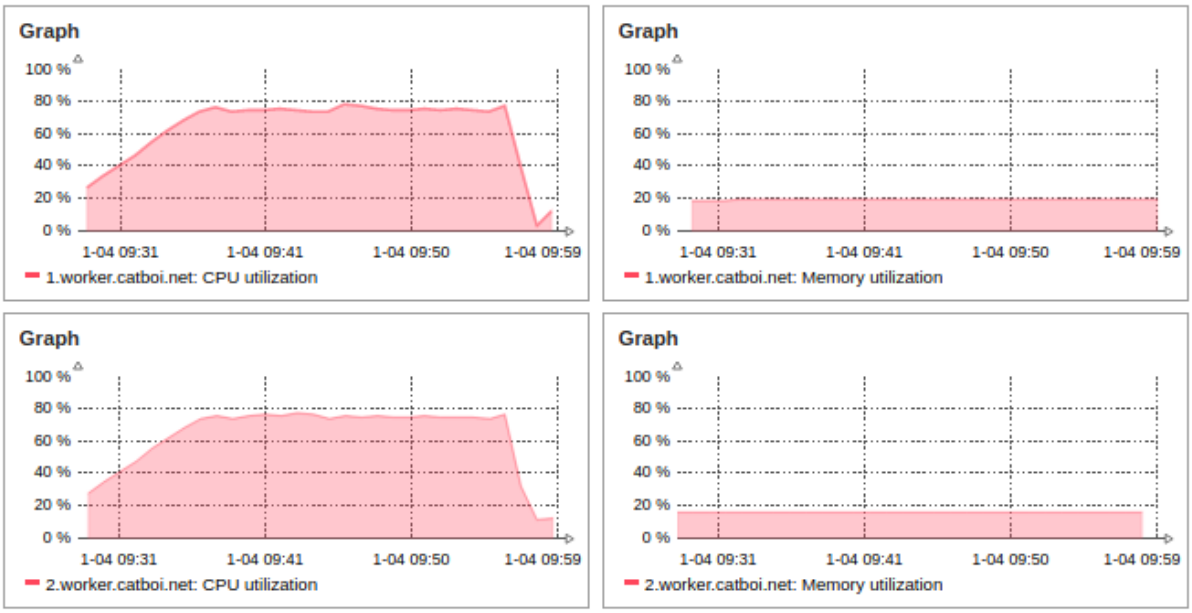


3.10. att. Transakciju izpilde, slodzes testa laikā

Tas liek atcerēties, ka praksē var būt nepieciešams veikt slodzes testēšanu un pārbaudīt programmatūras spēju darboties dažādās konfigurācijās, to izmainot atkarībā no testēšanas rezultātiem. Tāpat var noderēt arī metrikas izvade par šāda veida datiem, piemēram, Java gadījumā varētu izmantot JMX protokolu, lai gūtu ieskatu pavedienu stāvoklī, taču šīs sistēmas ietvaros tas pagaidām netika realizēts. Ja šeit tiktu tikai pacelts instanču skaits, tad pieejamie resursi netiktu izmantot lietderīgi, jo tiktu izveidoti lieki DB savienojumi, ņemot vērā ka tiem ir norādīts minimālais skaits. Piemēram, ja kā minimālais DB savienojumu

skaits ir norādīts 5, tad paceļot 4 lietotnes instances tiks izveidoti 20 savienojumi, nevis 10 savienojumi, kas būtu 2 instanču gadījumā, pat ja vairums no tiem būs dīkstāvē.

Šajā gadījumā ar lielāku pavedienu skaitu uz tiem pašiem serveriem tagad procesora noslodze arī ir ievērojami lielāka kā salīdzinājumā ar pirmo testu, kas liecina par to, ka tagad resursi tiek izmantoti efektīvāk, apstrādājot lielāku pieprasījumu skaitu (3.11. att.).



3.11. att. Paraugs procesora noslodzei pēc pavedienu skaita izmaiņām

Tātad papildinot tabulu 3.1., var iegūt kopskatu par konfigurācijas nozīmi (3.2. tabula).

3.2. tabula

Salīdzinājums testēšanai paceļot instanču skaitu un labojot lietotnes konfigurāciju

Scenārijs	**Izdarītie pieprasījumi**	**Pieprasījumi, kas netika apstrādāti**	**Izdevušies pieprasījumi, %**
Viena instance	322831	34080	89,4%
Divas instances	315729	16911	94,3%
Divas instances, labota konfigurācija	528843	7829	98,5%

Šeit gan jāpiezīmē, ka tīklošanas problēmu cēlonis var būt arī Docker Swarm pieeja savienojuma nodrošināšanai starp konteineriem, par ko pilnīgāku priekšstatu ļaus iegūt

slodzes testu veikšana, šeit iegūtos rezultātus salīdzinot ar Kubernetes orķestratora sniegtajiem rezultātiem.

3.5 Docker Swarm un Kubernetes veiktspējas salīdzinājums

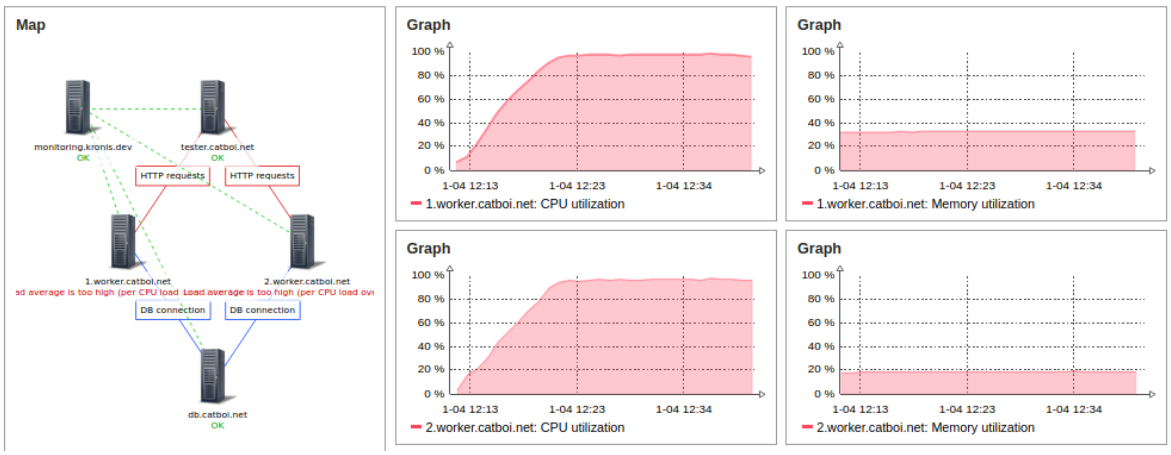
Realizētais monitorings un slodzes testi pavēra arī iespēju pārskatīt, kā orķestratori ir salīdzināmi savā starpā. Līdzīgi kā iepriekš, arī šajā testā tika paceltas 2 paralēlas lietotnes instances ar līdzīgu pavedienu konfigurāciju, izmantojot Kubernetes K3s distribūciju. Šiem testiem vajadzētu ilustrēt, vai arī Kubernetes gadījumā būs līdzīga situācija, kurā HTTP pavedienu skaits ievērojami ietekmē spēju pielāgoties noteiktai slodzei. Pēc testēšanas tika apkopota pieprasījumu apstrādes statistika (3.3. tabula).

3.3. tabula

Paraugs apstrādātajiem pieteikumiem dažādiem orķestratoriem

Orķestrators	**Pavedieni**	**Izdarītie pieprasījumi**	**Pieprasījumi, kas netika apstrādāti**	**Izdevušies pieprasījumi, %**
Kubernetes, K3s	20	304798	2315	99,2%
Kubernetes, K3s	40	443129	9867	97,7%

Šeit ir redzama pretējā situācija, kā ar Docker Swarm izmantošanas gadījumu, jo šeit vairāk kļūdu notiek izmantojot 20 pavedienus, nevis 40 pavedienus. To var skaidrot ar lielo procesora noslodzi otrajā testpiemērā, par ko arī monitorings izvadīja brīdinājumu (3.12. att).



3.12. att. Monitoringa izvade kas parāda lielo procesora noslodzi

Taču citādi ir novērojama līdzīga situācija iepriekšējam scenārijam - ar lielāku HTTP pavedienu skaitu ir iespējams apkopot vairāk pieprasījumu dotajā laikā. Lai gan pašiem konteineriem bija norādīts, ka tie nedrīkst izmantot vairāk par 90% procesora kodola resursu, šajā gadījumā kopējais resursu patēriņš tuvojas 100%, dēļ tā, ka arī orķestratora procesam uz servera ir nepieciešams darboties, kas Kubernetes gadījumā pret tiem ir prasīgāks, kā Docker Swarm orķestratoram.

Arī k6 rīka izvade palīdz skaidrot infrastruktūras noslodzi un testu rezultātus gadījumā ar 20 pavedieniem (3.14. att.) un gadījumā ar 40 pavedieniem (3.15. att.), kas tālāk ilustrē šo testpiemēru izpildes atšķirības.

```
data_received..............: 220 MB 121 kB/s
data_sent..................: 116 MB 64 kB/s
http_req_blocked...........: avg=22.73ms   min=0s        med=4.54µs    max=15.65s    p(90)=7.16µs    p(95)=8.69µs
http_req_connecting........: avg=22.72ms   min=0s        med=0s        max=15.65s    p(90)=0s        p(95)=0s
http_req_duration..........: avg=336.34ms  min=0s        med=214.12ms  max=1m0s      p(90)=305.53ms  p(95)=333.48ms
http_req_receiving.........: avg=68.08µs   min=0s        med=61.34µs   max=23.09ms   p(90)=101µs     p(95)=117µs
http_req_sending...........: avg=36.2µs    min=0s        med=30.76µs   max=29.41ms   p(90)=49.09µs   p(95)=58.27µs
http_req_tls_handshaking...: avg=0s        min=0s        med=0s        max=0s        p(90)=0s        p(95)=0s
http_req_waiting...........: avg=336.24ms  min=0s        med=214.02ms  max=1m0s      p(90)=305.42ms  p(95)=333.35ms
http_reqs..................: 304798 167.169243/s
iteration_duration.........: avg=2.97s     min=508.37ms  med=1.23s     max=1m1s      p(90)=1.69s     p(95)=30.62s
iterations.................: 303798 166.620784/s
location_x.................: avg=24.512114 min=19.713353 med=24.45702  max=28.571467 p(90)=26.333695 p(95)=26.92973
location_y.................: avg=56.875549 min=52.554769 med=56.836692 max=61.426234 p(90)=57.463545 p(95)=57.725129
vus........................: 4      min=0    max=999
vus_max....................: 1000   min=1000 max=1000
```

3.14. att. k6 rīka izvade testēšanai ar 20 HTTP pavedieniem

```
data_received..............: 332 MB 182 kB/s
data_sent..................: 175 MB 96 kB/s
http_req_blocked...........: avg=16.71ms   min=0s        med=4.34µs    max=15.65s    p(90)=6.6µs     p(95)=7.99µs
http_req_connecting........: avg=16.71ms   min=0s        med=0s        max=15.65s    p(90)=0s        p(95)=0s
http_req_duration..........: avg=427.36ms  min=0s        med=299.92ms  max=1m0s      p(90)=512.29ms  p(95)=590.09ms
http_req_receiving.........: avg=67.7µs    min=0s        med=59.49µs   max=19.25ms   p(90)=94.8µs    p(95)=111µs
http_req_sending...........: avg=35.21µs   min=0s        med=27.37µs   max=30.1ms    p(90)=45.05µs   p(95)=54.92µs
http_req_tls_handshaking...: avg=0s        min=0s        med=0s        max=0s        p(90)=0s        p(95)=0s
http_req_waiting...........: avg=427.26ms  min=0s        med=299.82ms  max=1m0s      p(90)=512.19ms  p(95)=589.99ms
http_reqs..................: 444129 243.36584/s
iteration_duration.........: avg=2.04s     min=510.57ms  med=1.3s      max=1m1s      p(90)=1.78s     p(95)=1.91s
iterations.................: 443129 242.817878/s
location_x.................: avg=24.475031 min=19.641751 med=24.41988  max=28.560947 p(90)=26.41922  p(95)=27.043024
location_y.................: avg=56.89859  min=52.554716 med=56.840974 max=61.420396 p(90)=57.523798 p(95)=57.699467
vus........................: 28     min=0     max=999
vus_max....................: 1000   min=1000  max=1000
```

3.15. att. k6 rīka izvade testēšanai ar 40 HTTP pavedieniem

Kā redzams, palielinot pavedienu skaitu, caurlaidspēja (angliski "throughput") pieauga no 167 apstrādātajiem pieprasījumiem sekundē līdz 243 pieprasījumiem sekundē, savukārt

vidējais testa iterācijas laiks samazinājās no 2,97 sekundēm līdz 2,64 sekundēm, par spīti pieaugumam kļūdu skaitā.

Saliekot rezultātus no iepriekšējās Docker Swarm testēšanas un Kubernetes testēšanas kopā, ir iespējams iegūt kopsavilkumu par orķestratoru veiktspēju (3.4. tabula) un spriest par šo rezultātu nozīmi.

3.4. tabula

Paraugs apstrādātajiem pieteikumiem dažādiem orķestratoriem

Orķestrators	**Pavedieni**	**Izdarītie pieprasījumi**	**Pieprasījumi, kas netika apstrādāti**	**Izdevušies pieprasījumi, %**
Docker Swarm	20	315729	16911	94,3%
Docker Swarm	40	528843	7829	98,5%
Kubernetes, K3s	20	304798	2315	99,2%
Kubernetes, K3s	40	443129	9867	97,7%

Rezultāti liecina, ka gan pašas lietotnes konfigurācijai, gan arī konteineru orķestratora izvēlei var būt liela ietekme uz infrastruktūras kopējo darbību. Testa un izstrādes vidēs vajadzētu varēt izmantot jebkuru no minētajām tehnoloģijām, taču, domājot par lietotņu izvietošanu produkcijas vidē, vajadzētu veikt slodzes testus, jo augstāk redzamās izdarīto pieprasījumu atšķirības ar lielāku paralēlo pieprasījumu skaitu varētu novest pie zemāka sistēmas drošuma - lielāka atteikto pieprasījumu skaita dēļ savienojuma noildzes iestāšanās.

Papildus tam, Kubernetes gadījumā, pat izmantojot K3s distribūciju, izveidojās situācija, kurā ar ierobežotiem skaitļošanas resursiem procesora noslodze bija ļoti liela, sasniedzot pat 100%, kas nenotika Docker Swarm izmantošanas gadījumā. Tas nozīmē, ka Kubernetes gadījumā ir jārēķinās ar lielāku resursu virstēriņu (angliski "overhead"), orķestratora tehniskās realizācijas dēļ.

Produkcijā, protams, neatkarīgi no izvēlētās orķestrācijas tehnoloģijas, klastera vadītājiem vajadzētu būt izvietotiem atsevišķi no serveriem, kuros tiks palaisti pārējie konteineri, lai neradītu situācijas, kurās sistēma zaudē responsivitāti uz mēģinājumiem tai pieslēgties administratīvu darbību veikšanai.

SECINĀJUMI

Maģistra darba ietvaros tika veiksmīgi realizēti izvirzītie mērķi:

- tika izveidots rīks konteineru ieviešanas atvieglošanai, automatizējot infrastruktūrai nepieciešamo konfigurācijas darbību veikšanu
- tika izveidota programmatūra, kura pēta iespēju COVID-19 kontaktus izsekot ar GPS datu palīdzību, kas kalpo gan par paraugu konteinerizētas lietotnes izstrādes praksēm, gan arī par paraugu horizontāli lietotņu mērogošanai, lai uzlabotu to darbību
- tika izveidoti arī slodzes testi ar k6 rīku, kuri ļāva salīdzināt abu apskatīto konteineru orķestratoru, Docker Swarm un Kubernetes, darbību un resursu patēriņu

Pēc šī darba izstrādes ir iespējams izdarīt vairākus secinājumus attiecībā gan uz izvēlēto pieeju infrastruktūras konfigurācijas automatizācijai, gan attiecībā uz lietotņu konteinerizāciju un orķestratoriem, kurus iespējams izvēlēties vižu uzturēšanai.

Runājot par infrastruktūras automatizāciju, tā viennozīmīgi var palīdzēt minimizēt cilvēciskā faktora radītos riskus attiecībā uz konfigurācijas mainīšanu, paku instalēšanu vai vides vispārīgu sagatavošanu darbam. Tāpat tas atļauj arī ietaupīt laiku, jo darba izstrādes laikā visi serveri tika pārinstalēti no jauna vairāk kā 10 reizes. Šī automatizācija arī atļauj būt drošākiem par to, ka, ja nu ar galveno serveri kaut kas notiks vai ja būs nepieciešams izveidot jaunas vides, tad to būs iespējams salīdzinoši viegli izdarīt, jo nebūs manuāli jāseko instrukcijām, kas laika gaitā var būt kļuvušas nepilnīgas vai pat neapraksta pilno konfigurācijas formātu. Infrastruktūras satura formalizēšana kodā (angliski "infrastructure as code" jeb IaC) viennozīmīgi var palīdzēt šo problēmu mazināšanai.

Tāpat SSH protokola izmantošana administratīvo darbību veikšanai var padarīt šos rīkus saderīgus ar lielu daudzumu dažādu operētājsistēmu distribūciju un to konfigurāciju, jo nebūs nepieciešams instalēt specifiskas pakas, kā tas jādara, piemēram, Salt rīka izmantošanas gadījumā. Tajā pa'ša laikā, komandu izsaukšana caur čaulu var sarežģīt pašu uzdevumu realizāciju, jo jārēķinās ar to, ka noteikti rīki var nebūt pieejami uz attālinātā servera. Kopumā apskatīto rīku bija salīdzinoši viegli izstrādāt, taču problēmas radīja datu padošanas formāts - argumentus nācās padot JSON formātā, jo Bash čaula nenodrošina darbu ar objektiem, līdzīgi PowerShell. Papildus tam, problēmas radīja arī izvades kanālu

ierobežotais skaits, t.i. STDOUT un STDERR, kas padarīja grūtāku izvades nošķiršanu sistēmas lietotājam un citiem procesiem, kuriem vajadzētu ievadīt noteiktus objektus, vienas komandas izvadi padodot citai kā ievadi.

Runājot par lietotņu konteinerizāciju, tā var palīdzēt ar lietotņu izvietošanu uz jaunām vidēm, jo konteineros būs ietverts viss kods, kas nepieciešams to palaišanai. Līdzīgi uz viena un tā paša servera var palaist vairākas instances, kurām pēc vajadzības arī var pielāgot konfigurāciju. Šeit noderīga izrādījās modulārā monolītiskā arhitektūra, kas var kalpot par vieglāk pārvaldāmu alternatīvu mikro servisu arhitektūrai, ja nepieciešams horizontāli mērogot lietotnes, lai varētu izturēt lielāku slodzi. Tajā pašā laikā tā pieprasa sekošanu noteiktām praksēm attiecībā uz konfigurācijas nodošanu, audita izvadi un citiem lietotnes aspektiem, taču kad tas ir izdarīts, tad pat dažādu tehnoloģiju lietotnes var pārvaldīt līdzīgā veidā ar vieniem un tiem pašiem mehānismiem - šeit tas tika nodemonstrēts, lietotnei esot sarakstītai Ruby ar Ruby on Rails, savukārt slodzes testiem esot sarakstītiem K6 rīkā (kas ietver Golang un iegultu JavaScript izpildes vidi), abas no tām pārvaldot vienādi.

Jāpiemin, ka noteiktus riskus var radīt datu apstrāde pašā datu bāzu vadības sistēmā, dēļ tā, ka procesora noslogošana vienas DBVS instances gadījumā var novest pie visus saistīto lietotņu lēndarbības, kas to padara par visas sistēmas problēmpunktu (angliski "single point of failure"). Tomēr, ja darbs norit pie relāciju datu apstrādes vai arī ģeospatiālo datu apstrādes, kas efektīvāk notiks pašā DBVS dažādu optimizāciju dēļ, var nākties apsvērt iespēju vertikāli mērogot tās instanci, tai piešķirot ievērojami vairāk resursus kā individuālajiem serveriem, uz kuriem tiks palaistas pārējās lietotnes instances. Alternatīvi var apsvērt iespēju izveidot DBVS klasteri, taču tad ir jārēķinās ar distributēto datu korektuma nodrošināšanas jautājumu, vai arī var pētīt iespēju ierobežot noteiktiem izsaukumiem pieejamos resursus, tos padarot ilgstošākus, taču liedzot tiem iespēju tik daudz ietekmēt kopējo sistēmas darbību, ja DBVS šādus mehānismus nodrošina.

Attiecībā uz izstrādāto lietotni, interpretētās valodas kopumā var izmantot šādu sistēmu izstrādei, ja ir pieejami mehānismi un līdzekļi to mērogošanai, taču par to ir jādomā jau izstrādes laikā, lai implementētie mehānismi būtu saderīgi ar paralelizāciju. Arī GPS datu izmantošana nerada pārāk lielu slodzi, taču ir jārēķinās ar to, ka centralizēta atrašanās vietu datu uzglabāšana var radīt problēmas attiecībā uz Vispārīgo datu aizsardzības regulu [40], ja

šāda veida sistēma tiktu implementēta un lietota Eiropas Savienībā. Līdzīgi, pat ja šāda arhitektūra ļautu agregēt datus un, nepublicējot individuālo personu pārvietošanās informāciju, atļautu parādīt gadījumu izplatību valsts teritorijā pat tiem, kam šāda veida lietotne nav instalēta, nav iespējams ignorēt tās radītos drošības riskus, ja notiktu ielaušanās attiecīgajā DBVS instancē un DB dati tiktu lejupielādēti vai izplatīti. Tāpat, šādas sistēmas realizācija prasītu lielu atsaucību un uzticību no sabiedrības locekļiem tās instalācijai, kā dēļ Apturi Covid arhitektūru kopumā var uzskatīt par piemērotāku tekošajai situācijai, ņemot vērā minētos ierobežojumus.

Runājot par konteineru orķestratoriem, ar izstrādāto rīku bija iespējams sagatavot klasterus gan priekš Docker Swarm, gan priekš Kubernetes K3s distribūcijas lietošanas, kā arī testēšana parādīja, ka abos gadījumos orķestratori ir spējīgi darboties ar līdzīgu efektivitāti. Tomēr, izskatās, ka Docker Swarm gadījumā tīklošanas implementācija ir nedaudz sliktāka par Kubernetes - pat izmainot lietotnes konfigurāciju, tāpat tika novērots lielāks kļūdaini apstrādāto pieprasījumu īpatsvars, lai gan Kubernetes pielietošanas gadījumā bija novērota lielāka procesora resursu patērēšana, kas zem lielas slodzes var sistēmu padarīt neatsaucību pret lietotāja ievadi.

Tomēr tika parādīts, ka abi no orķestratoriem var tikt izmantoti izstrādes un testa vižu uzturēšanai, pat situācijās ar ierobežotiem skaitļošanas resursiem, kas ir pieejami individuālajiem serveriem. Tas norāda uz iespēju realizēt decentralizētu infrastruktūru, kas atļautu katram projektam uzturēt konteineru orķestratora instanci ar pieņēmumu, ka to konfigurācija tiktu nodrošināta automatizētā veidā un tāpēc visiem orķestratoriem būtu pēc iespējas līdzīgāka, tā atvieglojot to pārvaldi. Savukārt attiecībā uz produkcijas vižu uzturēšanu, lietotnēm, kurām ir daudz lietotāju un ir raksturīga lielāka slodze, viennozīmīgi vajadzētu apsvērt lielāku skaitļošanas resursu piešķiršanu, tajā pat laikā domājot par drošumu - noturību pret attiecīgā orķestratora kļūmēm, kā arī individuālu serveru nepieejamību noteiktos laika posmos, tāpēc vajadzētu padomāt par vairāku orķestratora instanču pievienošanu klasterim.

PRIEKŠLIKUMI

Darba saturā tika parādīts, ka konteineru un konteineru orķestratoru tehnoloģijas atvieglo infrastruktūras pārvaldi - šīs tehnoloģijas var droši pielietot projektu izstrādē, taču to darot, ir jāizvērtē pieejamie rīki un to ieviešanas metodes. Pastāv iespēja, ka papildus skaitļošanas resursu ieguve var būt ekonomiski izdevīgāka, kā pašiem savu orķestratoru uzturēšana, ņemot vērā personāla izmaksas. Līdzīgi pakalpojumu līmeņa vienošanās (angliski "service level agreement") var nodrošināt papildus trešās puses resursu iesaistīšanu problēmsituāciju gadījumā. To vajadzētu izvērtēt, lai izvēlētos piemērotāko opciju.

Papildus tam, tika parādīts, ka pašiem savu rīku izstrāde infrastruktūras konfigurācijas pārvaldes automatizācijai nebūt nav pārāk sarežģīta, pieņemot, ka tiek izmantoti standarta protokoli to pārvaldei (SSH), tāpēc vajadzības gadījumā šādu rīku izstrāde varētu būt noderīga. Ja nepieciešams pārvaldīt infrastruktūru, var izmantot arī kādu no industrijā plaši pazīstamajiem rīkiem, piemēram Ansible vai Salt, ja paši pārvaldāmie serveri tos atbalsta. Šāds laika ieguldījums var sekmēt spēju vajadzības gadījumā ātri izveidot jaunas vides, vai aizstāt zaudētās, kā arī var palīdzēt horizontālās mērogošanas realizācijā.

Tika novērots arī tas, ka pašas lietotnes konfigurācija var ievērojami ietekmēt tās ātrdarbību un resursu patēriņu. Lai nodrošinātu to, ka infrastruktūrā pieejamie resursi tiek izmantoti optimāli, vajadzētu izstrādātajām lietotnēm, papildus vienībtestiem, integrācijas testiem un citiem testiem, padomāt arī par slodzes testu realizāciju. Tāpat šeit var palīdzēt arī informācija par to, kas rada aizkaves - vai nu no ārējiem testēšanas rīkiem, vai arī no metrikas, kas tiktu savāktas pašā lietotnē, piemēram izmantojot JMX protokolu un kādu no rīkiem.

Tāpat, ieteicams realizēt arī infrastruktūras monitoringu un apziņošanas sistēmu - darbā tika minēts piemērs ar Zabbix, kurš paziņoja par lielu procesora resursu patēriņu, kas negatīvi var ietekmēt visas lietotnes, kas atrodas uz attiecīgā servera. Ja šādam monitoringam pieslēdz apziņošanas sistēmu, tad par šāda veida kļūmēm paziņojumi var tikt izsūtīti automatizētā veidā, tā mazinot laiku, līdz reakcijas sākumam uz šo situāciju.

IZMANTOTĀ LITERATŪRA

1. / Internets. https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html#managed-node-requirements, skatīts 28.12.2020

2. Understanding variable precedence - Ansible Documentation / Internets. https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#understanding-variable-precedence, skatīts 28.12.2020

3: , , 2020

4: , , 2020

5. / Internets. https://docs.saltstack.com/en/getstarted/system/plugins.html, skatīts 28.12.2020

6. / Internets. https://docs.alpinelinux.org/user-handbook/0.1a/Working/openrc.html, skatīts 28.12.2020

7. / Internets. https://typer.tiangolo.com/, skatīts 28.12.2020

8. / Internets. https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7.1, skatīts 28.12.2020

9. / Internets. https://www.json.org/json-en.html, skatīts 28.12.2020

10. / Internets. https://jsonpickle.github.io/, skatīts 28.12.2020

11. / Internets. https://yaml.org/, skatīts 28.12.2020

12. / Internets. https://json5.org/, skatīts 28.12.2020

13. / Internets. https://docs.python.org/3/library/functions.html#exec, skatīts 28.12.2020

14. / Internets. https://docs.python.org/3/library/typing.html, skatīts 28.12.2020

15. / Internets. https://docs.ansible.com/ansible/latest/user_guide/become.html, skatīts 28.12.2020

16. / Internets. https://www.fail2ban.org/wiki/index.php/Main_Page, skatīts 28.12.2020

17. / Internets. https://www.portainer.io/, skatīts 28.12.2020

18. / Internets. https://caddyserver.com/, skatīts 28.12.2020

19. / Internets. https://certbot.eff.org/, skatīts 28.12.2020

20. / Internets. https://traefik.io/traefik/, skatīts 28.12.2020

21. / Internets. https://rancher.com/docs/rke/latest/en/os/, skatīts 28.12.2020

22. / Internets. https://k3s.io/, skatīts 28.12.2020
23. / Internets. https://docs.gitlab.com/ee/user/packages/container_registry/, skatīts 28.12.2020
24. / Internets. https://k8slens.dev/, skatīts 28.12.2020
25. / Internets. https://www.gnu.org/licenses/lgpl-3.0.html, skatīts 28.12.2020
26. / Internets. https://docs.docker.com/storage/volumes/, skatīts 28.12.2020
27. / Internets. https://www.graylog.org/, skatīts 28.12.2020
28. / Internets. https://www.elastic.co/elastic-stack, skatīts 28.12.2020
29. / Internets. https://docs.docker.com/config/containers/logging/syslog/, skatīts 28.12.2020
30. / Internets. https://www.vaultproject.io/, skatīts 28.12.2020
31. / Internets. https://apturicovid.lv/, skatīts 28.12.2020
32. / Internets. https://covid19.apple.com/contacttracing, skatīts 28.12.2020
33. / Internets. https://www.csb.gov.lv/en/statistics/statistics-by-theme/population/number-and-change/key-indicator/population-number-its-changes-and-density, skatīts 28.12.2020
34. / Internets. https://twitter.com/SPKCentrs/status/1313806336413642755, skatīts 28.12.2020
35. / Internets. https://spkc.maps.arcgis.com/apps/opsdashboard/index.html#/4469c1fb01ed43cea6f20743ee7d5939, skatīts 28.12.2020
36. / Internets. https://www.autentica.lv/lv/article/apturi-covid/, skatīts 28.12.2020
37. / Internets. https://tools.ietf.org/html/rfc4122, skatīts 28.12.2020
38. / Internets. https://kompose.io/, skatīts 28.12.2020

PIELIKUMI

README.md - Astolfo Cloud Servant rīka dokumentācija

ASTOLFO CLOUD SERVANT

A solution to automate the creation and testing of a variety of containerization solutions as well as run other tasks.

One of the problems that I have oftentimes stumbled upon is the initial setup of the infrastructure for running containerized workloads. Relying on managed control planes and SaaS/PaaS solutions can be both expensive and risky, since such services could be retired at any time, could create security vulnerabilities (as opposed to everything being 100% on-prem and not exposed to the outside world) and also have breaking changes with no control over when and how migrations are handled.

However, setting up Kubernetes and Docker Swarm, or even Hashicorp Nomad clusters can take time and even when following the official instructions, it can be a bit hard to walk through this process. That's why I wrote Astolfo, which is my attempt at providing the convenience of a CentOS graphical installer, but for setting up infrastructure. Of course, if you don't need the UI bits, just use the CLI directly.

Internally, it uses Paramiko to execute commands over SSH. This approach was chosen as opposed to just using Ansible or Salt, because oftentimes Ansible won't work well with imperative commands. This small set of scripts is also easily *hackable*, in case you want to implement new functionality.

If you'd like to see an example of how to use the application, feel free to read the Quick Start guide in GUIDE.md.

Requirements

There are some requirements for both the client and the servers to be able to run this solution.

For the client:

- Docker installed for lauching the tool
- or an installation of Python 3+ and venv (for development or for launching without Docker)
- a web browser capable of executing JavaScript (when the actual Web UI is implemented, instead of just the CLI)

For the servers:

- any RPM distribution supported by Docker or the other tools (CentOS/Fedora/RHEL/Oracle Linux); more to be supported in the future
- an installation of an SSH server, that supports connecting by using key authentication; passwords might be supported in the future
- a key that's stored on the remote server, that doesn't have a passphrase; passphrases might be supported in the future
- the user that you connect to should be root for now; permission control will change in the future, e.g. something like Ansible's "become" will be implemented

Development

For local development, you'll want a virtualenv for Python.

See how to create it here: venv — Creation of virtual environments

Then you'll need to activate it, for example, in GNU/Linux:

```
source venv/bin/activate
```

After that you should be able to install dependencies:

```
pip install -r requirements.txt
```

There's also a Visual Studio Code workspace in the root folder, for convenience.

If you want to build it with Docker, just run:

```
docker build -t astolfo .
```

Architecture

The app is based on a Typer CLI, with some additional magic sprinkled in, which allows for dynamically loading new commands at runtime. This allows for having a variable amount of tasks that are capable of registering themselves into the system. This also means, that, in theory, the tasks could have separate arguments and it would be pretty easy to implement new ones.

The CLI itself provides most of the administrative actions that you might want to do, therefore adding new hosts and tasks should be achievable entirely through it.

The output is also structured in a particular manner:

- all of the output that people might be interested in reading goes to STDERR
- all of the output that's machine readable goes to STDOUT

If you feel like the script is too noisy, you can just pipe STDOUT to /dev/null, for example:

```
python astolfo.py task run ping 1.worker.catboi.net 2>/dev/null
```

Usage example

Running in Docker

The run syntax for Docker is made to be as similar to invoking the app directly through the terminal, as possible.

For example, you can run the following:

```
docker run --rm -it astolfo info
```

This will run the command in interactive mode with TTY, to display the colors correctly for more user friendly output. It will also remove the container after it exits.

You'll probably also want to use persistent storage in the case of docker containers, for example:

```
docker run --rm -it -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo info
```

Only real quirk of Docker in this context is that passing things from STDIN and piping them from the host system will need to have the --tty option excluded, for example:

```
cat ~/.ssh/mysite/app/id_rsa | docker run --rm -i -v
astolfo_tasks:/app/tasks -v astolfo_infrastructure:/app/infrastructure -v
astolfo_files:/app/files astolfo host key 1.worker.catboi.net
```

Example of cleaning up the volumes afterwards:

```
docker volume rm astolfo_infrastructure astolfo_tasks astolfo_files
```

Also actual container name will probably need to be the full reference, not just `astolfo`, once the software is publically available.

Working with files

You can store files in the app directory, which can then be used for copying them to the remote system, deploying them in Swarm/Kubernetes etc.

Listing the files with their contents

You can list all of the files that are currently stored in the `files` directory, as well as get their contents with:

```
python astolfo.py file list
```

This will return JSON output with all of the information about them.

Adding a new file

Adding a new file is done through providing the information about it:

```
cat ~/my-files/test.yml | python astolfo.py file add test.yml
```

This will save the file under the `files` directory with the passed in name. You can also pipe other things to STDIN, even downloads that you trust.

Deleting a file

You can also remove certain files from the directory with the following command:

```
python astolfo.py file delete test.yml
```

Working with hosts

Hosts are the nodes that we will connect to through SSH. These need some information about the actual node, as well as an SSH key to authenticate with.

Listing the current hosts

You can list all of the hosts that the app currently knows about by running the following:

```
python astolfo.py host list
```

This will return JSON output with all of the information about them.

Adding a new host

Adding a new host is done through providing the information about it:

```
python astolfo.py host add 1.worker.catboi.net "Application server that
will run front-end and back-end containers" "CentOS 8" root id_rsa
```

This will save a new folder and `server.json` file in the `hosts` folder. Run the command without any arguments to see information about their format.

Adding an SSH key to host

To be able to connect to hosts through SSH, we need their keys. These can be added to the app through the following command:

```
cat ~/.ssh/mysite/app/id_rsa | python astolfo.py host key
1.worker.catboi.net
```

In this example, this will copy the key into `infrastructure/1.worker.catboi.net` folder, you can also do that manually if you'd like.

Deleting a host

If you'd like to remove a certain host from the inventory, you can run the following command:

```
python astolfo.py host delete 1.worker.catboi.net
```

Working with tasks

Tasks are the Python scripts that will allow us to execute certain functionality against the remote nodes and to react to the output we'll receive over SSH.

You can read the full documentation of the built in tasks in TASKS.md.

If you'd like more concrete examples of how to use the functionality, see some of the scripts under the script_examples folder.

Writing new tasks

The tasks themselves are just commands to be run over SSH. However, in contrast to other solutions that need to do all of the processing on the node, we can retrieve the output from the remote commands and figure out what to do locally, before sending the next command. While slower, this is far easier to work with, since we don't need to worry about sending Python scripts to the node and hope Python will work there.

See some of the example tasks in the `tasks` folder if you'd like to write new ones. There are a variety of utilities provided to connect to hosts and execute commands over SSH more easily. Apart from that, however, most of the processing simply involves reacting to the standard output returned by the remote machines.

Once a task has been finished, you can add it to the tasks folder, either with the management commands above, or perhaps just add the task directly to the folder.

Scripting

This script can also be used as a part other, more complex scripts.

For a somewhat basic example that automates setting up a Docker Swarm cluster on a few nodes, see `full-cluster-example.sh`.

TODO

- The actual icon is borrowed from the web, if anyone knows the artist, feel free to let me know so i can ask them for permission.
- Add Kompose support, for running it locally on a file

GUIDE.md - Astolfo Cloud Servant lietošanas pamācība

Quick Start Guide

This guide will demonstrate how to manage a CentOS 8 server with the Astolfo Cloud Servant tool.

Building the tool

First, let's build the tool locally:

```
docker build -t astolfo .
```

(container images might be published in the future)

Adding a server to manage

Then, let's add a host to manage, assuming that we have a server running somewhere in the cloud with an SSH key based authentication. If you don't, look up how to generate and save a SSH key, for example: How To Set Up SSH Keys.

Let's get information about how to add a host, with:

```
docker run astolfo host add --help
```

The output of which will be approximately the following:

```
Usage: astolfo.py host add [OPTIONS] HOST [DESCRIPTION] OS [USERNAME]
                           [KEYFILE]

  Will add a new host to the infrastructure folder.  This is much like the
  description of the hosts in Ansible, but for performance reasons some
  information should be known beforehand (such as the OS).

Arguments:
  HOST           Name of the host to add, will name the folder after it.
                 Example: db.myapp.com  [required]

  [DESCRIPTION]  The description of the host, to display to user. Example:
                 Database server for running PostgreSQL

  OS             Vague description of the operating system that the host is
                 running. Logic can be written for hosts based on it (for
                 example, using yum on CentOS, but apt on Debian). Example:
                 CentOS 8  [required]

  [USERNAME]     The username to connect to on the remote machine through
SSH.
                 The user needs to be able to execute commands with sudo
with
                 no password if not root.  [default: root]

  [KEYFILE]      The name of the SSH key file in the host folder. Example:
                 id_rsa  [default: id_rsa]


Options:
  --help  Show this message and exit.
```

So, let's do just that and add a new server, but this time creating a volume so that the files the Docker container generates are persisted:

```
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo host add 1.worker.catboi.net "Application server that will run
front-end and back-end containers" "CentOS 8" root id_rsa
```

This command means that we'll create a server with the hostname of 1.worker.catboi.net, a description and some basic information about the OS which it is running, and that we'll atttempt connecting to it with the root user, using id_rsa key.

It should output something like the following:

```
Creating new host: 1.worker.catboi.net
Writing to file: infrastructure/1.worker.catboi.net/server.json
Wrote 224 characters to file...
Successfully added host!
{
  "py/object": "cli.host_data.Host",
  "host": "1.worker.catboi.net",
  "description": "Application server that will run front-end and back-end
containers",
  "os": "CentOS 8",
  "username": "root",
  "keyfile": "id_rsa"
}
```

We can check whether the host has been imported with the following command:

```
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo host list
```

Which will return the list of all of the created hosts, including ours.

Now, we need to import the actual SSH key to use, which we can achieve by piping it to the STDIN channel of the Docker container:

```
cat ~/.ssh/mysite/app/id_rsa | docker run --rm -i -v
astolfo_tasks:/app/tasks -v astolfo_infrastructure:/app/infrastructure -v
astolfo_files:/app/files astolfo host key 1.worker.catboi.net
```

Here we assume that the key is in the user's home directory, under the ".ssh/mysite/app" path, though you might use any other command to retrieve its contents, instead of just cat.

The expected output of the command above is approximately as follows:

```
Adding key for host: 1.worker.catboi.net
Key will be added in file: infrastructure/1.worker.catboi.net/id_rsa
Wrote 3243 characters to file...
Successfully added host key!
```

That's it! Now we can access the server and run commands against it!

Running commands against the server

Now we can have a look at the tasks that are available to run.

We can do so with the following command:

```
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run --help
```

Which will list all of the tasks that are available with a bit of info about each. If you'd prefer JSON output, there is also the "task list" command.

You can also get the information about individual commands, by using the --help option:

```
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run ping --help
```

which will output the following:

```
Usage: astolfo.py task run ping [OPTIONS] HOST

  This task will ping the selected node and check whether it can be
  connected to through SSH. It will also check whether sudo works without
  password, if the account is not root.

Arguments:
  HOST  The node to connect to.  [required]

Options:
  --help  Show this message and exit.
```

So, let's ping the server!

```
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run ping 1.worker.catboi.net
```

This will cause the tool to use SSH to connect to the server and run the commands in the file against it. You'll see a lot of output from the command:

```
Connecting to host: 1.worker.catboi.net
Executing: whoami
Stages: {
  "Connect": {
    "py/object": "tasks.utils.stages.Stage",
    "name": "Connect",
    "description": "Will connect to the host through SSH",
    "order": 0,
    "successful": true,
    "command": "whoami",
    "output": [
      "root"
    ],
    "finished": true
  },
  "HasPermissions": {
    "py/object": "tasks.utils.stages.Stage",
    "name": "HasPermissions",
    "description": "Will check whether user is root or can execute sudo
without password",
    "order": 1,
    "successful": true,
    "command": "",
    "output": "",
    "finished": true
  }
}
{
  "py/object": "tasks.task_ping.PingResponse",
  "host_pinged": "1.worker.catboi.net",
  "host_user": "root",
  "connection_successful": true,
  "user_can_sudo": false
}
```

Each executed command in the remote shell will be listed (in this case "whoami"), there will be output of the stages that have been completed and which output and commands each of them was associated with, as well as there will be output objects in JSON which summarize the results, in this case, it's the "tasks.task_ping.PingResponse" object. These are output to STDOUT and can be passed to other commands, whereas STDERR output is meant for the user.

So, let's run something else and update our server's software!

```
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run update 1.worker.catboi.net
```

This might return something like the following:

```
Installing updates on host: 1.worker.catboi.net
Connecting to host: 1.worker.catboi.net
Executing: yum -d1 update -y
Stages: {
  "InstallUpdates": {
    "py/object": "tasks.utils.stages.Stage",
    "name": "InstallUpdates",
    "description": "Will install the updates",
    "order": 0,
    "successful": true,
    "command": "yum -d1 update -y",
    "output": [
      "Last metadata expiration check: 2:49:54 ago on Sat 02 Jan 2021
11:17:28 AM EET.",
      "Dependencies resolved.",
      "Nothing to do.",
      "Complete!"
    ],
    "finished": true
  }
}
{
  "py/object": "tasks.task_update.UpdateInstallResponse",
  "updates_installed": true
}
```

Internally, this will use the "yum" command to update the packages, while other Linux distributions, such as Debian might be suppoted in the future.

Passing data between the commands

Some commands will need input into them. For example, let's say that we can set up the Zabbix agent on our server, so that we can monitor it with the Zabbix server, which will aggregate data about it.

In addition to installing the services and enabling, we also need to set which server we'll allow access to. Therefore, some of the tool's commands will require a JSON argument with the necessary data.

For example, if we run the tool without any such arguments, we'll get error output:

```
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run zabbix_agent 1.worker.catboi.net
```

We'll be told that we're missing the necessary argument:

```
Usage: astolfo.py task run zabbix_agent [OPTIONS] HOST INPUT
Try 'astolfo.py task run zabbix_agent --help' for help.

Error: Missing argument 'INPUT'.
```

So how do we figure out the JSON we need to input? Quite simply, we use the "task arguments" instead of "task run", which will output the necessary arguments for the command, if any are needed:

```
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task arguments zabbix_agent
```

Which will produce:

```
{
  "py/object": "tasks.task_zabbix_agent.ZabbixAgentRequest",
  "server_address": ""
}
```

So, we know that we need to fill out the server address! Most of the examples of commands which require parameters under TASKS.md will also include examples of what valid values look like.

When attempting to input the JSON into the command, we don't even have to use multiple lines, the following input is completely valid:

```
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run zabbix_agent 1.worker.catboi.net '{"py/object":
"tasks.task_zabbix_agent.ZabbixAgentRequest", "server_address":
"ram.servers.kronis.eu"}'
```

If you need to run the same command against multiple servers, you can even store it in a Bash variable, like so:

```
ZABBIX_SERVER='{"py/object": "tasks.task_zabbix_agent.ZabbixAgentRequest",
"server_address": "ram.servers.kronis.eu"}'
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run zabbix_agent 1.worker.catboi.net "$ZABBIX_SERVER"
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run zabbix_agent 2.worker.catboi.net "$ZABBIX_SERVER"
```

The command above will result in a whole lot of output, but should finally output the following summary:

```
{
  "py/object": "tasks.task_zabbix_agent.ZabbixAgentResponse",
  "server_address": "ram.servers.kronis.eu",
  "repo_added": true,
  "package_installed": true,
  "config_backup": true,
  "server_update": true,
  "systemctl_enable": true,
  "service_restart": true
}
```

Further reading

In summary, you have the basic usage of the tool down. Some commands also take the output of other commands as input, for example, when creating a Docker Swarm cluster, you might want to pass the join tokens after initializing a cluster to other servers os that they can join it.

In practice, it would look a bit like the example above:

```
JOIN_TOKENS=$(docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run docker_swarm_cluster 1.worker.catboi.net)
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run docker_swarm_worker 2.worker.catboi.net "$JOIN_TOKENS"
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run docker_swarm_worker db.catboi.net "$JOIN_TOKENS"
docker run -v astolfo_tasks:/app/tasks -v
astolfo_infrastructure:/app/infrastructure -v astolfo_files:/app/files
astolfo task run docker_swarm_worker tester.catboi.net "$JOIN_TOKENS"
```

TASKS.md - Astolfo Cloud Servant paraugi uzdevumu izsaukšanai

Tasks

Tasks are the Python scripts that will allow us to execute certain functionality against the remote nodes and to react to the output we'll receive over SSH.

The rest of the documentation for the tool can be found in the README.md!

Working with tasks

Tasks are stored in the `tasks` directory, to which they can be added or removed from.

During the runtime of the application, they will be loaded dynamically, with any errors resuting in output.

Listing the current tasks

To see which tasks are currently available, you can run the following:

```
python astolfo.py task run
```

which will return the list of tasks and their descriptions in the regular CLI format.

Alternatively, you can also run:

```
python astolfo.py task list
```

This will return the task names and descriptions in JSON.

Adding a new task

You can add a new task to the app by piping the necessary .py file through STDIN.

```
cat ~/my_new_task.py | python astolfo.py task add my_new_task
```

You could even get them from the internet with curl/wget, but in that case you'd probably want to validate the checksum of the contents and trust the source.

If a task does not appear to be valid, it will be automatically removed at the end of the import, when using the CLI.

Alternatively, you can just copy the file to the `tasks` directory, but you need to have the filename start with `task_`, for example a `ping` task would have the filename `task_ping.py`.

Deleting a task

If you'd like to remove a task from the app, you can just run the following command:

```
python astolfo.py task delete my_new_task
```

This will delete the task's Python file from the `tasks` directory.

Listing stages

Each task has one or multiple stages that do everything it needs. Since each stage can have different output, this should allow for more easily tracking their execution.

Get the list of stages for a particular task with:

```
python astolfo.py task stages ping
```

Listing arguments

Many tasks expect an argument that contains JSON with the parameters that they require. Since this JSON also contains information for Python serizalization, you might need to check what they expect first.

You can do this with:

```
python astolfo.py task arguments zabbix_agent
```

A filled out instance of said argument JSON can later be passed in to it, like:

```
python astolfo.py task run zabbix_agent '{"py/object":
"tasks.task_zabbix_agent.ZabbixAgentRequest", "server_address":
"zabbix.catboi.net"}'
```

Examples of running tasks for generic shell functionality

Running arbitrary shell commands

```
python astolfo.py task run shell_execute 1.worker.catboi.net '{"py/object":
"tasks.task_shell_execute.ShellExecuteRequest", "shell_command": "date"}'
```

Pinging the host

```
python astolfo.py task run ping 1.worker.catboi.net
```

Get info about operating system

```
python astolfo.py task run os_version 1.worker.catboi.net
```

Get disk usage

```
python astolfo.py task run disk_usage 1.worker.catboi.net
```

List installed packages

```
python astolfo.py task run installed_packages 1.worker.catboi.net
```

Install updates

```
python astolfo.py task run update 1.worker.catboi.net
```

Examples of running tasks for manipulating the files on the system

Creating a directory

```
python astolfo.py task run directory_create 1.worker.catboi.net
'{"py/object": "tasks.task_directory_create.DirectoryCreateRequest",
"directory_path": "/docker/some/test/folder"}'
```

Deleting a directory

```
python astolfo.py task run directory_delete 1.worker.catboi.net
'{"py/object": "tasks.task_directory_delete.DirectoryDeleteRequest",
"directory_path": "/docker/some"}'
```

Sending a file

```
python astolfo.py task run file_send 1.worker.catboi.net '{"py/object":
"tasks.task_file_send.FileSendRequest", "file_name": "covid1984.yml",
"file_remote_directory": "/docker/covid1984"}'
```

Downloading a file

```
python astolfo.py task run file_download 1.worker.catboi.net '{"py/object":
"tasks.task_file_download.FileDownloadRequest", "file_name":
"covid1984_copy.yml", "file_remote_directory": "/docker/covid1984"}'
```

Deleting a file

```
python astolfo.py task run file_delete 1.worker.catboi.net '{"py/object":
"tasks.task_file_delete.FileDeleteRequest", "file_path":
"/docker/covid1984/covid1984.yml"}'
```

Examples of running tasks for installing packages

Setting up a Zabbix client

```
python astolfo.py task run zabbix_client 1.worker.catboi.net '{"server":
"zabbix.catboi.net"}'
```

Setting up fail2ban

```
python astolfo.py task run fail2ban 1.worker.catboi.net
```

Examples of running tasks for Docker

Run a Docker container

This might get a separate command in the future.

Until then, see `Running arbitrary shell commands`.

Stop and remove a Docker container

This might get a separate command in the future.

Until then, see `Running arbitrary shell commands`.

Examples of running tasks for Docker Compose

Run a Docker Compose stack

This might get a separate command in the future.

Until then, see `Running arbitrary shell commands`.

Stop a Docker Compose stack

This might get a separate command in the future.

Until then, see `Running arbitrary shell commands`.

Examples of running tasks for Docker Swarm

Setting up a Docker Swarm cluster with one master

```
JOIN_TOKENS=$(python astolfo.py task run docker_swarm_cluster
1.manager.catboi.net)
```

Joining a node to the Docker Swarm cluster as a manager

```
python astolfo.py task run docker_swarm_master 2.manager.catboi.net
"$JOIN_TOKENS"
```

Joining a node to the Docker Swarm cluster as a worker

```
python astolfo.py task run docker_swarm_worker 1.worker.catboi.net
"$JOIN_TOKENS"
```

(alternatively, you can just pass in the JSON output from the previous command directly)

Deploy Portainer on the Docker Swarm cluster

```
python astolfo.py task run docker_swarm_portainer 1.worker.catboi.net
```

Set a Docker Swarm cluster node label

```
python astolfo.py task run docker_swarm_label 1.worker.catboi.net
'{"py/object": "tasks.task_docker_swarm_label.DockerSwarmLabelRequest",
"target_host": "1.worker.catboi.net", "target_label": "type=worker"}'
```

Deploy a Docker Swarm stack

This might get a separate command in the future.

Until then, see `Running arbitrary shell commands`.

Delete a Docker Swarm stack

This might get a separate command in the future.

Until then, see `Running arbitrary shell commands`.

Examples of running tasks for Kubernetes (K3S)

Leaving a K3S cluster

```
python astolfo.py task run kubernetes_k3s_leave 1.worker.catboi.net
```

Setting up a K3S cluster with one master

```
JOIN_TOKENS=$(python astolfo.py task run kubernetes_k3s_cluster
1.manager.catboi.net)
```

Joining a node to the K3S cluster as a worker

```
python astolfo.py task run kubernetes_k3s_worker 1.worker.catboi.net
"$JOIN_TOKENS"
```

Getting the KUBECONFIG from the K3S cluster

```
python astolfo.py task run kubernetes_k3s_kubeconfig 1.worker.catboi.net
```

Deploy Portainer on the K3S cluster

```
python astolfo.py task run kubernetes_k3s_portainer 1.worker.catboi.net
```

Set a K3S cluster label

```
python astolfo.py task run kubernetes_k3s_label 1.worker.catboi.net
'{"py/object": "tasks.task_kubernetes_k3s_label.KubernetesLabelRequest",
"target_host": "1.worker.catboi.net", "target_label": "type=worker"}'
```

Set a K3S cluster secret

```
python astolfo.py task run kubernetes_k3s_secret 1.worker.catboi.net
'{"py/object": "tasks.task_kubernetes_k3s_secret.KubernetesSecretRequest",
"secret_type": "docker-registry", "name": "registrycredentials", "values":
["docker-server=https://registry.kronis.dev", "docker-username=KronisLV",
"docker-password=mPT4G5wAePZ5ge44i9FC", "docker-
email=kristians@kronis.dev"]}'
```

Deploy a Kubernetes manifest

This might get a separate command in the future.

Until then, see `Running arbitrary shell commands`.

Delete a Kubernetes deployment

This might get a separate command in the future.

Until then, see `Running arbitrary shell commands`.

4. pielikums

.gitlab-ci.yml - Konfigurācijas paraugs GitLab CI/CD (nepārtrauktajai integrācijai un piegādēm)

```
stages:
  - build_containers
  # TODO add SonarQube stage to scan the code
  # TODO add stage to redeploy to test environment if i ever want to host
this publically

build_astolfo_container:
  stage: build_containers
  retry: 1
  script:
    - echo "logging in to $CI_REGISTRY"
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
$CI_REGISTRY

    - IMAGE_TAG=$CI_REGISTRY/$CI_PROJECT_PATH/astolfo_cloud_servant:latest
    - echo "Building image $IMAGE_TAG"

    - GIT_TAG=$CI_COMMIT_TAG
    - "[ -z $GIT_TAG ] && GIT_TAG=latest"
    - IMAGE_GIT_TAG=$CI_REGISTRY/$CI_PROJECT_PATH/astolfo_cloud_servant:
$GIT_TAG
    - echo "Image Git tag $IMAGE_GIT_TAG"

    - docker build -t $IMAGE_TAG -t $IMAGE_GIT_TAG .
    - docker push $IMAGE_TAG
    - docker push $IMAGE_GIT_TAG
```

5. pielikums

swarm-cluster.sh - Paraugs pilna Docker Swarm klastera inicializācijai ar izstrādāto pārvaldes rīku

```
#!/bin/bash
set -e
cd ../src
source venv/bin/activate

# Install Docker on everything
python astolfo.py task run docker 1.worker.catboi.net
python astolfo.py task run docker 2.worker.catboi.net
python astolfo.py task run docker db.catboi.net
python astolfo.py task run docker tester.catboi.net

# Leave any old cluster
python astolfo.py task run docker_swarm_leave 1.worker.catboi.net
python astolfo.py task run docker_swarm_leave 2.worker.catboi.net
python astolfo.py task run docker_swarm_leave db.catboi.net
python astolfo.py task run docker_swarm_leave tester.catboi.net

# Create a Docker Swarm cluster
JOIN_TOKENS=$(python astolfo.py task run docker_swarm_cluster
1.worker.catboi.net)
echo "$JOIN_TOKENS"
python astolfo.py task run docker_swarm_worker 2.worker.catboi.net
"$JOIN_TOKENS"
python astolfo.py task run docker_swarm_worker db.catboi.net
"$JOIN_TOKENS"
python astolfo.py task run docker_swarm_worker tester.catboi.net
"$JOIN_TOKENS"

# Install Portainer
python astolfo.py task run docker_swarm_portainer 1.worker.catboi.net

# Add labels for deploying COVID1984 workload
python astolfo.py task run docker_swarm_label 1.worker.catboi.net
'{"py/object": "tasks.task_docker_swarm_label.DockerSwarmLabelRequest",
"target_host": "1.worker.catboi.net", "target_label": "type=worker"}'
python astolfo.py task run docker_swarm_label 1.worker.catboi.net
'{"py/object": "tasks.task_docker_swarm_label.DockerSwarmLabelRequest",
"target_host": "2.worker.catboi.net", "target_label": "type=worker"}'
```

kubernetes-cluster.sh - Paraugs pilna Kubernetes klastera inicializācijai ar izstrādāto rīku

```
#!/bin/bash
set -e
cd ../src
source venv/bin/activate

# Install Docker on everything
python astolfo.py task run docker 1.worker.catboi.net
python astolfo.py task run docker 2.worker.catboi.net
python astolfo.py task run docker db.catboi.net
python astolfo.py task run docker tester.catboi.net

# Leave any old cluster
python astolfo.py task run kubernetes_k3s_leave 1.worker.catboi.net
python astolfo.py task run kubernetes_k3s_leave 2.worker.catboi.net
python astolfo.py task run kubernetes_k3s_leave db.catboi.net
python astolfo.py task run kubernetes_k3s_leave tester.catboi.net

# Create a Docker Swarm cluster
JOIN_TOKENS=$(python astolfo.py task run kubernetes_k3s_cluster
1.worker.catboi.net)
echo "$JOIN_TOKENS"
python astolfo.py task run kubernetes_k3s_worker 2.worker.catboi.net
"$JOIN_TOKENS"
python astolfo.py task run kubernetes_k3s_worker db.catboi.net
"$JOIN_TOKENS"
python astolfo.py task run kubernetes_k3s_worker tester.catboi.net
"$JOIN_TOKENS"

# Install Portainer
python astolfo.py task run kubernetes_k3s_portainer 1.worker.catboi.net

# Add the secret for logging into GitLab
python astolfo.py task run kubernetes_k3s_secret 1.worker.catboi.net
'{"py/object": "tasks.task_kubernetes_k3s_secret.KubernetesSecretRequest",
"secret_type": "docker-registry", "name": "registrycredentials", "values":
["docker-server=https://registry.kronis.dev", "docker-username=KronisLV",
"docker-password=mPT4G5wAePZ5ge44i9FC", "docker-
email=kristians@kronis.dev"]}'

# Add labels for deploying COVID1984 workload
python astolfo.py task run kubernetes_k3s_label 1.worker.catboi.net
'{"py/object": "tasks.task_kubernetes_k3s_label.KubernetesLabelRequest",
"target_host": "1.worker.catboi.net", "target_label": "type=worker"}'
python astolfo.py task run kubernetes_k3s_label 1.worker.catboi.net
'{"py/object": "tasks.task_kubernetes_k3s_label.KubernetesLabelRequest",
"target_host": "2.worker.catboi.net", "target_label": "type=worker"}'
```

20201207195946_add_heatmap_generation_stored_procedures.rb - Migrācijas, kuras izveido DB puses procedūras karšu ģenerācijai

```
execute <<~EOL.strip
  -- function for creating a grid, from
https://trac.osgeo.org/postgis/wiki/UsersWikiCreateFishnet
  CREATE OR REPLACE FUNCTION ST_CreateFishnet(
    nrow integer,           -- number of rows in y-direction
    ncol integer,           -- number of columns in x-direction
    xsize float8,           -- cell size length in x-direction
    ysize float8,           -- cell size length in x-direction
    x0 float8 DEFAULT 0,    -- origin offset in x-direction; DEFAULT is 0
    y0 float8 DEFAULT 0,    -- origin offset in y-direction; DEFAULT is 0
    -- OUT "row" integer,
    -- OUT col integer,
    OUT geom geometry
  ) RETURNS SETOF geometry AS $$
  SELECT
    -- i + 1 AS row,
    -- j + 1 AS col,
    ST_Translate(
      cell,
      j * $3 + $5,
      i * $4 + $6
    ) AS geom
  FROM
    generate_series(0, $1 - 1) AS i,
      generate_series(0, $2 - 1) AS j,
    (
      SELECT ('POLYGON((0 0, 0 '||$4||', '||$3||' '||$4||', '||$3||' 0,0
0))')::geometry AS cell
    ) AS foo;
  $$ LANGUAGE sql IMMUTABLE STRICT;
EOL

execute <<~EOL.strip
  CREATE OR REPLACE PROCEDURE regenerate_heatmap(
    v_heatmap_cell_resolution_km integer default 4, -- how many km per
cell edge
    v_heatmap_minutes_ago integer default 1440, -- how many days minutes
will we use data about (1 day by default)
    v_heatmap_unique_batches_to_keep integer default 1 -- how many
different iterations of old data will we keep
  ) AS $$
  DECLARE
    v_country_envelope geometry;
    v_x_min float; v_x_max float;
    v_y_min float; v_y_max float;
    v_horizontal_cells integer; v_vertical_cells integer;
    v_kilometer_size float := 0.0090063014; -- map units per kilometer, a
magic number
```

```
    v_cell_size_x float := v_kilometer_size *
v_heatmap_cell_resolution_km; -- what is the size of a singular cell
    v_cell_vertical_proj float := 0.5; -- because of the way map is
projected, we need to adjust the cell height
    v_cell_size_y float := v_cell_size_x * v_cell_vertical_proj;
    v_series_creation_date timestamp := NOW(); -- so we can use the latest
data for filtering
    i_cell geometry;
  BEGIN
    SELECT ST_Envelope(geometry) INTO v_country_envelope
    FROM named_geometries
    WHERE name = 'latvia_borders';
    RAISE NOTICE 'Created envelope %', ST_AsText(v_country_envelope);
    RAISE NOTICE 'Cell resolution KM: %', v_heatmap_cell_resolution_km;
    RAISE NOTICE 'Cell size X: % and Y: %', v_cell_size_x, v_cell_size_y;

    SELECT ST_XMin(v_country_envelope) INTO v_x_min; RAISE NOTICE 'X min
%', v_x_min;
    SELECT ST_XMax(v_country_envelope) INTO v_x_max; RAISE NOTICE 'X max
%', v_x_max;
    SELECT ST_YMin(v_country_envelope) INTO v_y_min; RAISE NOTICE 'Y min
%', v_y_min;
    SELECT ST_YMax(v_country_envelope) INTO v_y_max; RAISE NOTICE 'Y max
%', v_y_max;

    SELECT CEIL((v_x_max - v_x_min) / v_cell_size_x) INTO
v_horizontal_cells;  RAISE NOTICE 'Horizontal cells: %',
v_horizontal_cells;
    SELECT CEIL((v_y_max - v_y_min) / v_cell_size_y) INTO
v_vertical_cells;    RAISE NOTICE 'Vertical cells: %', v_vertical_cells;

    DELETE FROM heatmap_data
    WHERE time IN (
      SELECT DISTINCT time
      FROM heatmap_data dt
      ORDER BY time DESC
      OFFSET v_heatmap_unique_batches_to_keep - 1
    );

    INSERT INTO heatmap_data
    SELECT
      gen_random_uuid() AS heatmap_data_uuid,
      ST_Centroid(aggregated_data.cell) AS location,
      v_series_creation_date AS time,
      COUNT(aggregated_data.device_uuid) AS intensity,
      v_series_creation_date AS created_at,
      v_series_creation_date AS updated_at
    FROM
    (
      SELECT
        cells.geom AS cell,
        point_data.*
      FROM (
        SELECT ST_SetSRID(ST_CreateFishnet(v_vertical_cells,
```

```
v_horizontal_cells, v_cell_size_x, v_cell_size_y, v_x_min, v_y_min), 4326)
AS geom
      ) cells
      INNER JOIN point_data
        ON ST_Within(point_data.location::geometry, cells.geom::geometry)
      INNER JOIN devices
        ON point_data.device_uuid = devices.device_uuid
      WHERE
        point_data.time > NOW() - (v_heatmap_minutes_ago || '
minutes')::INTERVAL
        AND devices.infected = TRUE
    ) aggregated_data
    GROUP BY
      ST_Centroid(aggregated_data.cell);
  END;
  $$ LANGUAGE plpgsql;
  -- CALL regenerate_heatmap();
EOL

execute <<~EOL.strip
  CREATE OR REPLACE FUNCTION check_point_in_latvia(
    v_location_x float,
    v_location_y float
  ) RETURNS boolean AS $$
  DECLARE
    in_latvia boolean := false;
  BEGIN
    SELECT
    ST_Within(
      ST_SetSRID(ST_MakePoint(v_location_x, v_location_y), 4326),
      (SELECT geometry FROM named_geometries WHERE name =
'latvia_borders')
    ) INTO in_latvia;
    RETURN in_latvia;
  END;
  $$ LANGUAGE plpgsql;
EOL
  end
```

covid1984.yml - paraugs Docker Swarm vides deklarācijai

```
version: '3.4'
services:
  covid1984_app_scheduled:
    image:
registry.kronis.dev/rtu1/kvps5_masters_degree_covid_1984/covid1984:latest
    restart: "unless-stopped"
    depends_on:
      - covid1984_postgis
    environment:
      - POSTGRES_HOST=covid1984_postgis
      - POSTGRES_DB=covid1984
      - POSTGRES_USER=covid1984
      - POSTGRES_PASSWORD=covid1984_nKkSb2VFt1xIJUPtySu8
      - APP_SCHEDULED_ENABLE=true
      - APP_SCHEDULED_FREQUENCY=2400
      - APP_HEATMAP_CELL_RESOLUTION_KM=8
      - APP_HEATMAP_MINUTES_AGO=1440
      - APP_HEATMAP_UNIQUE_BATCHES_TO_KEEP=1
      - APP_FINGERPRINT=2020-12-08_version_1
    deploy:
      placement:
        constraints:
          - node.hostname == 1.worker.catboi.net
      resources:
        limits:
          cpus: "0.90"
          memory: "3000M"
  covid1984_app:
    image:
registry.kronis.dev/rtu1/kvps5_masters_degree_covid_1984/covid1984:latest
    restart: "unless-stopped"
    depends_on:
      - covid1984_postgis
    ports:
      - 80:3000
    environment:
      - POSTGRES_HOST=covid1984_postgis
      - POSTGRES_DB=covid1984
      - POSTGRES_USER=covid1984
      - POSTGRES_PASSWORD=covid1984_nKkSb2VFt1xIJUPtySu8
      - APP_SCHEDULED_ENABLE=false
      - APP_FINGERPRINT=2020-12-08_version_1
    deploy:
      replicas: 2
      placement:
        constraints:
          - node.labels.type == worker
      resources:
        limits:
          cpus: "0.90"
```

```
          memory: "3000M"
  covid1984_postgis:
    image: postgis/postgis:11-3.0
    restart: "unless-stopped"
    ports:
      - 5432:5432
    environment:
      - POSTGRES_DB=covid1984
      - POSTGRES_USER=covid1984
      - POSTGRES_PASSWORD=covid1984_nKkSb2VFt1xIJUPtySu8
    volumes:
      - covid1984_postgis_data:/var/lib/postgresql/data
    deploy:
      placement:
          constraints:
          - node.hostname == db.catboi.net
      resources:
        limits:
          cpus: "0.90"
          memory: "3000M"
volumes:
    covid1984_postgis_data:
```

covid1984kubernetes.yml - paraugs Kubernetes vides deklarācijai

```
apiVersion: v1
items:
- apiVersion: v1
  kind: Service
  metadata:
    annotations:
      kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f
covid1984.yml -o
        covid1984-kubernetes.yml
      kompose.version: 1.21.0 (992df58d8)
    creationTimestamp: null
    labels:
      io.kompose.service: covid1984-app
    name: covid1984-app
  spec:
    type: NodePort
    ports:
    - port: 80
      targetPort: 3000
      protocol: TCP
      name: http
      nodePort: 80
    selector:
      io.kompose.service: covid1984-app
  status:
    loadBalancer: {}
- apiVersion: v1
  kind: Service
  metadata:
    annotations:
      kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f
covid1984.yml -o
        covid1984-kubernetes.yml
      kompose.version: 1.21.0 (992df58d8)
    creationTimestamp: null
    labels:
      io.kompose.service: covid1984-postgis
    name: covid1984-postgis
  spec:
    ports:
    - name: "5432"
      port: 5432
      targetPort: 5432
    selector:
      io.kompose.service: covid1984-postgis
  status:
    loadBalancer: {}
- apiVersion: apps/v1
  kind: Deployment
  metadata:
```

```
    annotations:
      kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f
covid1984.yml -o
        covid1984-kubernetes.yml
      kompose.version: 1.21.0 (992df58d8)
    creationTimestamp: null
    labels:
      io.kompose.service: covid1984-app
    name: covid1984-app
  spec:
    replicas: 2
    selector:
      matchLabels:
        io.kompose.service: covid1984-app
    strategy: {}
    template:
      metadata:
        annotations:
          kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f
covid1984.yml
            -o covid1984-kubernetes.yml
          kompose.version: 1.21.0 (992df58d8)
        creationTimestamp: null
        labels:
          io.kompose.service: covid1984-app
      spec:
        containers:
        - env:
          - name: APP_FINGERPRINT
            value: 2020-12-08_version_1
          - name: APP_SCHEDULED_ENABLE
            value: "false"
          - name: POSTGRES_DB
            value: covid1984
          - name: POSTGRES_HOST
            value: covid1984-postgis
          - name: POSTGRES_PASSWORD
            value: covid1984_nKkSb2VFt1xIJUPtySu8
          - name: POSTGRES_USER
            value: covid1984
          image:
registry.kronis.dev/rtu1/kvps5_masters_degree_covid_1984/covid1984:latest
          imagePullPolicy: ""
          name: covid1984-app
          ports:
          - containerPort: 3000
          resources:
            requests:
              cpu: 100m
              memory: "512000000"
            limits:
              cpu: 900m
              memory: "3145000000"
        nodeSelector:
```

```
          type: worker
        restartPolicy: Always
        serviceAccountName: ""
        volumes: null
        imagePullSecrets:
        - name: registrycredentials
  status: {}
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    annotations:
      kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f
covid1984.yml -o
        covid1984-kubernetes.yml
      kompose.version: 1.21.0 (992df58d8)
    creationTimestamp: null
    labels:
      io.kompose.service: covid1984-app-scheduled
    name: covid1984-app-scheduled
  spec:
    replicas: 1
    selector:
      matchLabels:
        io.kompose.service: covid1984-app-scheduled
    strategy: {}
    template:
      metadata:
        annotations:
          kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f
covid1984.yml
            -o covid1984-kubernetes.yml
          kompose.version: 1.21.0 (992df58d8)
        creationTimestamp: null
        labels:
          io.kompose.service: covid1984-app-scheduled
      spec:
        containers:
        - env:
          - name: APP_FINGERPRINT
            value: 2020-12-08_version_1
          - name: APP_HEATMAP_CELL_RESOLUTION_KM
            value: "8"
          - name: APP_HEATMAP_MINUTES_AGO
            value: "1440"
          - name: APP_HEATMAP_UNIQUE_BATCHES_TO_KEEP
            value: "1"
          - name: APP_SCHEDULED_ENABLE
            value: "true"
          - name: APP_SCHEDULED_FREQUENCY
            value: "2400"
          - name: POSTGRES_DB
            value: covid1984
          - name: POSTGRES_HOST
            value: covid1984-postgis
```

```
          - name: POSTGRES_PASSWORD
            value: covid1984_nKkSb2VFt1xIJUPtySu8
          - name: POSTGRES_USER
            value: covid1984
          image:
registry.kronis.dev/rtu1/kvps5_masters_degree_covid_1984/covid1984:latest
          imagePullPolicy: ""
          name: covid1984-app-scheduled
          resources:
            requests:
              cpu: 100m
              memory: "512000000"
            limits:
              cpu: 900m
              memory: "3145000000"
        nodeSelector:
          kubernetes.io/hostname: 1.worker.catboi.net
        restartPolicy: Always
        serviceAccountName: ""
        volumes: null
        imagePullSecrets:
        - name: registrycredentials
  status: {}
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    annotations:
      kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f
covid1984.yml -o
        covid1984-kubernetes.yml
      kompose.version: 1.21.0 (992df58d8)
    creationTimestamp: null
    labels:
      io.kompose.service: covid1984-postgis
    name: covid1984-postgis
  spec:
    replicas: 1
    selector:
      matchLabels:
        io.kompose.service: covid1984-postgis
    strategy:
      type: Recreate
    template:
      metadata:
        annotations:
          kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f
covid1984.yml
            -o covid1984-kubernetes.yml
          kompose.version: 1.21.0 (992df58d8)
        creationTimestamp: null
        labels:
          io.kompose.service: covid1984-postgis
      spec:
        containers:
```

```
        - env:
          - name: POSTGRES_DB
            value: covid1984
          - name: POSTGRES_PASSWORD
            value: covid1984_nKkSb2VFt1xIJUPtySu8
          - name: POSTGRES_USER
            value: covid1984
          image: postgis/postgis:11-3.0
          imagePullPolicy: ""
          name: covid1984-postgis
          ports:
          - containerPort: 5432
          resources:
            requests:
              cpu: 100m
              memory: "512000000"
            limits:
              cpu: 900m
              memory: "3145000000"
          volumeMounts:
          - mountPath: /var/lib/postgresql/data
            name: covid1984-postgis-data
        nodeSelector:
          kubernetes.io/hostname: db.catboi.net
        restartPolicy: Always
        serviceAccountName: ""
        volumes:
        - name: covid1984-postgis-data
          persistentVolumeClaim:
            claimName: covid1984-postgis-data
  status: {}
- apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    creationTimestamp: null
    labels:
      io.kompose.service: covid1984-postgis-data
    name: covid1984-postgis-data
  spec:
    accessModes:
    - ReadWriteOnce
    resources:
      requests:
        storage: 100Mi
  status: {}
kind: List
metadata: {}
```