VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Bakalaurinis darbas

Beserverinių funkcijų šalto paleidimo laiko optimizavimas

(Optimizing Serverless Functions Cold Start Time)

Atliko: 4 kurso 4 grupės studentas
Darius Andzevičius (parašas)
Darbo vadovas:
Gediminas Rimša (parašas)

Vilnius
2021

Turinys

Sąvokų apibrėžimai

IaaS – Infrastruktūra kaip paslauga (angl. Infrastructure as a Service)
PaaS – Platforma kaip paslauga (angl. Platform as a Service)
SaaS – Programinė įranga kaip paslauga (angl. Software as a Service)
FaaS – Funkcija kaip paslauga (angl. Function as a Service)
CaaS – Konteineriai kaip paslauga (angl. Container as a Service)
IoT – Daiktų internetas (angl. Internete of Things)
AWS – kompanijos *Amazon* debesų kompiuterijos platforma *Amazon Web Services*
IT – informacinės technologijos
VM – virtuali mašina
AOT (angl. Ahead of Time compilation) – išankstinis kompiliavimas

Įvadas

Debesų kompiuterija suteikia informacinių technologijų sistemoms priemones, kurios eliminuoja poreikį turėti fizinę techninę įrangą, kurioje ši sistema veiks, taip optimizuojant sistemos pradinius paleidimo bei vėlesnius palaikymo kaštus. Jos principas - egzistuojančios techninės įrangos ir virtualizacijos panaudojimas formuojant bendrai naudojamą infrastruktūrą [GRE+12]. Perėjimas prie šios paradigmos (angl. paradigm shift) padarė didelę įtaką naujų informacinių sistemų atsiradimui bei esamų sistemų palaikymui. Tai - perėjimas nuo kompiuterijos, kaip produkto turėjimo, į kompiuteriją kaip paslaugą, kuri suteikiama varotojams internetu iš didelės apimties duomenų centrų - „debesų“ [KSS10].

Iki 2014 m. vyravo trys pagrindiniai debesų kompiuterijos paslaugų modeliai - *Infrastruktūra kaip paslauga* (IaaS), *Platforma kaip paslauga* (PaaS) ir *Programinė įranga kaip paslauga* (SaaS). Šie modeliai suteikia naujas galimybes informacinių technologijų sistemoms, tačiau tuo pačiu kelia neįprastus tradicinei kompiuterijai iššūkius, į kuriuos turėtų būti atsižvelgta prieš naudojantis debesų kompiuterijos paslaugomis [GRE+12]. 2014 m. kompanija *Amazon* pristatė savo valdomoje debesų kompiuterijos platformoje *Amazon Web Services (toliau - AWS)* veikiantį naują paslaugų modelį - *funkcija kaip paslauga* (FaaS), kurį realizuoja *AWS Lambda* platforma. FaaS tikslas - panaikinti vartotojo poreikį rūpintis infrastruktūra, kurioje veikia jo aplikacija. *Amazon* buvo pirmoji įmonė, siūlanti šio tipo paslaugas [Lin14].

Pastarasis paslaugų modelis įgyvendina vadinamąją skaičiavimų be serverių (angl. serverless computing) sistemos architektūrą, kurios pagrindinis principas - vykdyti programas - išskaidytas aplikacijos dalis virtualiuose konteineriuose - kurios kiekviena individualiai gali būti skaliuojamos pagal poreikį. FaaS paslaugų tiekėjai siūlo sumažintus aplikacijos išlaikymo kaštus (angl. hosting costs), aukštą pasiekiamumą (angl. high availability), tolerancją klaidoms (angl. fault tolerance) bei dinaminį elastingumą (angl. dynamic elasticity) automatiškai prižiūrimoje ir valdomoje infrastruktūroje, kurioje šie konteineriai veikia [LRC+18]. Vienintelė vartotojo atsakomybė - pateikti programinį kodą, FaaS paslaugų modelio kontekste vadinamą funkcija, paslaugos tiekėjo platformai tinkamu formatu.

Terminas „*beserverinių skaičiavimų*“ sistema jokiu būdu nereiškia, kaip gali pasirodyti, jog fiziniai serveriai neegzistuoja. Tai terminas, apibūdinantis programavimo modelį ir sistemos architektūrą, kurioje programinio kodo dalys yra patalpinamos ir vykdomos debesyje, nereikalaujant vartotojui rūpintis aplikacijos infrastruktūra - vartotojas palieka šią atsakomybę debesų kompiuterijos tiekėjui.

FaaS paslaugų modelyje, tiekėjas rūpinasi vartotojo programinio kodo ir jo naudojamų bibliotekų konteinerizavimu (angl. containerization) debesyje, šių konteinerių išplečiamumu (angl. scalability), apkrovos paskirtymu (angl. load balancing) tarp jų. Vartotojas platfomos tiekėjui turi mokėti tik už patalpintos funkcijos iškvietimų skaičių ir veikimo laiką debesyje. Tokį modelį puikiai išnaudoja įvairios multimedijos apdorojimo, IoT duomenų agregavimo, srautų apdorojimo (angl. stream processing), pokalbių robotų (angl. chatbot), planuotų darbų (angl. scheduled job),

REST API, nuolatinės integracijos (angl. continuous integration), nuolatinio diegimo (angl. continuous deployment) bei nuolatinio pristatymo (angl. continuous delivery) aplikacijos [IBM20].

Nors savo savybėmis FaaS yra naudingas ir patrauklus paslaugų modelis, informacinių technologijų sistemos pritaikymas šiam modeliui yra ganėtinai sudėtingas ir turintis nemažai neapibrėžtumų. Iš pirmo žvilgsnio, beserverinių skaičiavimų platformos gali pritraukti dėmesį bet kokios aplikacijos migravimui į šią platfomą, tačiau yra sunku nuspėti, kokią įtaką aplikacijos migravimas į beserverinių skaičiavimų platformą turės aplikacijos greitaveikai, atsako laikui bei kaip gerai platforma sugebės paskirstyti apkrovą. Todėl prieš perkeliant aplikaciją į beserverinių skaičiavimų platfomą, būtina suprasti jos trūkumus, kad vartotojas galėtų įvertinti, ar šis paslaugų modelis tinka panaudos atvejui.

Darbo tikslas: nustatyti būdus optimizuoti JVM funkcijos šalto paleidimo laiką ir ištirti jų veiksmingumą.

Siekiant šio darbo tikslo, iškelti tokie **uždaviniai**:

1. Nustatyti būdus optimizuoti JVM funkcijos šalto paleidimo laiką.
2. Praplėsti [And21a] sukurtą įrankį beserverinių funkcijų šalto paleidimo trukmės matavimui atlikti kodo paketo archyvu pateikiamoms funkcijoms.
3. Ištirti nustatytus funkcijos šalto paleidimo laiką optimizuojančius būdus sukurtu matavimo įrankiu *AWS Lambda* platformoje JVM programavimo kalbos funkcijai.
4. Įvertinti ištirtų optimizavimo būdų pritaikymo galimybes.

1. Debesų kompiuterijos pradmenys

Iki debesų kompiuterijos atsiradimo, įmonės, norėdamos plėtoti IT sistemą, susidurdavo su įvairiomis, sunkiai sprendžiamomis techninėmis problemomis. Apsunkinančios aplinkybės, norint paleisti IT sistemą, buvo fizinių serverių pirkimas, jų išlaikymas ir palaikymas, įvairių techninių problemų sprendimas, tokių kaip defektuotos dalys, elektros srovės praradimas ar interneto ryšio trikdžiai. Didžioji dalis įmonių buvo pačios atsakingos už fizinių serverių palaikymą - nuo įvairių programų rašymo, kurios valdydavo įvykusias klaidas, ugniasienių (angl. firewall) bei interneto tinklo komutatorių konfigūravimo, iki serverių elektros sanaudų skaičiavimo. Toks infrastruktūros modelis buvo neefektyvus norint greitai ir kokybiškai vystyti elastingą ir sparčiai plečiamą IT sistemą.

2006-ųjų metų rūgpjūčio mėnesį įvyko tai, kas fundamentaliai pakeitė požiūrį į IT sistemų infrastruktūrą. *Amazon Web Services* pristatė pirmąją viešai prieinamą debesų kompiuterijos paslaugą *Elastic Compute Cloud (EC2)*. Ši IaaS tipo paslauga, pasitelkiant techninės įrangos virtualizaciją, suteikė galimybę IT sistemas laikyti *AWS* serveriuose nuomojantis jų dalis. Tai leido įmonėms, programuotojų grupėms ar įvairiems startuoliams, neturintiems didelio pradinio kapitalo serverių pirkimui, išsinuomoti serverius ir juos pasiekti per kelias minutes, o kaštai, reikalingi IT sistemos infrastruktūrai, sumažėjo dešimtimis kartų.

Šią paslaugą galime vadinti infrastruktūros ranga (angl. outsourcing) - tai yra visos IT sistemos kūrimo proceso dalies, kuri yra bendra ir visiems kitiems, kuriantiems IT sistemas - nuo fizinių patalpų, skirtų serveriams laikyti, iki bendrų aplikacijų savybių, tokių kaip autentifikacija - perkėlimas į bendrai naudojamą platformą. Tai turi kelis didelius pranašumus, lyginant šią paslaugą su tradicine IT infrastruktūra, kuri pasižymi tuo, jog vartotojo kompiuterinė įranga priklauso jam pačiam:

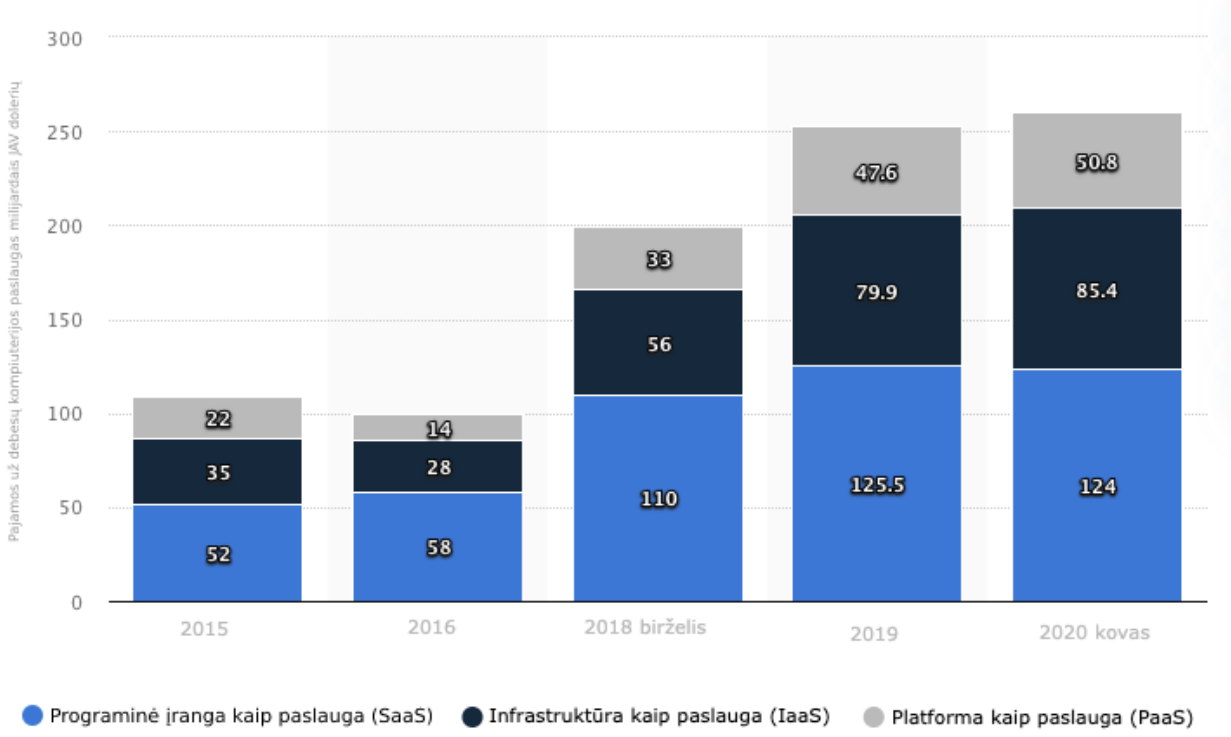
- Masto ekonomija (angl. economy of scale) - principas, leidžiantis debesų kompiuterijos tiekėjui, diegiančiam fizinę infrastruktūrą didelėmis apimtimis, įsigyti ir prižiūrėti šią įranga mažesne kaina, nei individualūs asmenys ar įmonės atskirai.

- Rizika. Mažesnė dalis specialistų turi turėti tam tikrų infrastruktūros žinių, reikalingų serverių priežiūrai. Specialistai susiduria su pasikartojančiomis problemomis, todėl gali greičiau spręsti įvykusias infrastruktūrines klaidas.

- Sistemos plečiamumas. Debesų kompiuterijos tiekėjas siūlo milžiniškas apimtis kompiuterinės įrangos išteklių, kas leidžia vartotojui lengvai prisitaikyti prie kuriamos sistemos reikalaujamų išteklių.

- Laikas nuo idėjos iki jos IT sistemos realizizavimo. Idėjos autoriui nėra būtina galvoti apie jo sistemos infrastruktūros pusę, nes ja jau pasirūpino debesų kompiuterijos tiekėjas - vartotojui belieka prisitaikyti prie paslaugų tiekėjo siūlomų paslaugų.

Šie pranašumai leidžia debesų kompiuterijos tiekėjui siūlyti kompetetingas paslaugų kainas, o vartotojui pirkti šias paslaugas apsimoka labiau, nei turėti savo fizinę infrastuktūrą. Todėl debesų kompiuterijos panaudojimas pastaruosius metus sparčiai populiarėjo, o gautos pajamos paslaugų tiekėjams leido efektyviai vystyti savo platformas ir siūlyti pastoviai tobulėjančias paslaugas (1 pav.).



1 pav. Debesų kompiuterijos pajamos (pagal [Mli21])

1.1. FaaS paslaugų modelis

Debesų kompiuterijos paslaugų tiekėjai turi dideles apimtis kopiuterinės įrangos, kurios virtualias dalis nuomoja vartotojams teikdami įvairias debesų kompiuterijos paslaugas. Dalis šios įrangos nėra nuolat išnaudojama, todėl jos išlaikymas nėra pelningas. Šią, pastoviai vienų ar kitų paslaugų neišnaudojamą įrangos dalį, galima panaudoti užsakomųjų skaičiavimų paslaugų (angl. on-demand service) infrastruktūrai realizuoti. Paprastai tariant, tai yra FaaS paslaugos principas - funkcija yra talpinama tuo metu nenaudojamoje kompiuterinės įrangos dalyje, o vėliau funkcijos naudojama įrangos dalis yra atlaisvinama, kai ši funkcija tampa nebenaudojama tam tikrą laiko tarpą.

Tokiu būdu paslaugos tiekėjas leidžia vartotojui paleisti ir laikyti savo aplikaciją ar jos dalis kaip atskiras, lengvai plečiamas, neturinčias būsenos (angl. stateless), laikinas (angl. ephemeral) funkcijas, talpinamas virtualiuose konteineriuose. Šių funkcijų gyvavimo ciklu (angl. lifecycle) rūpinasi pats paslaugų tiekėjas, sukurdamas jas, kai to prireikia, arba sunaikindamas jas, kai funkcijos tampa nebenaudojamos tam tikrą laiko tarpą. Toks principas vartotojui suteikia galimybę

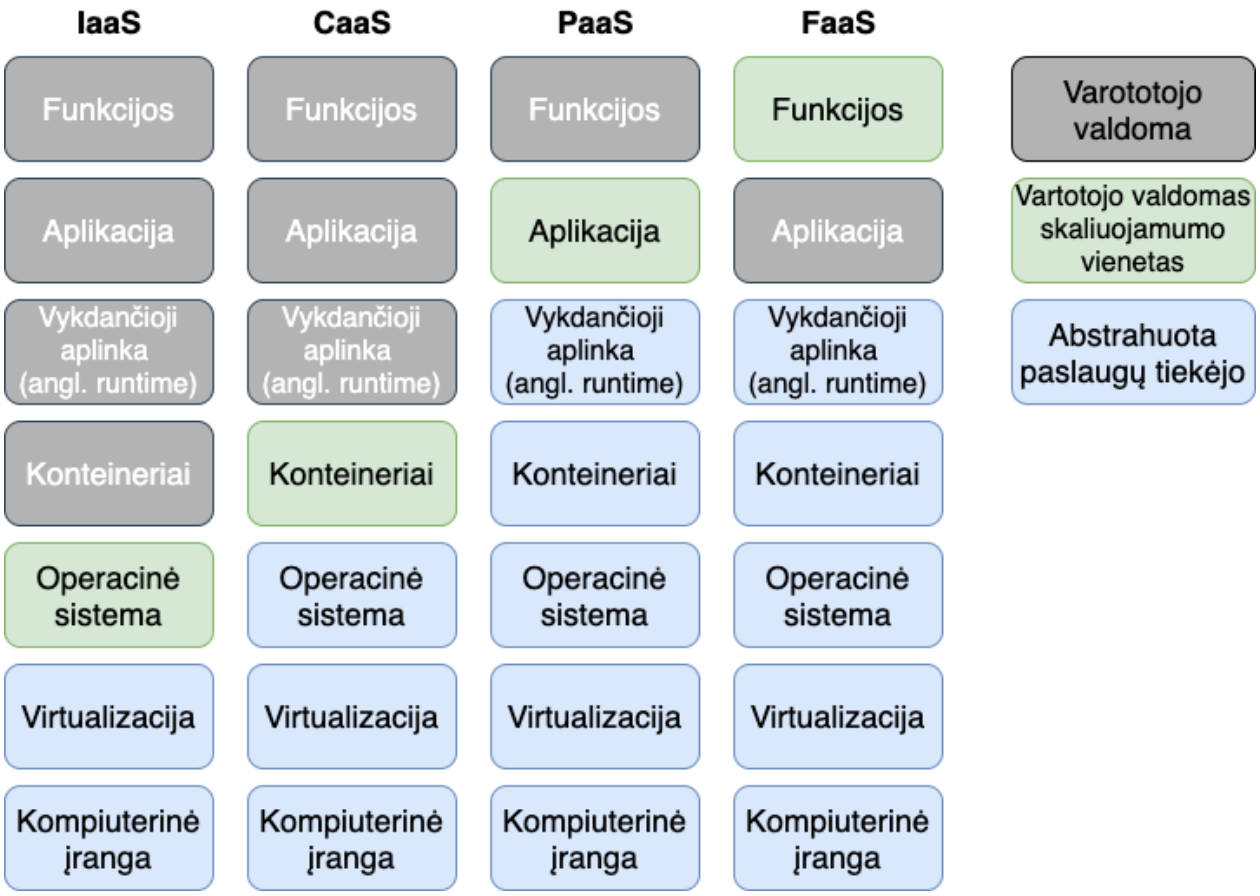
nesirūpinti infrakstruktūra, kurioje ši funkcija atliks darbą – vienintelis jo tikslas lieka sukurti bei sukonfiguruoti šią programą paslaugų tiekėjo platformai.

Nuomojamos įrangos dalies dydis yra konfiguruojamas vartotojo, priklausomai nuo to, kiek jo talpinama funkcija reikalauja išteklių, tačiau verta paminėti, kad ši konfiguracija yra ribojama – FaaS paslaugų tiekėjai leidžia skirtingus išskiriamų išteklių funkcijai dydžius, pvz. *AWS Lambda* platforma leidžia funkcijai išskirti net iki 10GB atminties [AWS20], kai tuo tarpu *Google Cloud* FaaS platforma *Cloud Functions* – iki 4GB [Goo20]. Priklausomai nuo išskiriamo atminties dydžio, funkcijai suteikiamas tam tikras kiekis procesoriaus galios, kuri daugelyje FaaS platformų apibrėžiama skirtingai. Vartotojas moka už šios atminties panaudojimą per laiką, pvz. *AWS Lambda* platformos 1GB atminties panaudojimas sekundei daugelyje regionų kainuoja 0.0000166667 dolerio, apvalinant funkcijos veikimo laiką 1 ms tikslumu [AWS21c].

Toks granuliarus FaaS paslaugos apmokestinimo modelis yra labai patrauklus vartotojui, nes jam netenka mokėti už jo aplikacijos neišnaudojamus kompiuterinius išteklius, o programinio kodo optimizavimas greitaveikai tiesiogiai konvertuojasi į mažesnę infrastruktūros kainą.

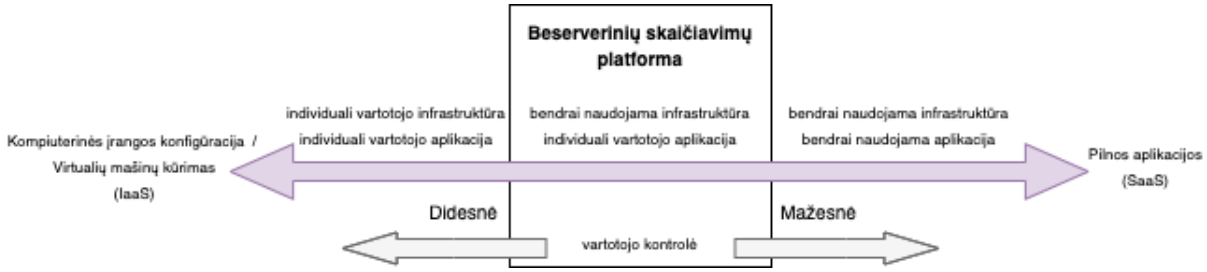
1.2. FaaS kitų debesų kompiuterijos paslaugų kontekste

Kiekvienas debesų kompiuterijos paslaugų modelis abstrahuoja tam tikras atsakomybes nuo vartotojo perleisdamas atsakomybę paslaugos tiekėjui (2 pav.), taip įgalindamas greitesnį aplikacijos funkcionalumo plėtojimą ir paleidimą bei mažindamas išlaikymo kaštus ir reikalingas vartotojo žinias apie skirtingus aplikacijos sluoksnius. FaaS modelio atveju, vartotojas rūpinasi tik programinio kodo aplikacijos sluoksniu, kuris yra talpinamas debesų kompiuterijos tiekėjo debesyje.



2 pav. Debesų kompiuterijos paslaugų lyginimas (pagal [McK16])

FaaS modelis iš esmės įgyvendina beserverinių skaičiavimų architektūrą, todėl šiuos du terminus galime laikyti sinonimais. Kad galėtumėme geriau suprasti pagrindines šios architektūros savybes, galime lyginti ją su kitais debesų kompiuterijos paslaugų modeliais IaaS ir SaaS. Vienas iš būdų paaiškinti beserverinių skaičiavimų architektūrą yra galvoti apie tai, kokią atsakomybę turi vartotojas, naudodamasis minėtais debesų kompiuterijos paslaugų modeliais (3 pav.). *Infrastruktūros kaip paslaugos* modelyje vartotojas turi daugiausia atsakomybės rūpintis kaip jo paties aplikacija bus paleista, plečiama bei kaip bus skirstoma apkrova. Priešingai, *programinės įrangos kaip paslaugos* modelyje, vartotojui naudojasi paslaugos tiekėjo sukurta programine įranga ir tuo pačiu neturi žinoti, kokia infrastruktūra supa šią įrangą. Galvodami apie beserverinių skaičiavimų architektūrą, galime įsivaizduoti, kad FaaS paslaugos modelis yra kažkur tarp IaaS ir SaaS – vartotojui pakanka savo sukurtą aplikaciją paleisti paslaugų tiekėjo, kuris pasirūpins infrastruktūra, debesyje.



3 pav. FaaS lyginant su IaaS ir SaaS (pagal [BCC+17])

Pagrindinis beserverinių skaičiavimų architektūros principas - galimybė išplėsti ir sutraukti aplikacijos dalių kopijų skaičių tiek, kiek tuo metu to reikalauja apkrova. Tokiu būdu esant nulinei apkrovai, vartotojas neturi mokėti debesų kompiuterijos tiekėjui už tuo metu nenaudojamus kompiuterinius išteklius, o esant didelei apkrovai, aplikacijos veikimas nestringa ir ši reaguoja ir vykdo užklausas taip pat efektyviai.

Šį principą įgyvendina FaaS paslaugų tiekėjas - vartotojui, kreipiantis į beserverinių skaičiavimų aplikaciją, debesų kompiuterijos platformos tiekėjo tarpinė programinė įranga (angl. middleware) nusprendžia, kuriai funkcijai priklauso ši užklausa, suranda konteinerį, kurioje ji yra patalpinta, ir nukreipia užklausą į ją. Jei šis konteineris neegzistuoja, arba egzistuoja, bet yra apkrautas vykdydamas kitą užklausą, platforma turi sukurti naują konteinerį, į kurį bus nukreipta vartotojo užklausa. Dažnu atveju, paslaugų tiekėjo FaaS platforma yra sukurta taip, jog būtų lengva naudotis šio tiekėjo kitomis debesų kompiuterijos paslaugomis, tokiomis kaip duomenų bazės ar duomenų saugyklos. Dažnu atveju, paslaugų tiekėjas savo platformoje siūlo pranešimų perdavimo sistemos funkcionalumą, kuriuo yra įgyvendinamas funkcijų komunikavimas bei duomenų perdavimas tarp jų, sukuriantis tam tikrą procesą (4 pav.). Funkcijai baigus darbą, platformos tikslas grąžinti atsakymą į vartotojo užklausą, surinkti jos vykdymo matus ir sustabdyti funkciją [BCC+17].

Šios architektūros funkcionalumo įgyvendinimo sunkumas yra toks, jog reikia atsižvelgti į sistemos kainą, plečiamumą, tolerancijią klaidoms bei funkcijų orkestravimą. FaaS platforma turi užtektinai greitai sukurti ir paleisti naują funkcijos konteinerį, kai to prireikia, perduoti jam užklausą ir grąžinti atsakymą. Taip pat platforma turi sugebėti dėti užklausas ar atskirų funkcijų rezultatus į pranešimų eiles, iš kurių šie pranešimai turi būti vykdomi nepažeidžiant tvarkos. Platforma turi gebėti dorotis su dideliais kiekiais užklausų ir efektyviai išplėsti sistemą. Taip pat efektyviai turi ją ir sutraukti tam, kad vartotojas neturėtų mokėti papildomai [BCC+17].

4 pav. Beserverinių skaičiavimų architektūra (pagal [Mic21])

Didelė FaaS platformos funkcijų plečiamumo problema yra „šalti“ funkcijos pasileidimai. FaaS platforma, neradusi funkcijos, tuo metu neapdorojančio užklausos, turi sukurti naują ir perduodi užklausą jai. Šios funkcijos sukūrimas užima šiek tiek laiko, kurio metu, aplikacijos vartotojas laukia atsakymo iš sistemos. Paprastai, funkcijos sukūrimas bei paleidimas atliekamas keliais žingsniais. Pirmiausia, FaaS platforma į virtualia mašiną, kurioje bus sukurtas konteineris, talpinantis funkcijos kodą, turi atsisiųsti vykdomo kodo paketą iš paslaugos tiekėjo duomenų saugyklos, kurioje yra saugomas šis kodo paketas. Po šio žingsnio, sistema turi sukurti patį konteinerį ir į jo atmintį užkrauti atsisiųstą kodo paketą. Galiausiai konteineris turi paleisti kodo paketą esantį atmintyje, įvykdyti vartotojo užklausą ir grąžinti rezultatą. Kaip vėlesniame darbo skyrelyje matysime, šis laiko tarpas priklauso nuo įvairių veiksnių.

1.3. FaaS trūkumai

Tam, kad visi FaaS platformos privalumai būtų pilnai išnaudoti, vartotojas turi iš anksto gerai apgalvoti kokiomis dalimis bus skaidoma jo aplikacija, kaip vyks komunikacija tarp funkcijų ir jų orkestravimu, užtikrinti, kad aplikacijos greitaveika bei atsako laikas nenukentėtų tose dalyse, kur tai yra svarbu – talpinamos funkcijos turi būti atitinkamai optimizuotos, o kai kurios aplikacijos dalys galbūt išvis neturėtų būti talpinamos į FaaS platformą.

Dėl šių priežasčių, FaaS platformos turi ir neigiamų savybių - kadangi FaaS debesų kompiuterijos paslaugų modelis yra ganėtinai naujas, šie aplikacijos talpinimo procesai ir gerosios praktikos nėra standartizuoti, todėl yra sunku optimaliai pritaikyti aplikaciją FaaS platformai.

Kai kurie FaaS platformų trūkumai kyla iš pačio sistemos principo - jie niekada nebus visiškai pašalinti, o tobulėjant FaaS plaformoms, bus išmokstama kaip su jais tvarkytis. Įvardinsime ryškiausius FaaS platformų trūkumus.

Gaišties laikas

Įprastoje - ne beserverinių skaičiavmų sistemos architektūroje - ar tai būtų monolitinė, ar mikroservisų architektūros stiliumi paremta sistema, tam tikri sistemos procesai yra įgyvendinami izoliuotos programos kontekste. Tokiu atveju įgyvendinamo proceso sritis (angl. scope) yra klasių ar modulių rinkinys, kuriuo naudojantis proceso įgyvendinimas vyksta metodų, naudojančių bendrus kompiuterinius išteklius, kvietimų pagalba. Šiuo būdu yra ganėtinai efektyviai išnaudojami kompiuteriniai ištekliai ir tai nedaro didelės įtakos sistemos greitaveikai. Kadangi beserverinių skaičiavimų sistemos architektūra yra paremta funkcijų skaidymu ir jų izoliavimu tam, kad ši būtų efektyviai skaliuojama, tie patys procesai beserverinių skaičiavimų plaftomoje gali būti įgyvendinti keliomis funkcijomis, kurios tarpusavyje komunikuoja ne naudodamosis bendrais kompiuteriniais ištekliais, o tinklu taip sukurdamos papildomą gaišties laiką.

Būsena

Kadangi kiekviena FaaS platformos funkcija gali būti išplečiama iki daugelio identiškų kopijų, veikiančių nepriklausomai viena nuo kitos, arba sutraukiama iki nei vienos, funkcijos neišlaiko savo būsenos (angl. state). Norint, kad funkcija turėtų tam tikrą būseną, ji privalo komunikuoti su komponentais, kurie laiko būseną ir gali šią būseną perduoti funkcijai. Dėl šios priežasties, atsiranda gaišties laikas, kurį FaaS platformoje norime sumažinti kiek galima labiau.

Aplikacijos lokalus testavimas

Beserverinių skaičiavimų architektūros programinį kodą yra sunku testuoti lokalioje aplinkoje, nes vartotojai nežino ir dėl to negali atkartoti FaaS platformos implementacijos detalių. Taip pat, dažnu atveju funkcijos turi bendrauti su kitomis debesų kompiuterijos teikiamomis paslaugomis, kaip žinučių perdavimo sistemos, duomenų bazės ar duomenų saugyklos. Šią komunikaciją yra sunku tiksliai atkartoti iš lokalios aplinkos.

Konfigūravimo kontrolės praradimas

Debesų kompiuterijos tiekėjai riboja konfigūracines FaaS platformos galimybes tam, kad išvengtų nekontroliuojamo sistemos išnaudojimo. Funkcijos kopijų skaičius, išskiriami kompiuteriniai ištekliai, maksimalus funkcijos veikimo laikas ar užklausos dydis ir kitos panašios konfi-

gūracijos - parametrai, kuriuos kiekvienas debesų kompiuterijos tiekėjas yra nustatęs pagal savo infrastruktūros galimybes.

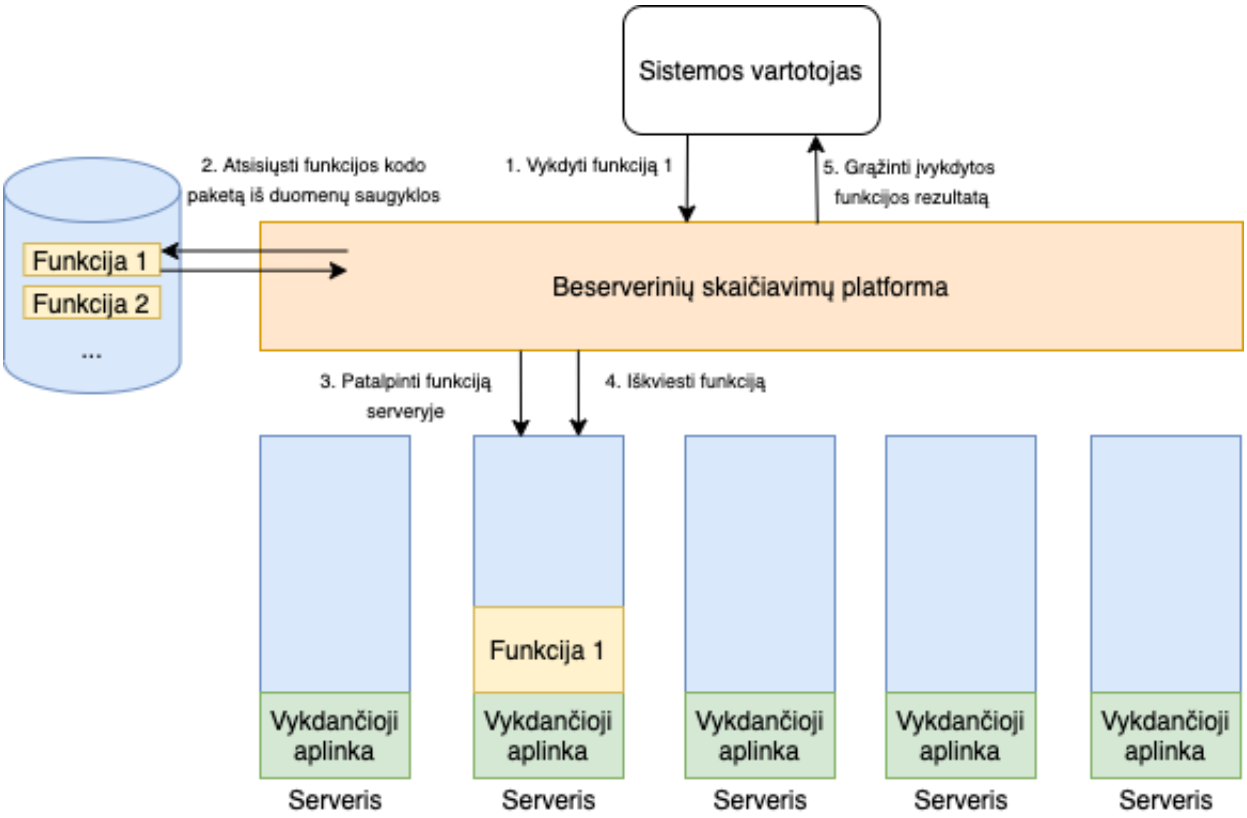
„Šalti“ funkcijų paleidimai

Tai turbūt didžiausias FaaS platformos trūkumas, kuris neretai atgraso vartotojus nuo naudojimosi šia platforma. Jo problematiškumas prikaluso nuo įvairių veiksnių, kuriuos stengiamasi sušvelninti tam, kad „šalti“ funckijų paleidimai užimtų kuo mažiau laiko. Būtent dėl šios FaaS platformų charakteristikos, ne visos aplikacijos turėtų būti talpinamos FaaS platformoje - ypatingai tos, kurios pateikia daug įvairios informacijos vartotojui, kuri grąžinama iš skirtingų serverio punktų (angl. endpoints), ar tos, kurios reikalauja sulaukti atsako tam tikram veiksmui, po kurio yra tęsiamas tam tikras procesas.

Taigi, norint laikyti savo aplikaciją FaaS platformoje, reikia atsižvelgti į daugelį veiksnių ir nuspręsti, ar tikrai tai bus naudinga ir sistemos vartotojui, ir sistemos savininkui. Dažniau FaaS platformoje turėtų būti talpinamos staigių, didelio kiekio duomenų apdorojimo, asincrhoniškos, pvz., paveikslėlių ar vaizdo įrašų apdirbimo, IoT duomenų agregavimo ar planuotų darbų vykdymo aplikacijos, nes tai nereikalauja kiek įmanoma greitesnio atsako laiko ir yra vykdomos ganėtinai retai, todėl ir kaštai vartotojui yra mažesni, nei šias aplikacijas talpinant IaaS platformoje. Tačiau FaaS platformų technologija spačiai tobulėja, minėtos problemos yra identifikuojamos ir šalinamos debesų kompiuterijos tiekėjų, ar švelninama jų daroma įtaka vartotojui.

2. Veiksniai, darantys įtaką „šaltam“ funkcijos paleidimui

Kaip ir buvo minėta praeitame skyriuje, beserverinių skaičiavimų platforma, norėdama įvykdyti sistemos vartotojo užklausą, turi atlikti keletą žingsnių iki užklausos rezultatas grąžinamas vartotojui (5 pav.).



5 pav. Beserverinių skaičiavimų vykdymo modelis (pagal [Hai18])

„Šaltas“ funkcijos paleidimas reiškia tai, jog vykdydama vartotojo užklausą beserverinių skaičiavimų platforma turi sukurti vienoje iš serverių virtualioje mašinoje konteinerį, į kurį atsiųs funckijos kodo paketą, kartu su vykdančiosios aplinkos priklausomybėmis, jį paleis ir perduos užklausos vykdymą. Funkcijos paleidimo procesą orkestruoja paslaugų tiekėjo platforma. Tiesa, funkcijos kodo paketo atsiuntimas ir paleidimas vadinamas „tiekėjo šaltu“ (angl. provider cold) paleidimu, kuris nutinka, kai funkcijos duomenys po tam tikro nebenaudojimo laiko yra ištrinami iš VM FaaS platformos tiekėjo, kurioje ši veikia, dėl ko kitu funkcijos kvietimu funkcijos kodo paketas turi būti atsiųstas iš naujo. Šiame darbe yra nagrinėjamas funkcijos „konteinerio šaltas“ (angl. container cold) paleidimas - programinis kodas, ar tai būtų konteinerio atvaizdas, ar archyvas, jau egzistuoja virtualioje mašinoje, kurioje funkcija bus paleista, FaaS platformai lieka jį inicijuoti ir paleisti funkciją. Panagrinėkime šio proceso dalis, kad galėtumėme suprasti kritinius jo taškus, dėl kurių atsiranda pridėtinis (angl. overhead) laikas vykdant vartotojo užklausą.

2.1. Funkcijos kodo paketo atsiuntimas iš duomenų saugyklos

Pirmasis žingnis, kurį atlieka beserverinių skaičiavimų platforma, gavusi sistemos vartotojo užklausą - randa patalpintą funkcijos kodo paketą duomenų saugykloje - funkcijų registre. Šis registras yra paprasčiausia paslaugų tiekėjo duomenų saugykla, pvz. AWS paslaugų tiekėjo atveju tai *AWS Elastic Container Services* atvaizdų saugykla ar *AWS S3* objektų saugykla [Hai18]. Beserverinių skaičiavimų platformai radus funkcijos kodo paketą yra sukuriamas konteineris, į kurį šį funkcijos kodą platforma perduos tinklu kartu su jo vykdančiąja aplinka, sudiegs jame ir perduos užklausą vykdymui. Šis žingsnis apibūdina „tiekėjo šalto“ funkcijos paleidimo sąvoką ir dėl jo gaunamas didžiausias pridėtinis laikas nuo užklausos gavimo iki atsakymo grąžinimo, todėl norima šį laiką kiek įmanoma sumažinti. Šio žingsnio sparta priklauso nuo keletos veiksnių. Kadangi funkcijos po tam tikro laiko tarpo nenaudojimo yra ištrinamos ir jų naudojami kompiuteriniai ištekliai atlaisvinami, vėliau kiekviena naujai kuriama funkcija gali atsidurti vis kitame serveryje ar kitoje anksčiau naudojamo serverio virtualioje mašinoje, todėl kiekvieną kartą kuriant naują funkcijos konteinerį yra iš naujo atsiunčiamas funkcijos kodo paketas ir patalpinamas jame. *AWS Lambda* platformai perpanaudojus ankstesniu funkcijos kvietimu sukurtą konteinerį funkcijos iškvietimas yra laikomas „šiltu“, tačiau tai nėra šio darbo tyrimo objektas. Kodo atsiuntimas į konteinerį vyksta tinklu, todėl funkcijos kodo paketo dydis tiesiogiai daro įtaką konteinerio paleidimo laikui. Funkcijos inicijavimo laikas yra glaudžiai susijęs su atvaizdo, kuriuo pateikiamas funkcijos programinis kodas, dydžiu, todėl yra itin svarbu šį dydį minimizuoti [And21a]. Funkcijos programinį kodą pateikiant archyvu yra tikimasi gauti panašius matavimų rezultatus.

2.2. Programinio kodo paketo pateikimas *AWS Lambda* platformai

Funkcijos programinio kodo pateikimas *AWS Lambda* platformai galimas dviem būdais:

- Programinio kodo archyvu (`.zip` arba `.jar`)
- Virtaualaus konteinerio atvaizdu

Iš esmės šie du programinio kodo pateikimo būdai beveik nesiskiria - abejais įmanoma pasiekti tą patį funkcionalumą, tačiau pateikiant programinį kodą archyvu yra įmanomas platesnis funkcijos išorinis konfigūravimas - prie platformai pateikiamo funkcijos programinio kodo galima prijungti įvairias išplėstis (angl. extensions), kitus programinio kodo sluoksnius (angl. layers) netalpinant šių komponentų į archyvą, kai tuo tarpu funkcija ir jos programinis kodas, pateikiamas virtaualaus konteinerio atvaizdu, nėra konfigūruojamas - atvaizdai yra nekintami (angl. immutable) ir tik skaitomi (angl. read-only), todėl sluoksniai bei išplėstys turi būti talpinami pačiuose funkcijos virtaualių konteinerių atvaizduose [Woo20].

Kodo pateikimas archyvu leidžia minimizuoti *AWS Lambda* platformai pateikiamo funkcijos programinio kodo paketo dydį, o taip pat perpanaudoti programinį kodą tarp daugelio funkcijų šį prijungiant sluoknsiu ar išplėstimi.

Didelis privalumas pateikiant funkcijos kodą virtaualaus konteinerio atvaizdu yra šio atvaizdo dydžio riba. Virtaualaus konteinerio atvaizdą *AWS Lambda* platformai galima pateikti iki 10 GB, kai tuo tarpu archyvu tiesiai iš vartotojo kompiuterinės įrangos – iki 50 MB, o archyvą patalpinus *AWS S3* objektų saugykloje ir naudojant ją – iki 250 MB [AWS21e]. Tai leidžia didesnės apimties programoms išnaudoti *AWS Lambda* siūlomus privalumus.

2.3. Funkcijai suteikiamos atminties dydis

Nurodydami funkcijai jai reikalingą atminties dydį, beserverinių skaičiavimų platfoma funkcijai proporcingai suteikia tam tikrą procesoriaus galios dydį, pvz. AWS Lambda paslaugų tiekėjo atveju, funkcijai suteikus 3538 MB atminties, ji gauna apytiksliai du kartus daugiau procesoriaus galios, nei tai pačiai funkcijai suteikus 1769 MB atminties [AWS21b]. Šiuos kompiuterinius išteklius galima panaudoti paspartinti funkcijos inicijavimą, ypatingai JVM kalbų atveju, kurios inicializuotis užtrunka ganėtinai ilgai, lyginant su interpretuojamomis programavimo kalbomis [Rob20]. Oficialioje *AWS Lambda* dokumentacijoje rašoma, jog *Lambda* funkcija, gaunanti 1769 MB atminties, atitinkamai gauna 1 vCPU (1 vCPU sekundės kreditą per sekundę) [AWS21d]. Tai reiškia, jog funkcija skaičiavimams atlikti per sekundę gali pilnai išnaudoti vieną virtualaus procesoriaus galios dydį. Atitinkamai, funkcijai suteikiant dvigubai tiek atminties – 3538 MB, ši galės naudoti vieno virtualaus procesoriaus galios dydį dvi sekundes per sekundę. Tai daro įtaką tiek funkcijos veikimo metu, tiek funkcijos inicijavimo fazėje, kurios metu vykdomas kodas už funkcijos valdiklio. Taip pat funkcijai suteikiant daugiau atminties, ši atitinkamai gauna daugiau procesoriaus loginių gijų, pvz., funkcija, turinti 1769 MB atminties gauna 2 procesoriaus logines gijas, kai tuo tarpu 3538 MB atminties funkcija gauna 3 procesoriaus logines gijas [And21a]. Šias gijas galima išnaudoti efektyvinant funkcijos inicializacijos metu vykdomą kodą – lygiagretinant procesus, tokius kaip ryšio užmezgimas su išoriniais servisais ar statinių duomenų, naudojamų programos, užkrovimas. Tačiau tai nepaspartina funkcijos konteinerio, kuriame veikia vartotojo pateikta funkcija, sukūrimo laiko, už kurį atsakinga *AWS Lambda* platforma – šis laikas ganėtinai pastovus visų atminties dydžiu funkcijoms [And21a; EUG+21], todėl inicijavimo metu veikiantį programinį kodą verta optimizuoti išnaudojant funkcijai suteikiamos atminties kiekį kiek įmanoma efektyviau.

2.4. Programavimo kalbai specifiniai veiksniai

Verta panagrinėti alternatyvius būdus specifinės programavimo kalbos funkcijos inicijavimui paspartinti – naudoti optimizuotą vykdymo aplinką, efektyvinančią programos veikimo bei paleidimo spartą, panaudoti iš anksto sukompiliuoto (AOT, angl. ahead-of-time) programinio kodo kompiliavimo principą, kuris leidžia išvengti kodo interpretavimo programos veikimo metu. JVM programavimo kalbų atveju programinis kodas yra vykdomas Java virtualios mašinos, kurios paleidimas užima nemažą laiko dalį. Šių kalbų atveju yra įmanoma sukurti programos vykdomąją aplinką į ją įtraukiant tik tuos modulius, kurie bus naudojami programos vykdymo metu, taip su-

mažinant Java virtualios mašinos į atmintį užkraunamų klasių kiekį bei sumažinant vykdomosios aplinkos dydį. Tai galima atlikti įrankiu `jlink`, pristatytu Java 9 versijoje kartu su kitomis Java moduliarumo galimybėmis. AOT JVM funkcijos kodo kompiliavimą galima pasiekti pasinaudojant `GraalVM` vykdymo aplinka, kuria programinį kodą yra įmanoma sukompiliuoti į vykdomąjį failą, vadinamą *native image*. Šis programinis failas yra kuriamas `GraalVM` kompiliatoriaus pagalba statinės programinio kodo analizės būdu, peržvelgiant kiekvieną galimą programos veikimo kelią, nekompiliuojant nepasiekiamo kodo, pritaikant įvairius kodo optimizavimo principus, proceso rezultate sukuriant minimalaus dydžio mašininio kodo rinkinį [WSH+19]. Tai, be abejo, yra neįprastas būdas kompiliuojamoms programavimo kalboms, tokioms kaip Java, pasižyminčiomis dinamiškumu, programinio kodo generavimu veikimo metu, todėl šis įrankis turi savų trūkumų bei sunkumų - `GraalVM` veikia „uždaro pasaulio“ principu, dėl ko visos klasės turi būti žinomos kompiliavimo metu. Java ypatybės, tokios, kaip dinaminis klasių užkrovimas, JNI (angl. Java Native Interface), reflekcija, dinaminis įgaliojimas (angl. Dynamic Proxy) suteikia lankstumo programuotojui, o kartu ir neužtikrintumo AOT kompiliatoriui [Wan21], tačiau šiuos aspektus yra įmanoma sukonfigūruoti rankiniu būdu taip leidžiant `GraalVM` žinoti apie galimus kintamųjų tipus bei kitus dinaminius aspektus programos veikimo metu.

3. Tyrimo metodika

Kiekvienam funckijos kvietimui *AWS Lambda* grąžina standartinį atsakymą, kuriame galime rasti informaciją apie funkcijos veikimo laiką bei kitus matavimus:

```
Duration: 54.27 ms Billed Duration: 6131 ms Memory Size: 512 MB
Max Memory Used: 85 MB Init Duration: 6076.61 ms
```

AWS Lambda vartotojui praneša kiek funkcijai buvo išskirta atminties (*Memory Size*), kiek ši funkcija naudojo daugiausiai atminties (*Max Memory Used*), kiek laiko buvo vykdyta užklausa (*Duration*), kiek laiko truko funkcijos inicijavimas *Lambda* platformoje (`Init Duration`) bei už kokį laiko tarpą yra apmokestinamas vartotojas (*Billed Duration*), kuris susideda iš vykdymo bei inicijavimo laiko ir yra apvalinamas 1 ms tikslumu. Funkcijos inicijavimas yra viena iš *Lambda* funkcijos gyvavimo ciklo fazių, kuri vykdoma pirmojo šios funkcijos kvietimo metu. Jos metu yra sukuriama arba „atšildoma“ vykdymo aplinka, atsiunčiamas funkcijos kodo paketas kartu su kitais jam priklausančiais funkcijos sluoksniais, inicijuojamos išplėstys (angl. extensions), jei funkcija tokias naudoja ir įvykdomas kodas, esantis už funkcijos valdiklio (angl. handler) [AWS21a]. Tai iš esmės apibūdina funkcijos „konteinerio šaltą“ paleidimą, todėl matavimuose remsimės `Init Duration` laiku. Šis laikas *AWS Lambda* platformos įrašomas tik tuo atveju, kai funkcija paleidžiama šaltai.

Už funkcijos valdiklio sukursime `boolean` tipo kintamąjį `isCold`, kuris inicijuojamas reikšme `true`. Funkcija, iškviesta pirmąjį kartą - paleista šaltai, pakeis šio kintamojo reikšmę į `false` ir įrašys žinutę, jog funkcija buvo paleista šaltai. Ne pirmąjį kartą kviečiama funkcija šios žinutės neįrašinės ir tai veiks kaip saugiklis tuo atveju, jei dėl tam tikrų priežaščių funkcija bus paleidžiama šiltai.

Java programavimo kalbos *Lambda* funkcijų inicijavimo laikas pasižymi nemažu standartiniu nuokrypiu, todėl kiekvieną funkciją matuosime po 100 kartų ir pavaizduosime kaip varijuoja tiriamas laikas su tam tikra funkcijos konfigūracija.

Visoms matuojamoms funkcijos suteiksime 4096 MB atminties - šiame tyrime nenagrinėsime matuojamų funkcijų inicijavimo laiko priklausomybės nuo joms suteiktos atminties dydžio, nes funkcijai skiriamos atminties bei procesoriaus galingumo panaudojimo galimybės optimizuojant šalto paleidimo laiką priklauso nuo pačios funkcijos ir jos atliekamų veiksmų šioje fazėje, todėl kiekviena funkcija šiuos kompiuterinius išteklius gali panaudoti skirtingai.

Lygindami skirtingais įrankiais sukurtus funkcijos kodo paketus, pateikiamus *AWS Lambda* platformai, pavaizduosime geriausius gautus rezultatus funkcijos kodo paketo dydžio atžvilgiu, o taip pat sulyginsime šių paketų dydžius pridėję į juos nenaudojamų failų taip sulygindami matavimo pagrindą tam, kad galėtumėme pamatyti naudojamų kodo pakavimo būdų pranušumus bei trūkumus.

3.1. *Jlink* įrankiu sukurta vykdymo aplinka

Savo testuojamai funkcijai `jlink` įrankio pagalba sukursime Java vykdomąją aplinką, į kuria įtrauksime tik tuos Java modulius, kuriuos naudoja mūsų funkcija, taip sumažindami vykdymo aplinkos dydį bei neužkraudami nenaudojamų modulių JVM startavimo metu.

Pateikdami funkcijos programinį kodą kaip virtaualaus konteinerio atvaizdą, šią sukurtą vykdomąją aplinką pateiksime kartu su `alpine:latest` baziniu atvaizdu taip minimizuodami atvaizdo dydį. Šiuos matavimus lyginsime su identiško programinio kodo funkcija, naudojančia įprastą *AWS* vykdymo aplinkos bazinį atvaizdą `amazoncorretto:11-alpine` bei leidžiančią mūsų programinį kodą kaip vykdomąjį `.jar` archyvą.

Kodą pateikdami `.zip` archyvu *jlink* įrankiu sukurta vykdomosios aplinkos daromą įtaką išmatuosime dviem būdais. Pirmuoju - vykdomąją aplinką pateikdami vienu archyvu kartu su programiniu kodu, o antruoju - prijungdami šią aplinką atskiru sluoksniu po funkcijos sukūrimo. Abu šių funkcijų inicijavimo laikus lyginsime su funkcijos, kurios programinį kodą sudaro sukompiliuotų klasės failų archyvas, o Java 11 vykdymo aplinka pateikiama *AWS Lambda* platformos, inicijavimo laiku.

3.2. *Native image* vykdomasis failas

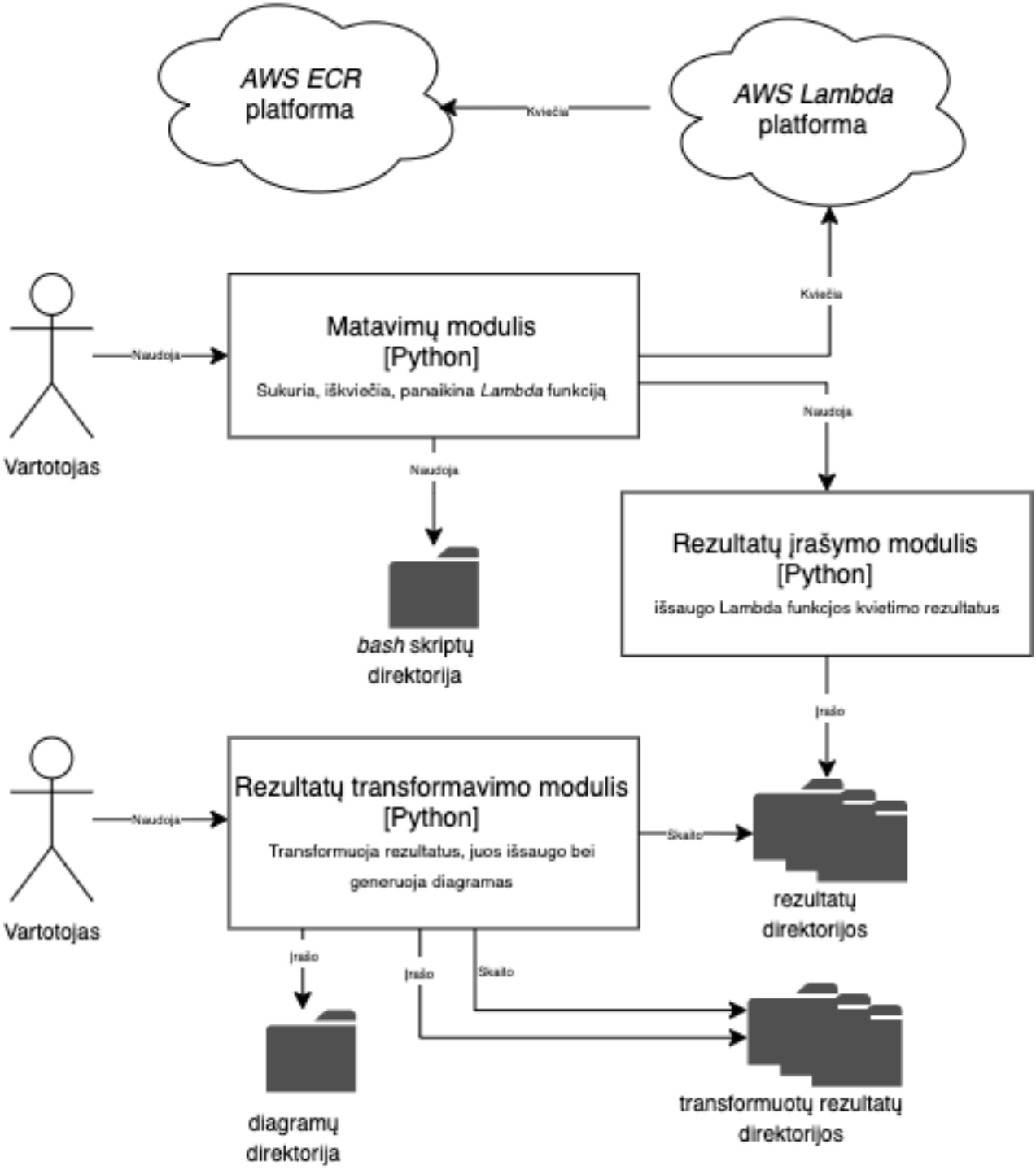
`GraalVM` vykdomosios aplinkos pagalbiniu įrankiu *native-image* sukompiliuosime savo funkcijos programinį kodą į vykdomąjį failą, kurį sudaro programinio kodo klasės, priklausomybių klasės, vykdomojo laiko priklausomybių klasės bei statiškai surišti JDK moduliai. Šis failas susideda iš mašininio kodo, skirto *Linux* operacinei sistemai AMD64 instrukcijų aibės architektūrai.

Kurdami virtaualaus konteinerio atvaizdą savo testuojamai funkcijai pasinaudosime `GraalVM` baziniu atvaizdu `ghcr.io/graalvm/graalvm-ce:latest`. Kūrimo metu pasinaudosime kelių fazių atvaizdo kūrimo procesu, kurio primoje fazėje sugeneruosime *native image* vykdomąjį failą, o antroje fazėje pateiksime šį vykdomąjį failą kartu su `debian:buster-slim` baziniu atvaizdu taip minimizuodami atvaizdo dydį.

Jau sukompiliuotą *native image* vykdomąjį failą rankiniu būdu išimsime iš prieš tai sukurto virtaualaus konteinerio atvaizdo paleisdami šį konteinerį ir išeksportuodami jo turinį. Šį vykdomąjį failą archyvuosime ir pateiksime *AWS Lambda* platformai `.zip` archyvu.

3.3. Matavimams atlikti naudojamos programos aprašymas

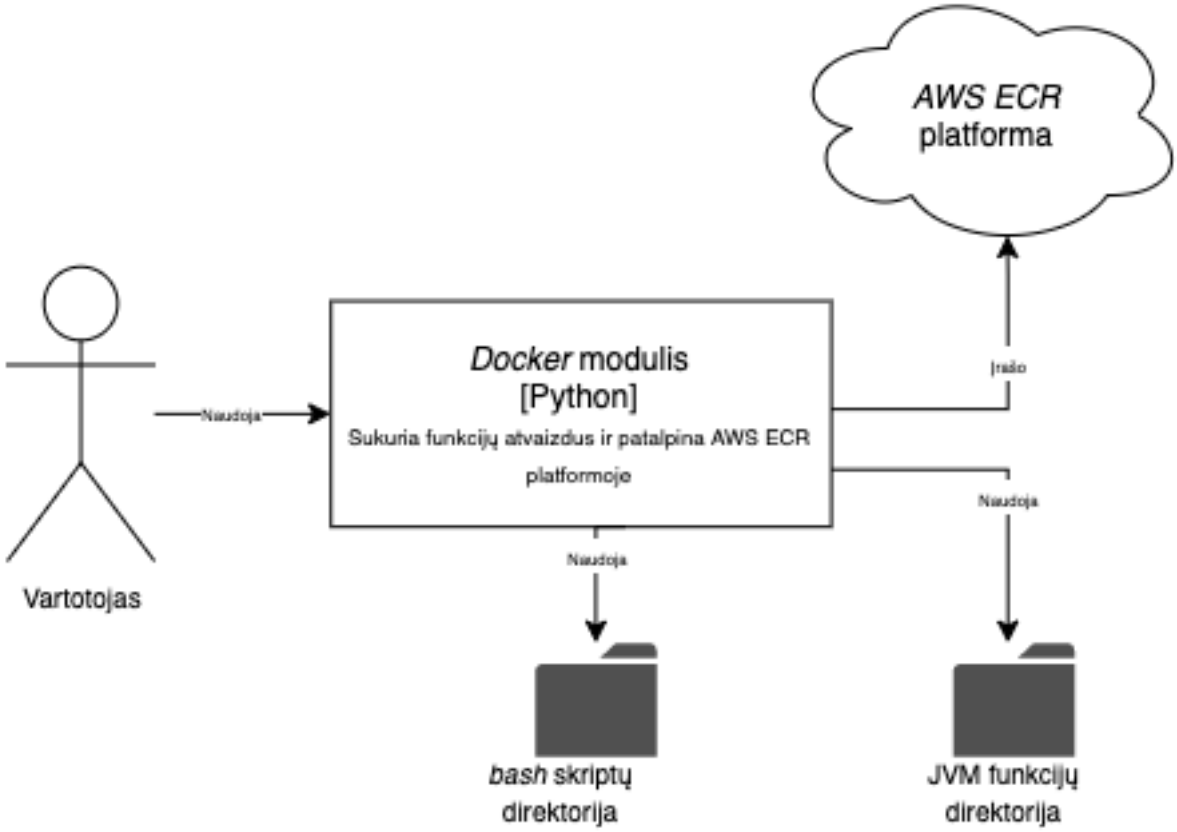
Matavimams atlikti *Python* programavimo kalba buvo suprogramuota aplikacija, automatizuojanti funkcijos sukūrimo, iškvietimo bei sunaikinimo veiksmus - programinis kodas pateikiamas `Git` repozitorijoje [And21b]. Šiems veiksmams atlikti naudojami *bash* skriptai, kurie vykdo *AWS* tekstinės sąsajos įrankio (angl. AWS Command line interface) komandas. Iš *AWS Lambda* platformos funkcijos iškvietimo rezultatai išsaugomi ir kaupiami atitinkamoje funkcijos versijos direktorijoje, iš kurios vėliau šie rezultatai yra nuskaitomi, transformuojami ir gaunami matuojami duomenys. Šiam procesui įgyvendinti suprogramuotas matavimų modulis (6 pav.).



6 pav. Matavimų modulis

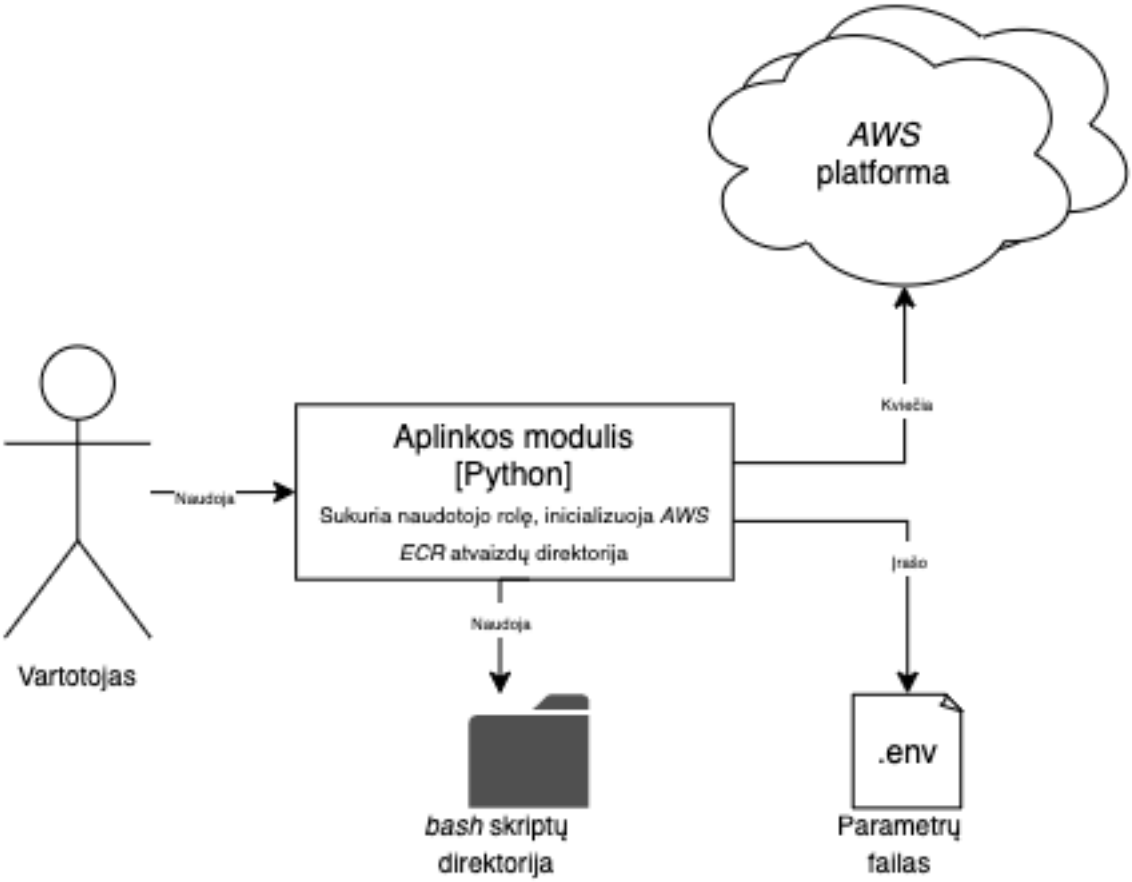
AWS Lambda platformai funkcijas pateiksime *Docker* atvaizdo forma, kuri iš jų sukurs virtualius konteinerius ir juose patalpins mūsų programinį kodą. Kiekviena matuojama funkcija patalpinta atskiroje direktorijoje, kurioje egzistuoja *Dockerfile* failas, iš kurio bus kuriamas konteinerio atvaizdas. Šie atvaizdai bus talpinami *AWS ECR* (angl. Elastic Container Registry) platformoje, iš kurios *AWS Lambda*, kurdama funkcijų konteinerius, atsisiųs ir panaudos šiuos atvaizdus. Atvaizdų generavimui ir patalpinimui *Amazon ECR* platformoje suprogramuotas *Docker* modulis (7

pav.).



7 pav. *Docker* modulis

Prieš naudojantis *AWS ECR* platforma, vartotojas joje turi sukurti atvaizdų repozitoriją. Tam automaziuoti sukurtas aplinkos modulis, kuris *bash* skriptų pagalba *AWS* platformoje sukuria vartotojo rolę, kuri bus naudojama atlikti veiksmus platformoje, o taip pat inicializuoja naują *AWS ECR* repozitoriją. Duomenis, kurie bus reikalingi darbui su *Lambda* funkcijomis, išsisaugo konfigūraciniame `.env` faile (8 pav.).



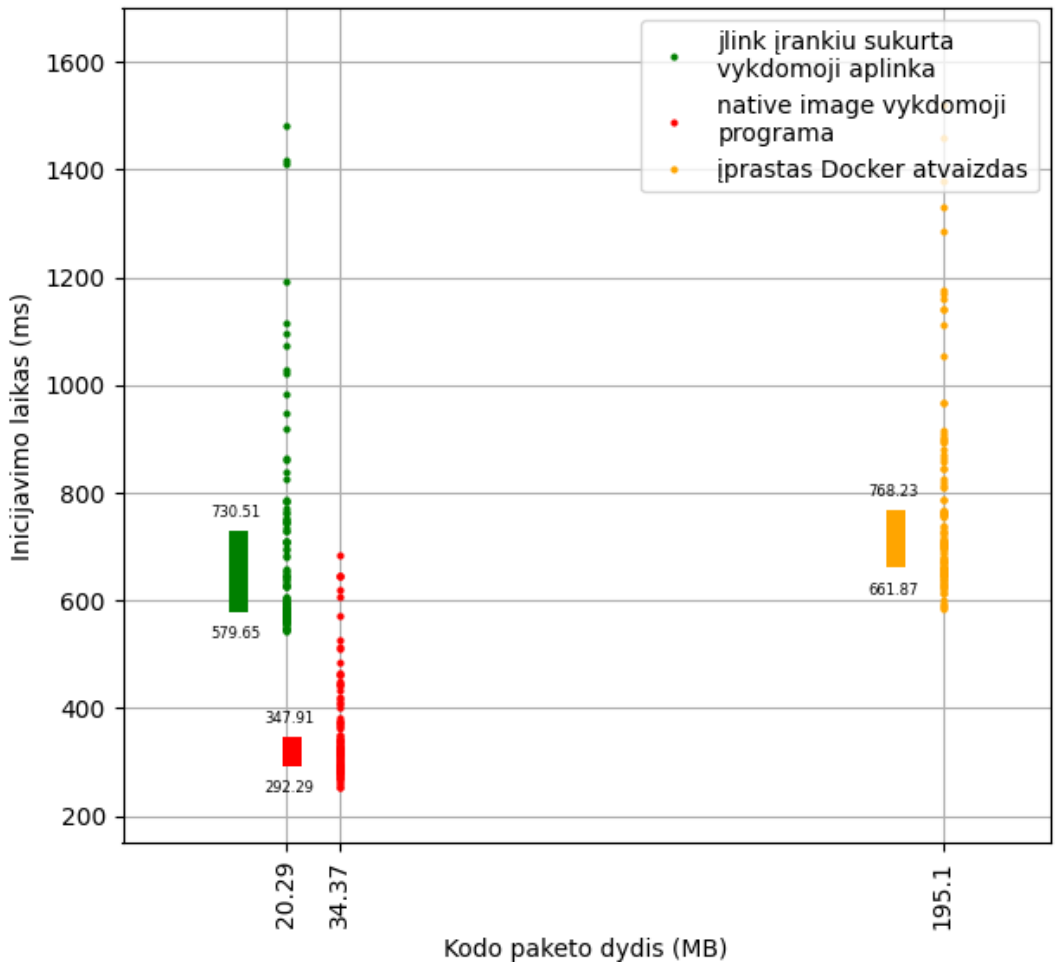
8 pav. Aplinkos konfigūracijos modulis

4. Skirtingais būdais pateikiamų kodo paketų inicijavimo laiko tyrimas

Matavimai buvo atlikti *AWS* `eu-west-1` regione. Verta paminėti, kad gauti rezultatai gali priklausyti nuo paros laiko, kurio metu buvo atlikti funkcijų matavimai - esant didesnei *AWS* serverių apkrovai, funkcijos inicijavimo laikas gali prailgėti, tačiau tai šiame darbe nebus tiriama. Taip pat FaaS paslaugų tiekėjai, tarp jų ir *AWS*, suvokia apie šaltų funkcijos paleidimų sukeliamas problemas ir stengiasi švelninti jų daromą įtaką, todėl atliekant tuos pačius matavimus po tam tikro laiko tarpo rezultatai gali skirtis.

4.1. Konteinerio atvaizdu pateikiamo kodo paketo tyrimas

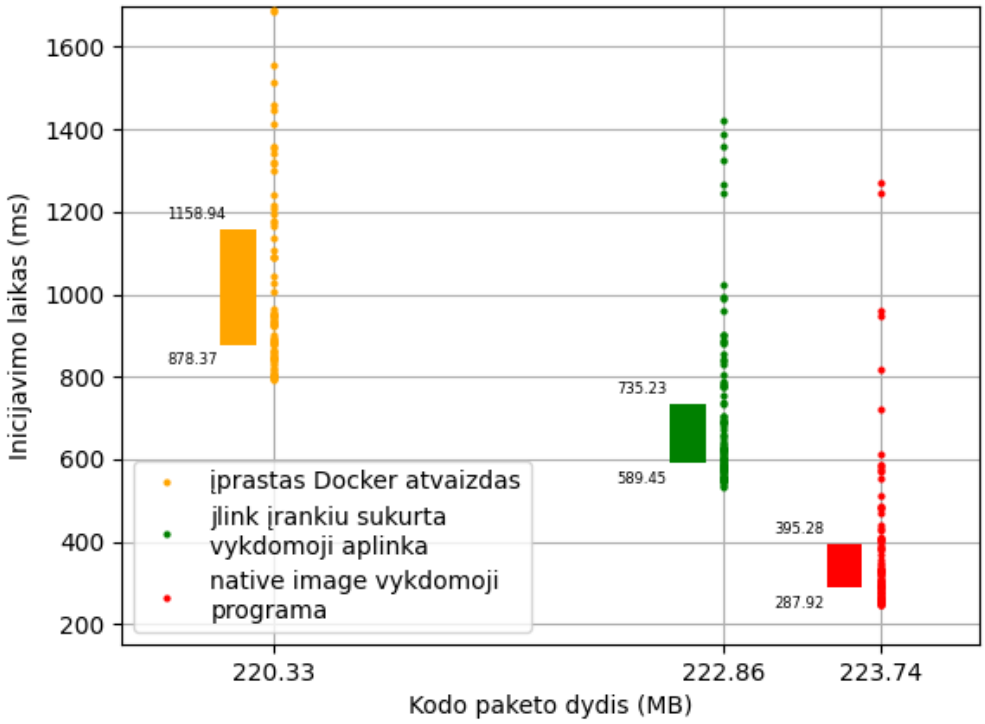
Konteinerio atvaizdu pateikiamo kodo paketo matavimų rezultatus pavaizduosime sklaidos diagrama. Šalia sklaidos taškų pavaizduosime reikšmes, esančias tarp 30 ir 70 procentilių.



9 pav. Skirtingais būdais pateikiamo funkcijos programinio kodo atvaizdu matavimai

Iš (9 pav.) matavimų rezultatų matome, jog *native image* vykdomoji programa, nors neturinti mažiausio atvaizdo dydžio, sugebėjo inicijuotis sparčiausiai – vidutiniškai per 318 ms. Nors `jlink` įrankiu sukurtos vykdomosios aplinkos virtualaus konteinerio atvaizdas užėmė mažiausiai vietos, tačiau jo inicijavimo greitis panašus į atvaizdo, užimančio 195,1 MB. Taip pat galime pastebėti, jog atvaizdu pateikiamo funkcijos programinio kodo inicijavimo laikai pasižymi didele variacija – tam galimai turi įtakos tam tikri platformos atliekami optimizacijos veiksmai. Taip pat yra tikėtina, jog mūsų matavimuose naudojamos funkcijos *AWS Lambda* platformos yra talpinamos skirtinguose daugiau ar mažiau apkrautuose serveriuose, kas irgi gali turėti įtakos šiai variacijai paaiškinti, tačiau norint pagrįsti šias mintis reikėtų detalesnių matavimų, tiriančių aplinkos, kurioje mūsų funkcijos veikia, savybes.

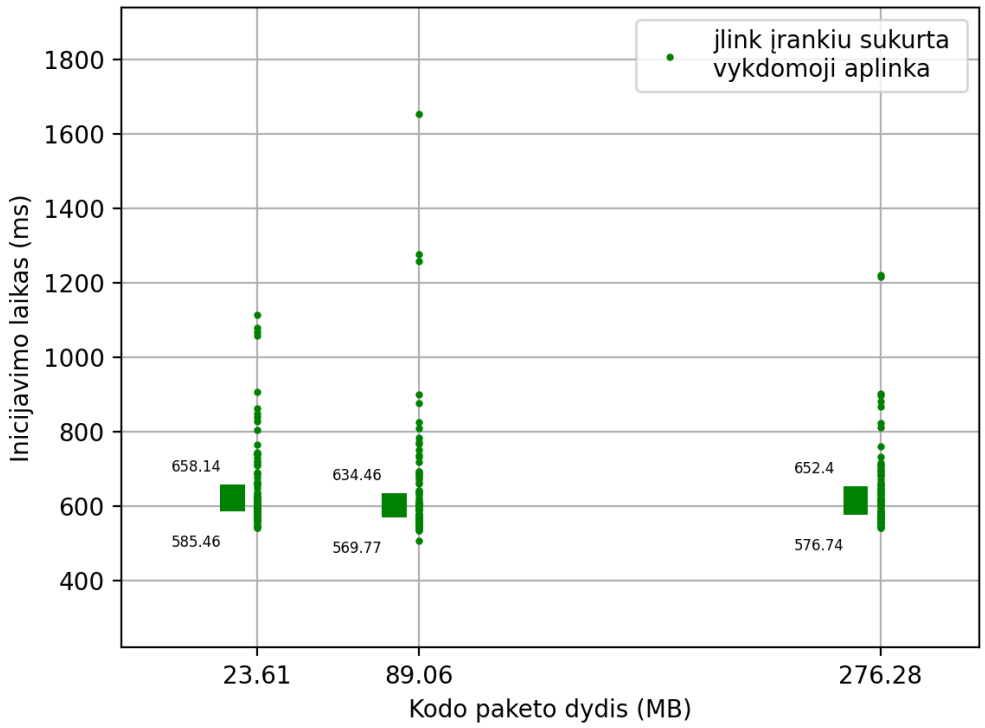
Šiuos funkcijų atvaizdus išplėsime ir sulyginsime jų dydžius tam, jog suvienodintume matavimų salygas ir galėtume atsižvelgti į būdo, kuriuo pateikiamas funkcijos programinis kodas, daromą įtaką šaltam paleidimui. Bus tikimasi pamatyti padidėjimą tarp visų funkcijų inicijavimų laikų.



10 pav. Skirtingais būdais pateikiamo funkcijos programinio kodo atvaizdu matavimai

Iš (10 pav.) rezultatų pastebime, jog inicijavimo laikas įprastu Docker atvaizdu pateikiamos funkcijos atvaizdui padidėjus nuo 195,1 MB iki 220,33 MB išaugo apie 42 %. Netikėta, jog `jlink` vykdomąją aplinka bei `native image` vykdomojo failo būdais pateiktos funkcijos padidėjus atvaizdo dydžiui ilgiau inicijuotis neužtruko, nors atvaizdai buvo išplėsti tokiu pat būdu – pripildant juos įvairiais `.jar` archyvais.

Įsitikinsime, jog `jlink` vykdomąja aplinka pateikiamo programinio kodo atvaizdo dydis neturi įtakos „konteinerio šaltam“ paleidimui matuodami skirtingų atvaizdų dydžių funkcijas, pateikiamas šiuo būdu.



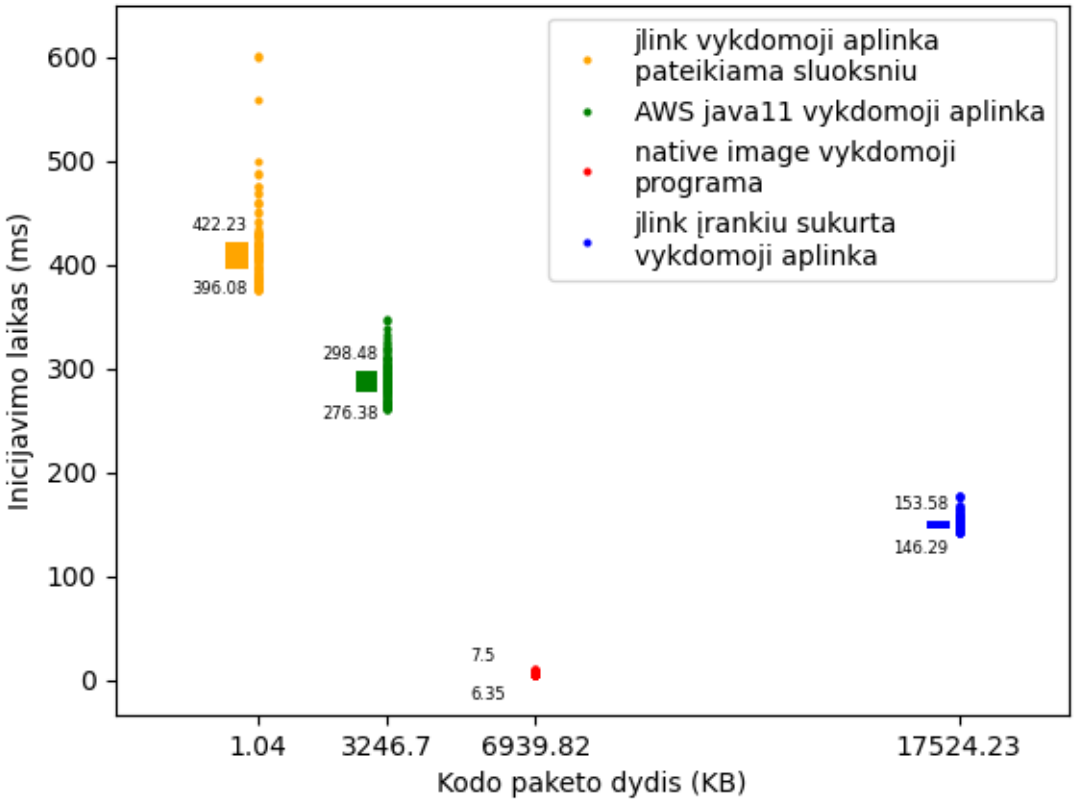
11 pav. Skirtingų atvaizdų dydžių jlink vykdymo aplinka pateikiamo programinio kodo matavimai

Iš (11 pav.) matavimų duomenų galime matyti, jog funkcijos programinį kodą pateikiant `jlink` vykdomąja aplinka funkcijos inicijavimo laikas beveik nekinta. Galime manyti, jog `native image` vykdomojo failo atvaizdu pateiko programinio kodo atvaizdo dydis taip pat nedaro įtakos funkcijos inicijavimo laikui. Sunku paaiškinti kodėl gaunami tokie rezultatai, tačiau tam gali turėti įtakos *AWS Lambda* platformos optimizacijos bei naudojama `firecracker microvm` virtualizacijos technologiją [ABI+20], tačiau tam reikėtų detalesnio tyrimo ir šios technologijos analizės.

Gauti matavimų rezultatai parodo, jog `native image` vykdomuoju failu pateikiamas funkcijos programinis kodas inicijuojasi žymiai greičiau nei kitais būdais. Taip pat konteinerio inicijavimo laikas išlieka ganėtinai pastovus, kitant atvaizdo dydžiui, ko negalima teigti apie įprastu Docker atvaizdu kartu su `.jar` vykdomuoju archyvu pateikiamo programinio kodo.

4.2. Archyvu pateikiamo kodo paketo tyrimas

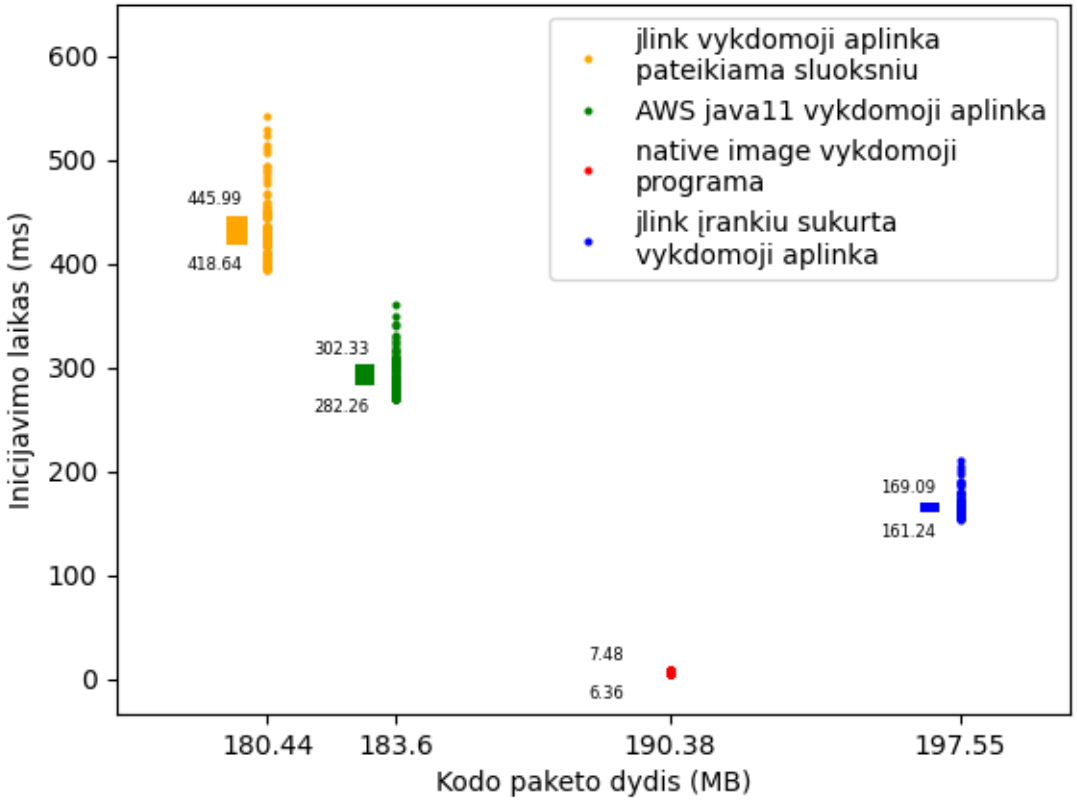
Archyvu pateikiamo kodo paketo tyrimo rezultatus pavaizduosime tokiu pačiu būdu, kaip ir praeito būdo matavime.



12 pav. Skirtingais būdais pateikiamo funkcijos programinio kodo archyvo matavimai

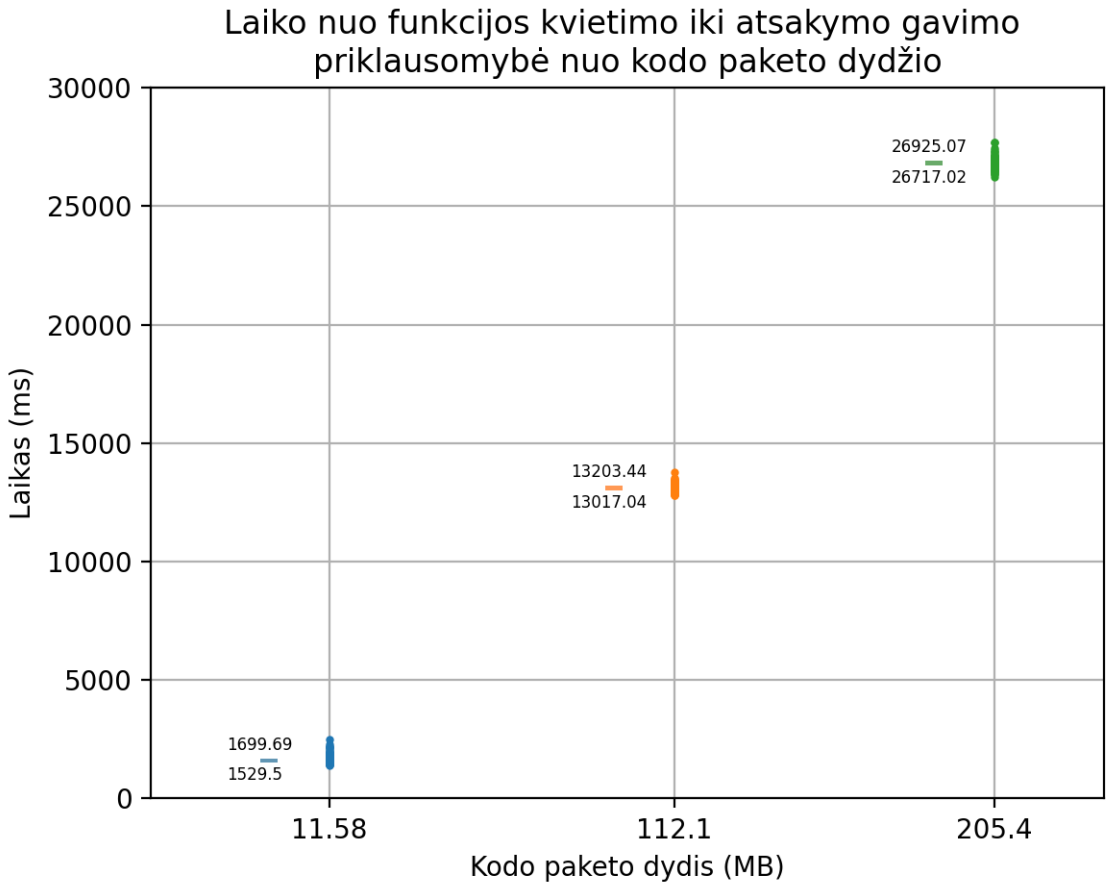
Gauti (12 pav.) duomenys parodo, jog sparčiausiai inicijavosi taip pat, kaip ir funkcijos kodo paketo atvaizdo tyrimo dalyje – *native image* vykdomoji programa. Tačiau pateikiama archyvu ji sugebėjo tai atlikti vidutiniškai vos per 6,8 ms, kai tuo tarpu antroji greičiausiai inicijavusi funkcija buvo ta, kurios programinis kodas pateikiamas archyve kartu su `jlink` įrankiu sukurta vykdomaja aplinka. Ši tai sugebėjo padaryti vidutiniškai per 150 ms – 22 kartus lėčiau (arba 2205%), nei *native image* vykdomoji programa. Įdomu, jog ta pati vykdomoji aplinka, kurta `jlink` įrankiu ir pateikta atskiru sluoksniu inicijuotis užtruko ilgiausiai. Tai leidžia manyti, jog sluoksnio prijungimas prie funkcijos vyksta jos inicijavimo metu, dėl ko atsiranda papildomas gaišties laikas.

Palyginome šiais būdais pateikiamų kodo paketų inicijavimo laiką sulygindami archyvų dydžius, norėdami patikrinti kokią įtaką archyvo dydis daro funkcijos inicijavimo laikui.



13 pav. Skirtingais būdais pateikiamo funkcijos kodo archyvo matavimai

Gauti (13 pav.) rezultatai nustebino - archyvu pateikiamas funkcijos kodo paketo dydis nedaro reikšmingos įtakos funkcijos inicijavimo laikui, kurį fiksuoja *AWS Lambda* funkcijos iškvietimo atsakyme. Šie rezultatai yra labai panašūs į auksčiau pateiktus rezultatus, kur tiriamų funkcijų archyvų dydžiai buvo minimalūs. Tačiau atliekant šiuos matavimus buvo pastebėta, jog nuo funkcijos patalpinimo bei išvietimo iki funkcijos grąžinamo rezultato praeidavo nemažas laiko tarpas, kurį taip pat galime laikyti šalto funkcijos paleidimo inicijavimo laiku. Šį laiko tarpą išmatuosime trijų skirtingų kodo paketo archyvo dydžių funkcijoms, naudojančioms *AWS* Java 11 vykdomąją aplinką ir pavaizduosime kaip šis laikas kinta priklausomai nuo kodo paketo archyvo dydžio. Kadangi ankstesniame matavime šį reiškinį pastebėjome visoms funkcijoms, galime manyti, jog šis laikas nepriklauso nuo archyvo turinio ir jam daro įtaką tik archyvo dydis.



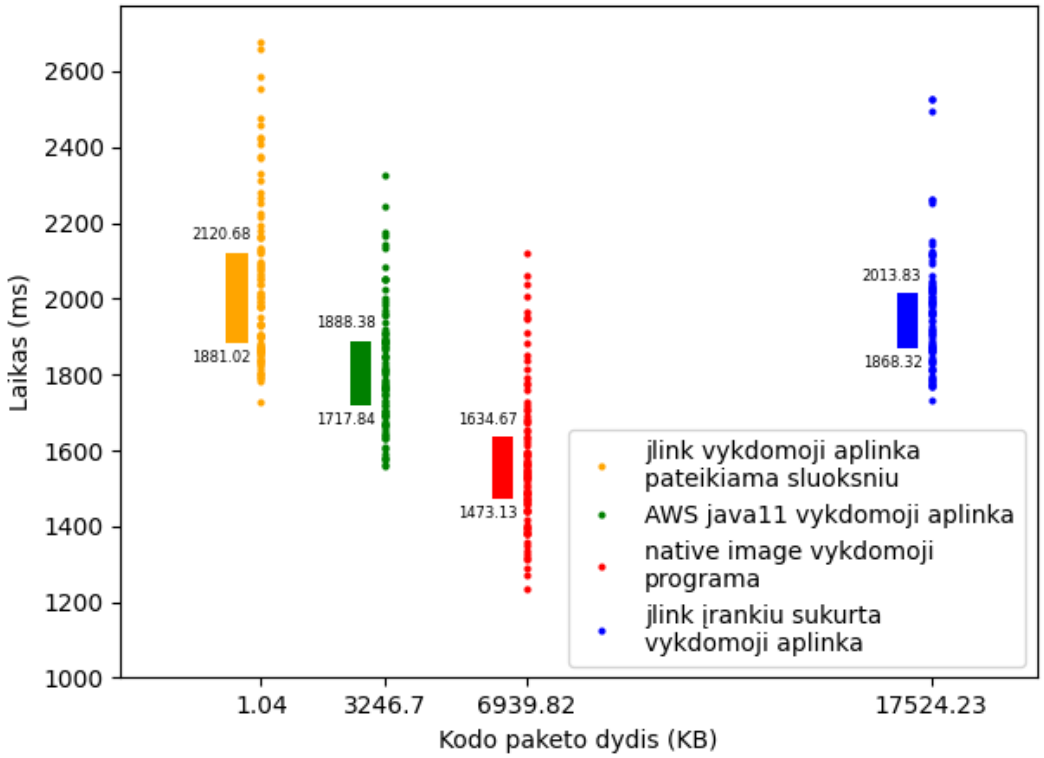
14 pav. Skirtingų funkcijos kodo paketų archyvų dydžių atsakymo laiko palyginimas

Kaip matome grafike (14 pav.), funkcija, kurios kodo paketas užima 11,58 MB, vidutiniškai nuo jos kvietimo iki atsakymo grąžinimo trunka 1614,5 ms. Funkcijos kodo paketui išaugus maždaug 100-tu MB, to paties laiko matavimas trunka 13110,2 ms – apie 8,1 karto ilgiau (arba 812%). Padidėjus apytiksliai dar 100-tu MB, šis laikas auga vidutiniškai iki 26821 ms – lyginant su 11,58 MB dydžio kodo paketo funkcija augimas yra 1661%. Panašūs funkcijos šalto paleidimo kodo paketo archyvo dydžio matavimų rezultatai pastebimi ir [MEH+18] bei [PS17]. Matome tiesinę priklausomybę tarp funkcijos kodo paketo archyvo dydžio bei inicijavimo laiko, kuris susideda iš kodo paketo atsiuntimo ir išarchyvavimo.

Panagrinėjus šį reiškinį atidžiau, tai įvyksta, kai funkcijos kodas yra pirmą kartą pateikiamas *AWS Lambda* platformai – ši tas pačias paleidžiamas funkcijas talpina toje pačioje virtualioje mašinoje tam, jog skaliuojamos funkcijos kopijos galėtų perpanaudoti tą patį – jau atsisiųsta į VM ir išarchyvuotą – programinį kodą. Šį reiškinį galime laikyti „tiekėjo šaltu“ funkcijos paleidimu [LRC+18]. Kadangi mūsų matavimo būdu kiekviena funkcija po iškvietimo yra ištrinama, kartu yra ištrinamas ir jos programinis kodas, kuris naujo funkcijos matavimo metu yra iš naujo atsiunčiamas į tam tikrą VM, todėl tokiu būdu kiekvieną kartą gauname „tiekėjo šaltą“ paleidimą, kuris realiu FaaS panaudos atveju turėtų pasitaikyti itin retai – tik tada, kai programinis kodas yra keičiamas arba funkcija yra ištrinama *AWS Lambda* platformos po ilgo laiko nepanaudojimo.

Šis kodo paketo atsiuntimo laikas nėra apmokestinamas ir į `Init Duration` matavimą neįtraukiamas, todėl tai mūsų matavimams įtakos nedaro. Mūsų matuojamą laiką galime įvardinti kaip „konteinerio šaltą“ paleidimo laiką [LRC+18]. Vadinasi, galime daryti išvadą, kad kodo paketo archyvo dydis funkcijų „konteinerio šaltiems” paleidimams įtakos nedaro – šis veiksnys pasireiškia tik „tiekėjo šaltiems“ funkcijų inicijavimams, kurie būdingi ir konteinerio atvaizdu pateikiamoms funkcijoms – įdomu pastebėti, jog matuojant funkcijas, pateikiamas konteinerio atvaizdu, nebuvo leidžiama kviesti funkcijos, kol jos atvaizdas nebuvo atsiųstas į VM – funkcijos kvietimai grąžino klaidos pranešimą ir pranešė, jog funckija kol kas yra būsenos „laukianti“ (angl. pending), kai tuo tarpu funkciją, pateikiamą archyvu – galima kviesti vos tik ją sukūrus.

Jeigu atsižvelgtumėme į šią „tiekėjo šaltą“ sąvoką, ankstensius duomenis – minimalaus kodo paketo archyvų dydžių matavimus – galėtume pavaizduoti prie funkcijos `Init Duration` laiko pridedami laiką, užtrukusį atsisiųsti ir išpakuoti archyvo failus.



15 pav. „Tiekėjo šalto“ funkcijos paleidimo laiko palyginimas

Iš (15 pav.) rezultatų galime matyti, jog `jlink` įrankiu sukurta vykdomoji aplinka, pateikiama kartu su programiniu kodu, praranda savo pranašumą, nes jos kodo paketo archyvo dydis yra išplėčiamas, dėl ko „tiekėjo šalti“ funkcijų paleidimai užtrunka ilgiau, nei, naudojant *AWS Lambda* Java 11 vykdomąją aplinką. Todėl norint išvengti ilgų „tiekėjo šaltų“ funkcijų incijavimų, reikia kiek įmanoma labiau sumažinti funkcijos kodo paketo archyvo dydį.

Matavimai parodė, jog lyginant funkcijos programinio kodo pateikimą `.zip` archyvu su prog-

raminio kodo pateikimu konteinerio atvaizdu, archyvu pateikiamas kodas yra pranašesnis ir funkcijos geba inicijuotis greičiau. Geriausi rezultatai gaunami programinį kodą sukompiliuojant į `native image` vykdomąjį failą ir pateikiant šį archyvu. `Jlink` vykdomąja aplinka pateikiamas programinis kodas taip pat paspartina funkcijos inicijavimąsi, tačiau ne taip žymiai, kaip `native image` būdu. Taip pat pamatėme, kad „tiekėjo šalti“ paleidimai stipriai priklauso nuo funkcijos kodo paketo dydžio.

4.3. Rezultatų apibendrinimas

Palyginsime abu funkcijos programinio kodo pateikimo būdus keliais aspektais. Atvaizdu, kuriuo yra pateikiama funkcija *AWS Lambda* platformai, iš esmės yra įmanoma funkciją įgyvendinti bet kokia pasirinkta programavimo kalba, pateikiant šios programavimo kalbos vykdomąją aplinką kartu su programiniu kodu, kai tuo tarpu funkcijos pateiktos archyvu programavimo kalba yra ribojama tomis kalbomis, kurias palaiko *AWS Lambda* paltforma. Dėl šios priežasties programinis kodas, supakuotos į konteinerio atvaizdą, yra lankstenis aplinkai, kurioje jis veikia, ir nepriklauso nuo platformos. FaaS paslaugų tiekėjų funkcijos kodo paketų dydžiai nėra vienodi, tačiau *AWS Lambda* platformos atveju, atvaizdu pateikiamos funkcijos kodo paketo dydis gali siekti iki 10 GB, kai tuo tarpu pateikiamas archyvu - iki 250 MB. Tačiau archyvu pateikiamos funkcijos inicijavimo laikas gaunamas spartesnis, o šio archyvo paruošimas reikalauja mažiau žinių - konteinerio atvaizdo paruošimas reikalauja papildomų žinių, pvz., Docker konteinerizacijos variklio atveju šis yra paruošiamas pagal aprašytą `Dockerfile`, kurį programuotojas turi sukonfigūruoti pats.

Savybė	Atvaizdas	Archyvas
Programavimo kalbos palaikymas	Neribojamas	Ribotas
Lankstumas ir priklausomumas nuo platformos	Lankstus, mažas	Ribotas, mažai lankstus
Leidžiamas maksimalus kodo paketo dydis	10 GB	50 MB lokalus, 250 MB S3 objektų saugyklos
Inicijavimo laikas	Lėtesnis	Spartesnis
Kodo paketo paruošimo sudėtingumas	Sudėtingesnis	Paprastesnis

1 lentelė. Programinio kodo paketo pateikiamo archyvu bei atvaizdu palyginimas

Prieš naudojantis FaaS platforma verta apsvarstyti funkcijos programinio kodo pateikimo būdus. Archyvu pateikiama funkcija yra labiau ribota ir pateikiama paprasčiau, gaunamas greitesnis inicijavimo laikas, tačiau konteinerio atvaizdu pateikiama funkcija yra lankstenė ir ši gali veikti nepriklausomai nuo platformos kurioje ji leidžiama.

Palyginsime ir JVM programinio kodo optimizavimo būdus, pritaikytus mūsų matuojamoms beserverinėms funkcijoms. `Jlink` įrankiu sukurta vykdomoji aplinka yra kuriama įtraukiant Java modulius, naudojamus pateikiamos funkcijos programinio kodo, tačiau Java moduliarumas nėra visiškai trivialus ir reikalauja žinių. `Native image` vykdomojo failo sukūrimas, lyginant su

`jlink` įrankiu kuriama vykdomaja aplinka, reikalauja daugiau žinių, kai kuriais atvejais programinio kodo pakeitimų, papildomos konfigūracijos, o kartais išvis nėra įmanomas dėl savo „uždaro pasaulio“ savybių, todėl yra sudėtingesnis jo panaudojimas. `Native image` yra mašininio kodo rinkinys, todėl natūralu, kad šis kodas yra priklausomas nuo procesoriaus architektūros, kai `jlink` yra vykdomoji aplinka, vykdanti Java baitinę programa (angl. bytecode). Tačiau mašininiam kodui interpretuoti nėra reikalinga Java virtuali mašina, todėl `native image` vykdomoji programa veikia daug efektyviau ir beserverinė funkcija geba inicijuotis daug sparčiau.

Savybė	Jlink	Native image
Panaudojimo galimybė	Lanksti	Ribota, kartais neįmanoma
Panaudojimo sudėtingumas	Vidutinis	Sudėtingas
Priklausomumas nuo platformos	Vidutinis	Didelis
Inicijavimo laikas	Lėtesnis	Spartesnis

2 lentelė. Jlink įrankio bei native image vykdomosios programos sukūrimo palyginimas

Jeigu beserverinei funkcijai yra ypatingai svarbus inicijavimo laikas, verta panagrinėti galimybes sukompiliuoti programinį kodą į `native image` vykdomąją programą, tačiau yra galimybė, kad to padaryti nepavyks be didesnių kodo ar pasirinktų bibliotekų pakeitimų. `Jlink` įrankiu sukurta vykdomoji aplinka, kaip matėme matavimuose, rodo spartesnius inicijavimo laiko rezultatus, nei įprasta FaaS platfomos siūloma vykdomoji aplinka, tad šiuo būdu lengviau sutrumpinti funkcijos inicijavimo laiką.

4.4. Veiksnių „šaltam“ funkcijos paleidimui daromos įtakos minimizavimas

Funkcijos inicijavimo laikas yra glaudžiai susijęs su atvaizdo, kuriuo pateikiamas funkcijos programinis kodas, dydžiu, todėl yra itin svarbu šį dydį minimizuoti [And21a] - atvaizdo kūrimui rinktis minimalaus dydžio bazinį paketą, į funkcijos atvaizdą nepakuoti nenaudojamų failų, eliminuoti nenaudojamą programinį kodą, rinktis mažiau vietos užimančias funkcijos naudojamas išorines bibliotekas. Taip pat pateikiant kodą virtualaus konteinerio atvaizdo būdu, funkcijos vykdomoji aplinka turi būti supakuojama kartu su programiniu kodu, kas stipriai išplėčia atvaizdo dydį, ypatingai JVM programavimo kalbų atveju - Java vykdomoji aplinka gali sudaryti didžiąją dalį atvaizdo dydžio, priklausomai nuo to, kiek ir kokių išorinių bibliotekų platformos vartotojo funkcija naudoja. Kuriant funkcijos konteinerio atvazidą, verta naudoti kelių fazių kūrimo procesą, kurio metu pradinėje fazėje funkcijos programinis kodas yra paruošiamas paleisti ir naudojami kūrimo įrankiai, o sekančioje fazėje šis paruoštas paleidimui programinis kodas yra nukopijuojamas ir naudojamos tik vykdomosios aplinkos priklausomybės taip sumažinant atvaizdo dydį. Java įrankio `jlink` bei `GraalVM` vykdomosios aplinkos pagalba yra įmanoma minimizuoti konteinerio atvaizdo dydį o taip pat paspartinti funkcijos inicijavimo laiką, tačiau tam reikalingos papildomos žinios.

Gauti rezultatai ir išvados

Rezultatai

1. Praplėstas įrankis, naudojamas funkcijų matavimams atlikti.
2. Išmatuota funkcijos šalto paleidimo laiko priklausomybė nuo programinio kodo pateikimo būdo:
    - kodo paketo pateikiamo archyvu;
    - kodo paketo pateikiamo konteinerio atvaizdu.
3. Nustatyi du būdai, optimizuojantys JVM funkcijos šalto paleidimo laiką, ir išmatuota jų įtaka:
    - `jlink` įrankiu sukuriama vykdomoji aplinka;
    - `native image` vykdomasis failas, sukuriamas išankstinio programinio kodo kompiliavimo pagalba.

Išvados

1. Funkcijos, kurios programinis kodas pateikiamas archyvu, šalto paleidimo laikas yra mažesnis nei funkcijos, pateikiamos konteinerio atvaizdu.
2. JVM programavimo kalbos beserverinės funkcijos inicijavimo laiką galima sumažinti dešimtimis kartų panaudojus sukompiliuotą `native image` vykdomąjį failą ir šį pateikus archyvu. Šis optimizavimo būdas ne visada yra įmanomas, galintis reikalauti programinio kodo pakeitimų.
3. `Jlink` įrankiu sukurta vykdomoji aplinka paspartina funkcijos šaltą paleidimą maždaug dvigubai, šis įrankis nereikalauja programinio kodo pakeitimų, tačiau šiuo būdu nėra gaunamas sparčiausias funkcijos šalto paleidimo laikas ir jį verta rinktis tada, kai `native image` funkcijos kompiliavimo būdas nėra optimalus.
4. „Tiekėjo šaltam“ paleidimui ypatingai didelę įtaką daro funkcijos programinio kodo paketo dydis, kai tuo tarpu „konteinerio šaltam“ paleidimui didesnę įtaką turi funkcijos vykdomoji aplinka ir vykdymo būdas

Galimos tolesnių tyrimų kryptys

Gauti rezultatai parodė, kad JVM funkcijos šalta paleidimą galima sumažinti dešimtimis kartų naudojant `native image` vykdomąjį failą, todėl verta panagrinėti `GraalVM` vykdomosios aplinkos galimybes bei ribas kuriant šį vykdomąjį failą tam, jog būtų galima lengviau nuspręsti ar funkciją, kurią norima optimizuoti šiuo būdu, yra įmanoma sukompiliuoti į `native image` ir jeigu taip, tai kiek daug pastangų - programinio kodo pakeitimų, konfigūracijos, skirtingų bibliotekų panaudojimo - tam prireiktų. Taip pat naudinga paanalizuoti `firecracker microvm` virtualizacijos technologiją, naudojamą *AWS Lambda* platformos įgyvendinant funkcijų konteinerizaciją [ABI+20], atsižvelgti į šios technologijos veikimo principus siekiant sumažinti funkcijos šaltą paleidimo laiką. Verta patyrinėti AppCDS (angl. Application Class Data Sharing) principą, kuris leidžia dalintis tomis pačiomis į atmintį užkrautomis Java klasėmis skirtingoms Java virtualioms mašinoms, esančioms toje pačioje VM, daromą įtaką didėsnės apimties JVM funkcijų šaltiems paleidimams. Taip pat aktualu patyrinėti alternatyvių įprastiems JVM funkcijų karkasų, labiau pritaikytų debesų kompiuterijos infrastruktūrai, pasižyminčių greitu programos paleidimu bei mažu atminties suvartojimu, tokių kaip `Quarkus` ar `Micronaut`, kuriais naudojantis programos taip pat turi palaikymą būti sukompiliuotos į *native image* vykdomąjį failą, daromą įtaką funkcijos šaltiems paleidimams.

Summary

This paper overviews FaaS execution principles, shortcomings and advantages of such platforms, raises and describes the problem of function cold start and identifies factors for which this problem occurs. By measuring functions in *Amazon Web Services Lambda* FaaS platform and collecting results it suggests ways to optimize JVM function cold start time.

Literatūra

[ABI+20] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka ir Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, p.p. 419–434, Santa Clara, CA. USENIX Association, 2020-02. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[And21a] Darius Andzevičius. Beserverinių funkcijų šalto paleidimo laiko tyrimas. Projektinis darbas, 2021-05.

[And21b] Darius Andzevičius. Serverless Benchmark. https://github.com/dariusandz/serverless-benchmark, 2021.

[AWS20] AWS. AWS Lambda now supports up to 10 GB of memory and 6 vCPU cores for Lambda Functions. https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions, 2020. Tikrintas 2021-03-27.

[AWS21a] AWS. AWS Lambda execution environment. https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html, 2021. Tikrintas 2021-04-28.

[AWS21b] AWS. AWS Lambda FAQs. https://aws.amazon.com/lambda/faqs/, 2021. Tikrintas 2021-04-05.

[AWS21c] AWS. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/, 2021. Tikrintas 2021-03-27.

[AWS21d] AWS. Configuring Lambda function memory. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html, 2021. Tikrinta 2021-05-04.

[AWS21e] AWS. Lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, 2021. Tikrinta 2021-05-07.

[BCC+17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, et al. *Research Advances in Cloud Computing*. Springer, Singapore, 2017. 1-20.

[EUG+21] Unai Elordi, Luis Unzueta, Jon Goenetxea, Estíbaliz Loyo, Ignacio Arganda-Carreras ir Oihana Otaegui. On-demand Serverless Video Surveillance with Optimal Deployment of Deep Neural Networks, 2021.

[Goo20] Google. Quotas. https://cloud.google.com/functions/quotas, 2020. Tikrintas 2021-03-28.

[GRE+12] Joel Gibson, Robin Rondeau, Darren Eveleigh, and Qing Tan. Benefits and challenges of three cloud computing service models. In *International Conference on Computational Aspects of Social Networks, CASON*, pp. 198–205, Sao Carlos, Brazil. IEEE, 2012.

[Hai18] Steven Haines. Serverless computing with AWS Lambda, Part 1. https://www.infoworld.com/article/3210726/serverless-computing-with-aws-lambda.html, 2018. Tikrinta 2021-04-05.

[IBM20] IBM. OpenWhisk common use cases. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-use_cases, 2020. Tikrintas 2021-03-13.

[KSS10] Ali Khajeh-Hosseini, Ian Sommerville, and Ilango Sriram. Research challenges for enterprise cloud computing, 2010.

[Lin14] Jeremy Lindblom. AWS re:Invent 2014. https://aws.amazon.com/blogs/developer/aws-reinvent-2014/, 2014. Tikrintas 2021-03-13.

[LRC+18] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless computing: an investigation of factors influencing microservice performance. In *IEEE International Conference on Cloud Engineering (IC2E)*, pp. 159–169, Orlando, FL, USA. IEEE, 2018.

[McK16] John McKim. Abstracting the Back-end with FaaS. https://serverless.zone/abstracting-the-back-end-with-faas-e5e80e837362, 2016. Tikrinta 2021-04-03.

[MEH+18] Johannes Manner, Martin Endreß, Tobias Heckel ir Guido Wirtz. Cold Start Influencing Factors in Function as a Service. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, p.p. 181–188, 2018. DOI: 10.1109/UCC-Companion.2018.00054.

[Mic21] Microsoft. Multicloud solutions with the Serverless Framework. https://docs.microsoft.com/en-us/azure/architecture/example-scenario/serverless/serverless-multicloud, 2021. Tikrinta 2021-04-05.

[Mli21] Kimberly Mlitz. Cloud services market spending by segment worldwide from 2015 to 2020 (in billion U.S. dollars). https://www.statista.com/statistics/540499/worldwide-cloud-computing-revenue-by-segment/, 2021. Tikrintas 2021-03-13.

[PS17] Hussachai Puripunpinyo ir M.H. Samadzadeh. Effect of optimizing Java deployment artifacts on AWS Lambda. *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, p.p. 438–443, 2017. DOI: 10.1109/INFCOMW.2017.8116416.

[Rob20] Mike Roberts. Analyzing Cold Start latency of AWS Lambda. https://blog.symphonia.io/posts/2020-06-30_analyzing_cold_start_latency_of_aws_lambda, 2020. Tikrinta 2021-05-07.

[Wan21] Sutao Wang. *Thin Serverless Functions with GraalVM Native Image*. Magistrinis darbas, ETH Zurich, 2021-04-22.

[Woo20] Julian Wood. Working with Lambda layers and extensions in container images. `https://aws.amazon.com/blogs/compute/working-with-lambda-layers-and-extensions-in-container-images/`, 2020. Tikrinta 2021-05-17.

[WSH+19] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. In *Proc. ACM Program. Lang.* P. 29, New York, NY, USA. Association for Computing Machinery, 2019.