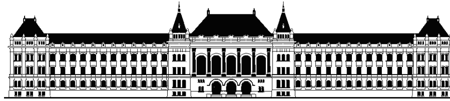


Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Kubernetes és Gardener AutoScaling funkciójának teljesítményanalízise

Készítették
Nagy Ádám Zsolt
Fodor Balázs

Konzulensek
Németh Balázs
Dr. Sonkoly Balázs

Tudományos Diákköri Konferencia

2018. október 26.

Tartalomjegyzék

Kivonat **4**

Abstract **6**

1. Bevezetés **8**

1.1. Számítási felhők 8

1.2. Microservice architektúra 9

1.3. Kubernetes, disztribúciók, telepítők 9

2. Kapcsolódó munkák **11**

2.1. Felhő szolgáltatás típusok 11

2.2. Felhő platformok teljesítmény mérése 13

3. Kubernetes **15**

3.1. Architektúra 15

3.1.1. Master csomópont 15

3.1.2. Worker csomópont 16

3.2. Kubernetes objektumok 17

3.3. Ön-gyógyulás, automatikus skálázás 18

3.4. Gardener 19

3.5. Motiváció 20

4. Mérési környezet **21**

4.1. Környezetek kialakítása 22

4.2. Teszt alkalmazás 22

4.3. Mérések 23

5. Eredmények **27**

5.1. Lineáris mérések 27

5.2. Exponenciális mérések 33

5.3. „Dombszerű” mérések 39

5.4. Egységugrás mérések 45

6. Konklúzió **48**

Irodalomjegyzék **51**

Köszönetnyilvánítás

Szeretnénk megköszönni Németh Balázsnak és Dr. Sonkoly Balázsnak a munkánk során nyújtott szakmai támogatásukat és iránymutatásukat. Közös beszélgetéseink során észrevételeikkel és ötleteikkel hozzájárultak a munkánk folyamán felmerült nehézségeink leküzdéséhez és szakmai fejlődésünkhöz.

Köszönjük továbbá a HSN Laboratóriumnak, hogy a mérések elvégzéséhez szükséges infrastruktúrát biztosította számunkra.

Kivonat

Napjainkban a vállalatok egyre jobban kihasználják a cloud adta lehetőségeket. Ha egy vállalat nem rendelkezik saját erőforrásokkal, akkor jobban megéri egy publikus cloud szolgáltatást igénybe venni, mint saját szervereket vásárolni, mivel így elég csak a ténylegesen szükséges erőforrásért fizetni és az üzemeltetést is a szolgáltató végzi helyettük. Ha rendelkezik szerverekkel, infrastruktúrával, akkor is alkalmazhatnak valamilyen, akár nyílt forráskódú, privát cloud megoldást, mivel a virtualizáció segítségével elrejti előlük a fizikai infrastruktúrát, így könnyítve meg a fejlesztést.

A Kubernetes egy Google által fejlesztett és 2014-től nyílt forráskódú konténer orkesztrációs platform. Segíti konténerizált alkalmazások fejlesztését, telepítését, skálázását, menedzselését, valamint ezen folyamatok automatizálását. Nagyon elterjedt platform, ami funkciói mellett annak is köszönhető,hogy a korábbi monolitikus architektúra helyett a komponensekre bontott alkalmazások fejlesztését támogatja. Ezáltal az alkalmazások skálázhatóbbá és robusztusabbá válnak, a komponenseik lecserélhetőek, újrahasznosíthatóak.

A Kubernetes rendszer egyik legkedvezőbb tulajdonsága az AutoScaling. Ez azt jelenti, hogy a terheléstől függően fel- vagy leskálázza az alkalmazást, vagyis több vagy kevesebb konténert indít és tart a rendszerben. Így az alkalmazást kiszolgáló erőforrás mennyisége a terheléstől függően dinamikusan változik.

Sok projekt foglalkozik azzal, hogy a Kubernetes rendszert kibővítse, új funkciókkal ruházza fel, vagy magát a telepítést és menedzselést segítse. A Gardener egy olyan rendszer, ami Kubernetes clusterek menedzselését és automatizált létrehozását valósítja meg, vagyis Kubernetes Clusters as a Service-t nyújt. A natív Kubernetesszel való összehasonlításhoz azért választottuk a Gardenert, mert az általa létrehozott felhasználói clustereknek az architektúrája eltér a standard felépítéstől. A rendszer alapja a Seed Cluster, mely egy admin clusternek tekinthető, ami a felhasználóknak allokált clusterek (Shoot Cluster) kontroll síkját tartalmazza. A standard Kubernetes clusterben, a Shoot Clusterhez képest az a különbség, hogy a központi vezérlő elem a clusteren belül található. Munkánk során többek között azt vizsgáljuk, hogy ennek a Gardener-féle cluster felépítésnek milyen hatása van a Kubernetes nyújtotta funkciókra, különös tekintettel az AutoScalingre.

Dolgozatunkban bemutatott eredmények alapjai olyan mérések, melyek jól jellemzik egy valós rendszer terhelését különböző helyzetekben. A tesztek közben vizsgáljuk a Kubernetes AutoScaling gyorsaságát és robusztusságát. Ehhez kérés sorozatokat küldünk egy változtatható erőforrás igényű szervernek és közben monitorozzuk az érzékelhető késleltetésre gyakorolt hatását. Ezáltal meg tudjuk vizsgálni, hogy a Kubernetes clusterbe telepített alkalmazások, különböző terhelések mellett, milyen felhasználói élményt nyújtanak és ezt befolyásolja-e egy olyan architektúra-módosítás, mint amilyen a Gardener esetében történt.

Abstract

Today companies increasingly take advantages of the opportunities of cloud platforms. If a company does not have resources itself, it worth to use the services of a public cloud provider, rather than buying expensive servers, because this way they only have to pay for the resources actually being used and the operation is also carried out by the provider. Even though a company has servers and infrastructure, a private cloud solution, even an open-source one, can be used, because it hides the physical infrastructure with the help of virtualization technologies, which makes the development easier.

Kubernetes is a container orchestration platform developed by Google, and was open-sourced in 2014. It supports the development, deployment, scaling and management of containerized applications and the automation of these processes. Kubernetes is a widespread platform, which, in addition to its functions, is due to the fact that instead of the previous monolithic architecture, it supports the development of multi component applications (microservices). Thereby the applications become more scalable and robust, the components are replaceable and reusable.

One of the most favorable feature of Kubernetes is AutoScaling. This means that it scales the application up or down depending on load, i.e., it starts and holds more or fewer containers in the system. So, the amount of resources that serve the application change dynamically with the load.

Many projects address the expansion of Kubernetes, adding new features or assisting installation and management. Gardener is a system that realizes management and automated deployment of Kubernetes clusters, so it provides Kubernetes Clusters as a Service. For the comparison with the native Kubernetes we choose Gardener, because the architecture of the user cluster provided by Gardener is different from the standard structure. The base of the system is a Seed Cluster, which can be considered as an admin cluster, that contains the control plane of the clusters allocated to users (Shoot Cluster). In the standard Kubernetes cluster, compared to the Shoot Cluster, the difference is, that the central controller element is located inside the cluster. During our work, among others, we examined what kind of effects has the structure of Gardener clusters on the functions and features of Kubernetes,

especially on AutoScaling.

The basis of the results, presented in our paper, are measurements, that represent the load of a real life system in different situations. During the tests we analyse the speed and robustness of the AutoScaling feature of Kubernetes. For this purpose we send request sequences to a server, whose resource demand is variable, and in the meantime we monitor its impact on the perceptible delay. This way we are able to examine the user experience of applications deployed in a Kubernetes cluster during different loads and if it is influenced by such an architecture change like in the case of Gardener.

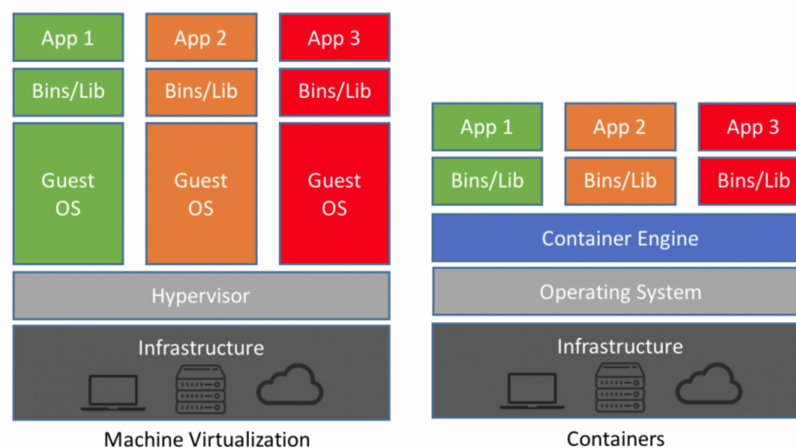
1. fejezet

Bevezetés

1.1. Számítási felhők

A számítási felhők (cloud) igény szerinti hozzáférést biztosítanak számítási erőforrásokhoz (hálózatokhoz, szerverekhez, alkalmazásokhoz, szolgáltatásokhoz, tárhelyekhez). Általában valamilyen virtualizációs technológiát alkalmaznak annak érdekében, hogy a felhasználó az általa meghatározott erőforráshoz jusson [1].

A virtualizáció egy rendszer szintű absztrakció. Elrejti a fizikai hardware tulajdonságait, a felhasználó a virtualizációs réteget használva éri el a számítógép erőforrásait. Többféle virtualizációs technológia létezik, az egyik ilyen a virtuális gép (virtual machine - VM). Ebben az esetben az egyes végrehajtási környezeteknek saját operációs rendszerük van, a gazda gép (host) hardver erőforrásai virtualizáltak, ezeket érik el a rajta futó virtuális gépek. A másik elterjedt technológia a konténer alapú virtualizáció, ahol a host gép kernelén osztoznak a végrehajtási környezetek.



1.1. ábra. *Virtuális gépek és konténerek közötti különbségek*[1]

[1]`https://blog.netapp.com/blogs/containers-vs-vms/`

Az 1.1 ábrán látható a fő különbség a két technológia között. Míg a VM-ek saját operációs rendszerrel is rendelkeznek, és jobban el vannak szeparálva mind a host géptől, mind a többi virtuális géptől, addig a konténerek közös operációs rendszert használnak. Mivel nincs szükség saját OS-re, ezért indításuk gyorsabb, azonban kevésbé vannak elszeparálva egymástól, ami biztonsági kockázatot jelenthet.

A virtualizáció előnye, hogy elrejti a valódi fizikai hardvert, és akár több virtuális környezet is futtatható ugyanazon a fizikai gépen. Ezzel a megoldással hatékonyan használhatók ki a fizikai erőforrások, ami elengedhetetlen a felhő rendszerek effektív működéséhez.

A számítási felhőknek több típusa is van attól függően, hogy ki tartja karban és kiknek enged hozzáférést. Ez alapján megkülönböztetünk publikus cloudot, amit tetszőlegesen bárki igénybe vehet, általában egy cloud szolgáltató tarja karban. A privát cloud általában egy szervezet belső használatú rendszere, és a tagjainak enged hozzáférést. Valamint ezen két cloud típus kombinációja is előfordul, ezt nevezzük hibrid cloudnak.

1.2. Microservice architektúra

Manapság nagyon elterjedt alkalmazás fejlesztési megközelítés a microservice architektúra. Ennek a módszernek az a lényege, hogy egy összetett alkalmazást különálló, külön üzemeltethető komponensekből építenek fel, a komponensek valamilyen jól definiált hálózati interfészen keresztül kommunikálnak egymással [2].

Ez a megközelítés ellentétben áll a korábban sok helyen használt monolitikus fejlesztéssel. Ebben az esetben az alkalmazás egy komponensből áll, nem bontható különálló egységekre. Ez megnehezíti az alkalmazás skálázását, karbantartását és telepítését, ezzel szemben a microservice architektúra moduláris, ezért támogatja a komponensek egymástól független fejlesztését, könnyebb skálázhatóságát, az újrahasznosíthatóságot és a komponensek cserélhetőségét.

A konténerizációra épülő platformok támogatják a microservice architektúra szerinti fejlesztést, ugyanis lehetővé teszik az alkalmazás komponensek szeparálását konténerek segítségével. Ilyen konténerizációra épülő platform a Kubernetes is.

1.3. Kubernetes, disztribúciók, telepítők

A Kubernetes egy kezdetben Google által fejlesztett, majd 2014-től nyílt forráskódú, konténer orkesztrációs platform. Széles körben elterjedt, ami, többek között, annak köszönhető, hogy támogatja a microservice architektúra alapú fejlesztést, alkalmazások telepítését, futtatását és menedzsmentjét. A Kubernetes rendszer olyan komponensekkel rendelkezik, melyek segítik az alkalmazás automatikus skálázását

(AutoScaling) és hiba utáni helyreállását (Self-Healing) [3].

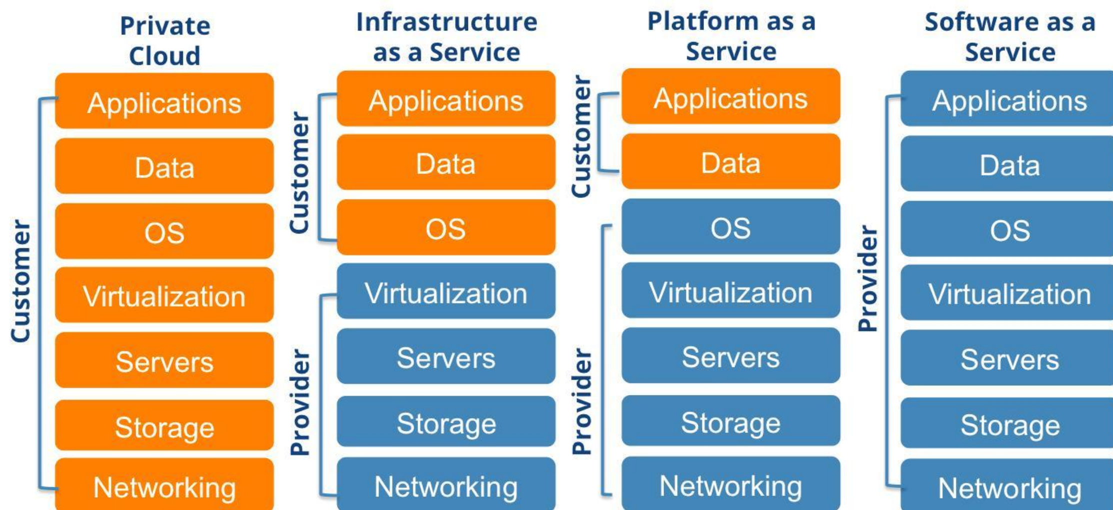
Több Kubernetes-re épülő disztribúció is készül, melyek az alap funkciókat egészítik ki vagy azok kezelését segítik. Ezenkívül számos telepítő (installer) is készült a rendszerhez, melyek Kubernetes klaszterek telepítését könnyítik meg. A Gardener egy installer, különlegessége, hogy az alap Kubernetes architektúrától eltérően, ahol a vezérlő sík a klaszter központi csomópontján található, itt egy külön központi Kubernetes klaszterbe van kiszervezve a felhasználói rendszerek vezérlése. A különbségekkel részletesen a 3 fejezet foglalkozik.

2. fejezet

Kapcsolódó munkák

2.1. Felhő szolgáltatás típusok

A felhő szolgáltatásokat (Cloud Services) hagyományosan három nagy szolgáltatási modellbe szokás sorolni. Ezek az infrastruktúra szolgáltatás (Infrastructure as a Service, röviden IaaS), platform szolgáltatás (Platform as a Service, röviden PaaS) és a szoftver szolgáltatás (Software as a Service, röviden SaaS)[1].



2.1. ábra. *Főbb felhő szolgáltatási modellek közötti különbségek*[1]

A legalapvetőbb felhő szolgáltatás kategória az infrastruktúra szolgáltatás. Ahogy a 2.1 ábra második oszlopában is látható, ebben a modellben a szolgáltató számítási, tárolási, hálózati és egyéb alapvető erőforrásokat biztosít a felhasználónak, melyen alkalmazásokat tud telepíteni és futtatni, beleértve az operációs rendszereket is. A felhasználó egy virtualizált erőforrás halmazt kap ezért nem kell az alatta lévő fizikai infrastruktúrával foglalkozzon, de az operációs rendszerek, tárolás, telepített alkal-

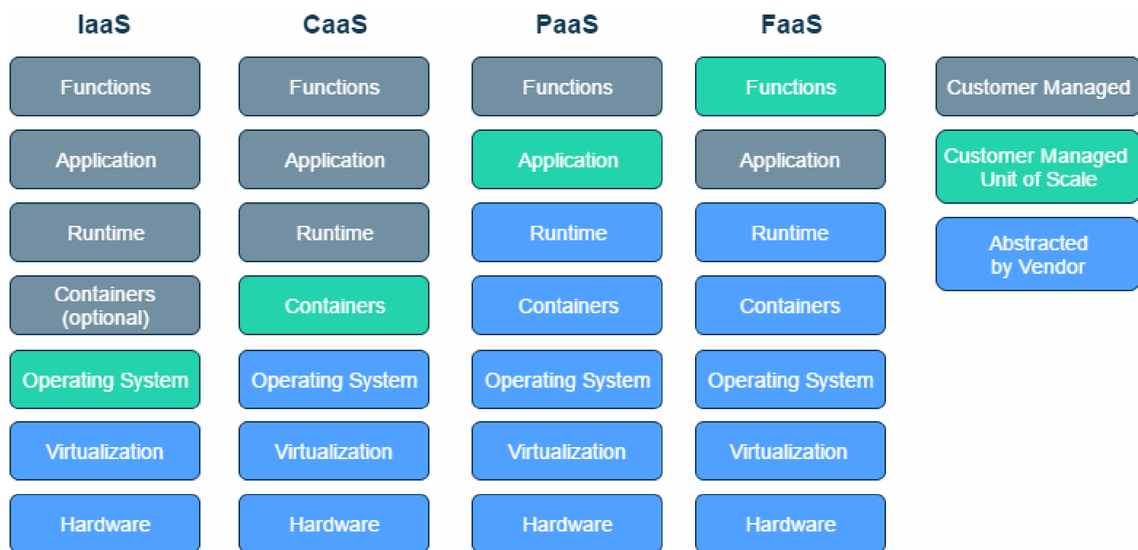
[1] `https://www.sevone.com/white-paper/monitoring-cloud-infrastructure-performance-eliminate-visibility-gaps`

mazások és bizonyos hálózati elemek menedzselése továbbra is az ő feladata. Ilyen szolgáltatást nyújt például az Amazon Web Services (AWS) vagy az OpenStack[4].

A következő modell a platform szolgáltatás. Itt a szolgáltató egy olyan környezetet ad a felhasználó számára melyben alkalmazásokat tud fejleszteni és futtatni. Ezek a platformok általában több programozási nyelvvel és könyvtárral, különféle eszközökkel, szolgáltatásokkal támogatják a fejlesztést. A felhasználó nem kezeli az operációs rendszert, sem a hálózati és tárolási erőforrásokat, viszont az alkalmazás és az alkalmazást futtató környezet konfigurálása az ő feladata marad. Egyik legismertebb ilyen szolgáltatás a Google App Engine.

A felhő szolgáltatások jelentős részét a szoftver szolgáltatások teszik ki. Itt a szolgáltató valamilyen alkalmazáshoz nyújt hozzáférést a felhasználó számára, melyet legtöbbször böngészőn keresztül lehet elérni. Ebbe a kategóriába tartozik, többek között, a Google Apps alkalmazásai, a Dropbox és a Prezi.

A felhő számítás térhódításával egyre több szolgáltatás jelent meg, melyek nem sorolhatóak egyértelműen a fenti kategóriák egyikébe sem. Ennek köszönhető a CaaS és a FaaS, vagyis a konténer szolgáltatás (Container as a Service) és a függvény szolgáltatás (Function as a Service), fogalmak megszületése. A 2.2 ábrán látható, hogy a skálázási alapegység alapján a CaaS az IaaS és a PaaS között helyezkedik el, míg a FaaS a legmagasabb szintű absztrakciót biztosítja.



2.2. ábra. *Szolgáltatás modellek a skálázási alapegységük alapján*[2]

A microservice architektúrájú alkalmazásfejlesztés elterjedésével kézenfekvő volt, hogy a konténer alapú alkalmazásokat is valamilyen felhő környezetben lehessen fejleszteni és futtatni. A szolgáltató itt egy olyan környezetet nyújt, melyben a konténerek kezelését a rendszer végzi [5]. A felhasználónak csak magas szintű utasításokat

[2] https://serverless.zone/abstracting-the-back-end-with-faas-e5e80e837362

kell kiadni, a többit elvégzi a rendszer. Például ha azt szeretnénk, hogy az alkalmazásunk alkalmazkodjon a forgalom növekedéséhez elég csak az automatikus skálázást konfigurálnunk, ezután a rendszer a terhelés függvényében növeli vagy csökkenti a konténerek számát. A CaaS platformok egyik legfontosabb képviselője a Kubernetes.

Egy másik új és érdekes megközelítés a függvény szolgáltatás. A konténer szolgáltatáshoz képest egy szinttel magasabb az absztrakció, mivel itt a felhasználóknak nem teljes alkalmazásokat kell fejleszteniük, hanem olyan rövid futási idejű, állapotmentes függvényeket melyek bizonyos események hatására hívódnak meg. Ez a megoldás azért előnyös, mert a fejlesztőknek még konténerek létrehozásával sem kell törődjenek, tisztán a funkció implementálására koncentrálhatnak. A függvényt a környezet a programozási nyelvnek megfelelő konténerben futtatja, majd ha az lefutott leáll a konténer is. Ebből adódik a másik előnye a konténer szolgáltatásokkal szemben: a felhasználó csak a tényleges futás közbeni erőforrás használatért fizet [6]. Az egyik legismertebb ilyen kommerciális megoldás az AWS Lambda, az open-source vonalon pedig a Kubernetes-re épülő Kubeless.

2.2. Felhő platformok teljesítmény mérése

A felhő platformoknál az fizikai erőforrások minél jobb kihasználása kulcsfontosságú a teljesítmény szempontjából. Az 1.1 fejezetben bemutatott konténerek és virtuális gépek közötti különbségek mélyebb elemzését mutatja be a [7] szerzője. Munkájában arra a következtetésre jut, hogy mind teljesítmény, mind skálázhatóság szempontjából előnyösebb a konténerek használata, habár hátrányként említi az izoláció problémáját a virtuális gépekhez képest.

A microservice architektúrájú fejlesztés elterjedésével egyre fontosabbá vált, hogy az ezeket támogató platformok, mint a Kubernetes, teljesítménye hatékonyan monitorozható legyen. A [8] cikkben a microservice-ek teljesítmény monitorozásának és modellezésének kérdését vizsgálják. A szerzők arra a következtetésre jutottak, hogy a hagyományos monitorozási eszközök mellett olyanokra is szükség van, melyek képesek a konténer alapú, rugalmas, skálázható alkalmazásokat is hatékonyan monitorozni. Változást figyeltek meg a performancia modellezés felhasználási területében is. Míg hagyományosan a tervezési idejű modellezésnél a kapacitás tervezésen volt a hangsúly, a microservice-ek esetében a skálázhatóság és a rugalmasság miatt a költség irányítás és a megbízhatóság, futási idejű adaptációs képesség tervezése a fő szempont.

A [9] szerzői egy Kubernetes pod és konténer életciklus modell mutatnak be. Ez a modell teljesítmény és erőforrás tervezésnél, valamint alkalmazások tervezésénél és strukturálásánál nyújt segítséget.

Munkánkhoz szorosabban kapcsolódik A Kubernetes skálázását vizsgáló [10]. A cikkben a szerzők a Kubernetes CPU kihasználtságon alapuló skálázódási algoritmusát vizsgálták és hasonlították össze teljesítményét az általuk bemutatott algoritmusával.

Dolgozatunkban szintén a Kubernetes CPU alapú skálázódását vizsgáljuk, viszont fő célunk, hogy a felhasználói élményre gyakorolt hatását vizsgáljuk és kiderítsük, hogy egy olyan architekturális változtatás a rendszeren, melyet a Gardener végzett, hogyan befolyásolja ezt.

3. fejezet

Kubernetes

A Kubernetes (röviden k8s) jelenleg a vezető konténer orkesztrációs platform. Számos rendszer épül az alapjaira és a fejlesztő cégek is egyre gyakrabban térnek át erre vagy ezen alapuló rendszerre. Ennek oka a gyors, komponensekre osztott alkalmazások fejlesztésének lehetősége, valamint az alkalmazások robusztusságának és skálázhatóságának biztosítása.

3.1. Architektúra

A Kubernetes rendszer magas szintű architektúrája látható a 3.1 ábrán. Egy Kubernetes klaszter kétféle csomópontot tartalmaz: a Master, mely a vezérlési síkot tartalmazza, valamint a Worker, ami pedig konténerek futtatásáért felelős. A klaszter vezérlése a Master csomóponton keresztül történik, részletesen a 3.1.1 alfejezet tárgyalja. A Worker csomópontokon futnak a Kubernetes objektumok, vezérlésüket a Master végzi. A 3.1.2 alfejezet tartalmazza a Worker csomópontok felépítését.

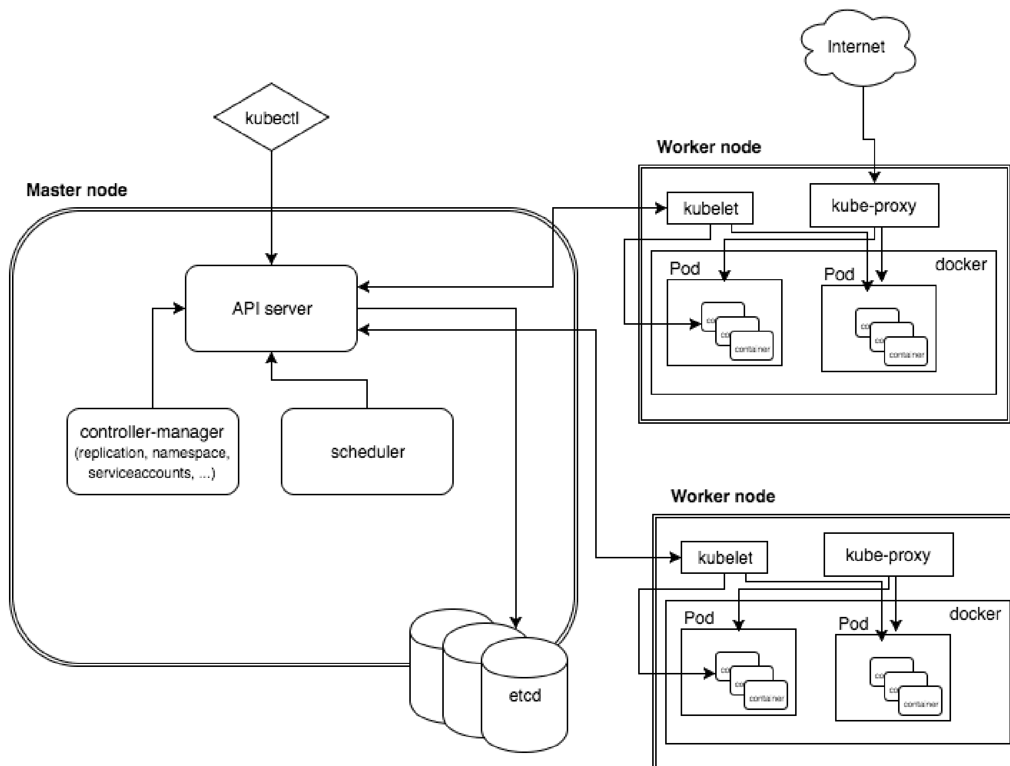
3.1.1. Master csomópont

A master csomópont felelős a klaszter irányításáért. Jellemzően egy van belőle, de nagy rendelkezésre állású klasztereknél akár több is lehet. A vezérlő sík több komponensre van bontva a könnyebb fejleszthetőség és cserélhetőség érdekében. A komponensek együttes működése révén tudja a klaszter fogadni a felhasználói kéréseket, ütemezni a konténereket, autentikációt végezni, hálózatot vezérelni és elvégezni a skálázási és klaszter egészségi feladatokat[11]. A master csomópont és komponensei a 3.1 árba bal oldalán láthatóak.

Az egyik legfontosabb komponens az etcd[2], ami egy globálisan elérhető konfiguráció tár. Ez tárolja a rendszer mindenkori állapotát és mivel globálisan elérhető,

[1] `https://x-team.com/blog/introduction-kubernetes-architecture/`

[2] `https://github.com/etcd-io/etcd`



3.1. ábra. *A Kubernetes felépítése*[1]

ezért a worker csomópontok is ezen keresztül tudják a legfrissebb konfigurációkat alkalmazni.

Az API szerver egy jól definiált JSON REST API segítségével lehetővé teszi a Kubernetes konténer alapú objektumainak (Pod, Service) és ezen objektumok életciklusának menedzselését. Az életciklus menedzsment részei az ön-gyógyulás (self-healing), skálázás (scaling), frissítés és befejezés.

A controller-manager szerver tartalmazza a különböző vezérlőket, amik az API szerveren keresztül figyelik a klaszter állapotát és a jelenlegi állapotot az elvárt állapot felé mozdítják.

Az ütemező feladata kiválasztani, hogy a felhasználó által futtatni kívánt konténer a klaszter melyik worker csomópontjára kerüljön. Figyeli a nem ütemezett Podokat, és csomópontokhoz rendeli őket a Podban definiált követelmények szerint. Ilyen követelmény lehet a konténer futásához szükséges processzor erőforrás vagy RAM mennyiség.

3.1.2. Worker csomópont

Egy Kubernetes worker csomóponton azok a szolgáltatások futnak amik elengedhetetlenek konténerek futtatásához, és ahhoz, hogy a master rendszerből menedzsel-

hetőek legyenek a csomópontok. A worker a 3.1 jobb oldalán látható, az általa tartalmazott komponensekkel együtt.

Minden worker csomópont tartalmaz konténer futtatási környezet. Ez felelős a konténerek futtatásáért és a képfájljainak kezeléséért. A legelterjedtebb a Docker[12], viszont a Kubernetes támogat bármilyen CRI[3] kompatibilis környezeteket is.

A kubelet nevű komponens biztosítja, hogy a csomóponthoz rendelt Pod leírásokban definiált konténerek fussanak és egészségesek legyenek. Ez a szolgáltatás kommunikál a master csomóponttal.

A kube-proxy egy hálózati proxy szerver, amely a komponensek közötti kommunikációért felel.

3.2. Kubernetes objektumok

A Kubernetes rendszer alap objektuma a Pod, amely szorosan együttműködő konténerek halmaza. Az egy Podon belüli konténerek egy alkalmazásnak tekinthetőek, életciklusuk megegyezik, ugyanazon a környezeten, köteteken és IP címtartományon osztoznak és a Pod minden konténere ugyanazon a csomóponton fut. A Podok leírásában metaadatok is megadhatóak, ilyen például a CPU erőforrás igény amit ütemezéskor vesz figyelembe az ütemező.

Általában nem egy darab Poddal dolgozunk, hanem a Pod replikált példányainak egy csoportjával. A Replica set-et Podok leírását tartalmazzák és skálázás során ennek segítségével indítanak új példányokat a Podból. A Replica set felelős azért is, hogy mindig a definiált példányszámú Pod fusson.

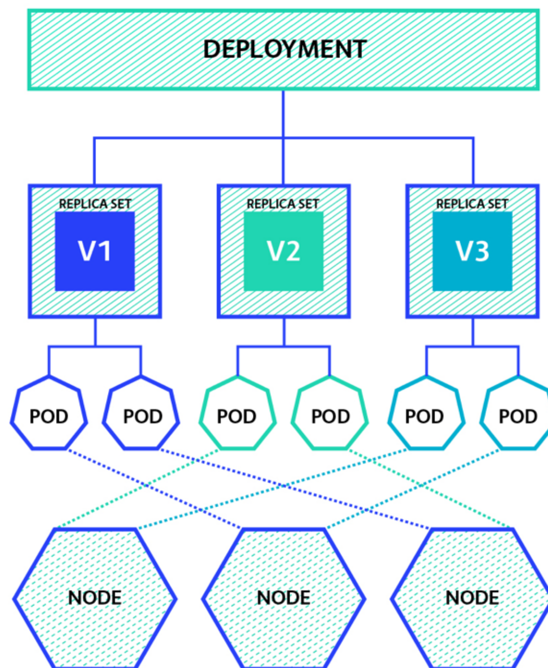
A Deployment a leggyakoribb felhasználó által kezelt objektum. A Deployment Replica setekből áll, ezért ez a magasabb szintű objektum felelős a replikált Podok életciklusának kezeléséért. A Podok frissítése szintén az Deploymenten keresztül lehetséges. A konfigurációját módosítva a Kubernetes elvégzi a megfelelő módosításokat a Replica setekben. A 3.2 ábrán látható a különféle objektumok kapcsolata egymással.

Fontos objektum még a Service, mely egy absztrakció, ugyanolyan funkciójú Podok egy logikai halmazát jelenti, melyek így egy entitásként látszanak. A Service terhelés elosztóként is viselkedik, a hozzá címzett kéréseket elosztja a benne található Podok között[5].

[3]`https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/`

[4]`https://thenewstack.io/kubernetes-deployments-work/`

[5]A `https://kubernetes.io/docs/concepts/` honlapon bővebb információ található az itt felsorolt, valamint további objektumokról.



3.2. ábra. *A Kubernetes alapvető objektumai*[4]

3.3. Ön-gyógyulás, automatikus skálázás

A Kubernetes több olyan funkcionalitást nyújt, ami megkönnyíti alkalmazások futtatását és megbízhatóvá tételét. Ilyen az ön-gyógyulás (Self-Healing) és az automatikus skálázás (AutoScaling).

A kubelet szolgálatás, ami minden worker node-on fut, periódikusan lekérdezi a Docker daemont a konténerek állapotáról. Ha egy konténer hibás állapotba került (unhealthy), akkor a Podban beállított újraindítási szabályt alkalmazza. Általában ez azt jelenti, hogy újraindítja a konténert vagy leállítja és újat indít helyette.

Az AutoScaling biztosítja, hogy egy adott típusú Podból, attól függően, hogy mekkora a terhelés, több vagy kevesebb példány fusson a rendszerben. Alap beállításként a terhelést CPU használatban méri a Kubernetes, azonban lehet definiálni saját metrikákat is. Az AutoScaling funkciót egy Kubernetes objektum, a Horizontal Pod AutoScaler (HPA)[6] biztosítja. HPA létrehozásakor definiálni kell, hogy melyik objektum példány terhelését figyelje, általában ez egy Deployment, mekkora az a CPU használati érték, ami fölött a rendszert skálázni kell és a skálázás korlátok között tartása érdekében a minimális és maximális Podszámot.

A CPU használat számítása úgy történik, hogy a Podban definiálva van egy CPU request érték, azaz, hogy a CPU mekkora részére van szüksége a futáshoz. Ez millico-

[6] `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`

re értékben van megadva, ahol 1000 millicore jelenti egy teljes CPU mag használatát. A HPA periodikusan lekérdezi az általa felügyelt Podokhoz tartozó metrikákat, ebből a CPU használatot veszi figyelembe és a következő képlet alapján számítja ki, hogy hány példánynak kell futnia a rendszerben:

$$\text{kívánt példányszám} = \left\lceil \text{jelenlegi példányszám} * \frac{\text{jelenlegi CPU használat}}{\text{kívánt CPU használat}} \right\rceil$$

A metrikákat a metrics szerver[7] gyűjti és teszi elérhetővé egy REST API-n keresztül. A metrics szerver a Worker csomópontokon futó kubelet szolgáltatástól kéri le a Podok CPU és memória használati adatait.

3.4. Gardener

A Gardener egy olyan rendszer ami Kubernetes klaszterek létrehozását és menedzselését segíti, azaz Kubernetes Clusters as a Service-t nyújt a felhasználóknak.

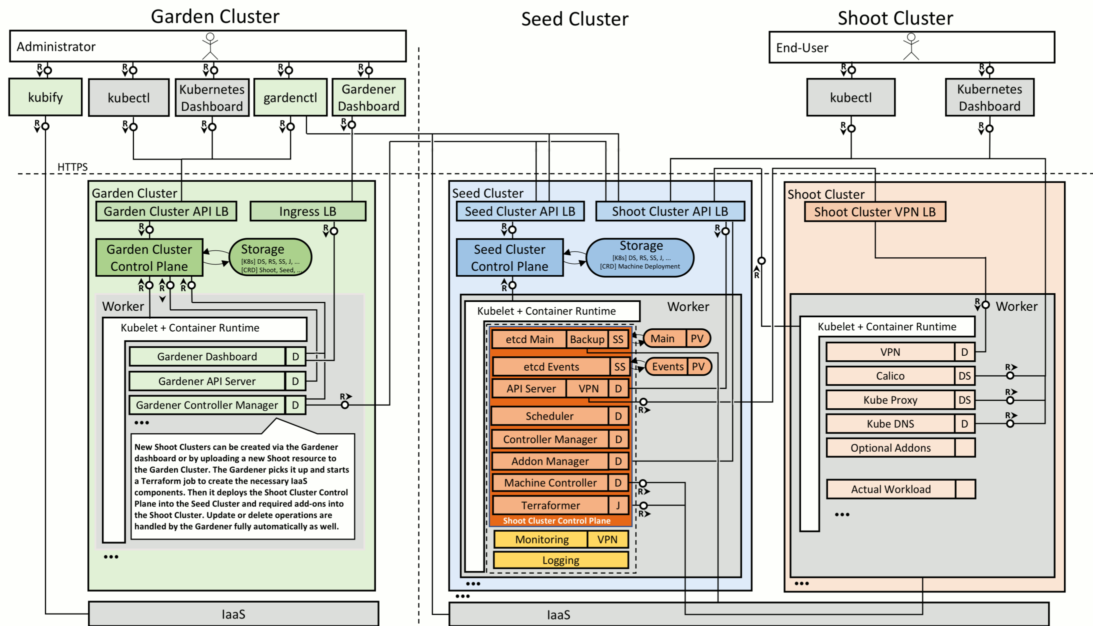
A rendszer központja a mag klaszter (Seed Cluster), itt tárolják a felhasználói klaszterek (Shoot Cluster) vezérlési síkját Kubernetes objektumokban. A vezérlési síkhoz tartozik az API szerver, az ütemező, a vezérlők és az etcd. A felhasználói klaszter worker node-jain futó kubelet a publikus interneten keresztül éri el a Seed Clusterben lévő Shoot Cluster API szervert, a Shoot Clusterben lévő Pod-ok azonban VPN (virtual private network) kapcsolaton keresztül kommunikálnak.

A Gardener rendszernek van egy központi kezelő felülete, ez az úgy nevezett Gardener Dashboard. Ez egy külön klaszterrel, a Garden klaszterrel kommunikál, ami szintén egy Kubernetes klaszter, kibővített API szerverrel, aminek a segítségével a Shoot Clustereket képes nyílvántartani, kezelni. Itt fut a Gardener, ami egy vezérlő. Figyeli a Shoot Cluster erőforrásokat és létrehozza, módosítja vagy törli ezeket. Az architekúra részletesen a 3.3 ábrán látható.

A felhasználói Shoot Cluster abban különbözik a vanilla Kubernetes-től, hogy a vezérlő elemek egy külön klaszterben találhatók, míg vanilla esetén a master node tartalmazza ezeket. A worker node-okon lévő Podok és a Seed Clusterben lévő API szerver közötti kommunikáció is eltérő. Vanilla esetén a Podok a Kubernetes szolgáltatást használva érik el az API szervert token alapú titkosított kommunikációval. Gardener esetén a Podtól az API szerver felé és a visszairányú kommunikáció is egy VPN kapcsolaton keresztül valósul meg.

Az architektúrában végzett módosításnak az az eredménye, hogy a kontroll réteg az operátor felügyelete alatt áll, felügyelheti, karbantarthatja a vezérlő erőforrásokat, így ez nem a felhasználó felelőssége. A Gardener felhasználói szempontból

[7] `https://kubernetes.io/docs/tasks/debug-application-cluster/core-metrics-pipeline/`



3.3. ábra. *A Gardener rendszer felépítése [13]*

egy biztonságos, központosított megoldás arra, hogy végfelhasználóknak Kubernetes klasztereket szolgáltassunk.

3.5. Motiváció

A Kubernetes rendszer nagyon elterjedt, disztribúciók, installerek készülnek hozzá, ami annak köszönhető, hogy a funkciók amiket nyújt, nagyban megkönnyítik konténerizált alkalmazások futtatását és menedzsmentjét.

Az AutoScaling fontos funkciója a Kubernetes rendszernek, hiszen alkalmazások futtatásánál kulcsfontosságú, hogy az alkalmazás adaptálódni tudjon a terheléshez. Ha nem skálázódik megfelelően, akkor megnövekedhet a kérések kiszolgálásának az ideje, ezáltal nő a szerver felhasználók által érzékelt válaszideje ami a felhasználói élményt (User Experience, röviden UX) negatívan befolyásolja. Ez a szolgáltatás népszerűségét csökkentheti, ami ügyfelek, felhasználók elvesztéséhez vezethet. Emiatt vizsgáltuk a Kubernetes AutoScaling képességét különböző klaszterek esetén, ahol az architektúrális felépítésből adódóan lehet eltérés a skálázódásban is. Méréseink során különböző kéréssorozatokkal terheltük a HPA-t és különböző metrikák alapján vizsgáltuk a rendszer reakcióját a terhelésre.

4. fejezet

Mérési környezet

A mérések elvégzéséhez két Kubernetes klasztert telepítettünk külön felhő rendszerekbe, így a méréseket egymástól teljesen függetlenül tudtuk elvégezni. Az AutoScaling működését több klaszterbeállítás is befolyásolja. A 4.1 táblázatban kiemeltünk ezek közül négyet, melyeket a mérések során változtattunk. A metrics szerver metrika gyűjtési periódusa, ami alap esetben egy perc, azt határozza meg, hogy mennyi idő telik el két erőforrás használat lekérdezés között. A klaszter vezérlő síkján lévő controller managernek több, a HPA működését befolyásoló beállítása van. Megadható a fel- és leskálázási késleltetés, ami két skálázás közötti minimum várakozási időt definiálja, továbbá az is, hogy milyen időközönként szinkronizálja a Podok számát. A Gardener Shoot klaszteren egyféle klaszterbeállítással végeztük el a méréseket, a vanilla klaszteren különböző beállításokat próbáltunk ki. Először az alap beállításokat használtuk, majd a Shoot klaszterrel megegyező beállításokkal vizsgáltuk a teljesítményt. Ezt követően a késleltetés csökkentése érdekében jelentős módosításokat hajtottunk végre.

	Shoot	Vanilla1	Vanilla2	Vanilla3
metrics szerver metrika gyűjtési periódus [s]	60	60	60	5
controller manager HPA felskálázási késleltetés [min]	1	3	1	1
controller manager HPA leskálázási késleltetés [min]	15	5	15	1
controller manager HPA Pod szinkronizálási periódus [s]	30	30	30	10

4.1. táblázat. *Az AutoScaling funkciót befolyásoló beállítások és a mérések során használt értékek*

A Kubernetes rendszer AutoScaling funkcióját két aspektusból vizsgáltuk. Mértük különböző terhelések esetén a skálázódás mértékét, valamint a terhelés érzékelhető késleltetésre gyakorolt hatását. Ezeket a méréseket komplex kéréssorozatokkal vé-

geztük.

Vizsgálatokat végeztünk a Kubernetes klaszter állandó nagyságú terhelésekre adott reakcióját illetően, ebből következtetéseket vontunk le a rendszer kritikus kihasználtságára vonatkozóan. A kihasználtságot a következőképpen számítottuk:

$$\text{kihasználtság} = \frac{\text{küldési sebesség}}{\text{egy szerver által másodpercenként kiszolgált kérések száma}}$$

Dolgozatunkban az a kritikus kihasználtság, ami esetén a rendszer válaszideje csupán a kérések 10%-a esetén nagyobb, mint a terheletlen rendszer válaszidejének négyszerese (a tesztalkalmazásunk válaszideje fél másodperc). A paraméterek meghatározásánál a felhasználói élmény figyelembe vétele volt a cél, mivel egy szolgáltatás négyszer hosszabb válaszideje a felhasználói élményt negatívan befolyásolhatja. Azt a küszöb értéket, ahány kérés esetén még elfogadott ekkora késés, 10%-ban állapítottuk meg.

4.1. Környezetek kialakítása

A vanilla Kubernetes klasztert egy privát OpenStack rendszerbe telepítettük kubeadm[1] segítségével. A HPA megfelelő működéséhez a metrics szervert külön telepíteni kellett a klaszterbe.

A Gardener rendszert Amazon Web Services-re telepítettük a GitHub-on megtalálható landscape-setup scriptek segítségével [14]. Ez létrehozta a Garden klasztert és beállította, hogy ez legyen a Seed klaszter is. A Gardener Dashboard segítségével létrehoztunk egy három Worker node-ból álló Shoot klasztert. A Shoot klaszterbe a metrics szerver automatikusan telepítésre került, így ezt külön nem kellett elvégeznünk.

A klaszterekben lévő csomópontok számát és erőforrásait részletesen a a 4.2 táblázat tartalmazza. A mérések szempontjából fontos volt, hogy a Worker csomópontok erőforrásai megegyezzenek a két klaszter esetén. Mivel CPU kihasználtság alapú skálázódást vizsgáltunk, ezért a memória használat alacsony, így a méréseket nem befolyásolja az eltérő RAM méret.

4.2. Teszt alkalmazás

A mérések futtatásához egy saját Node.js[2] alapú webszervert készítettünk. Úgy alakítottuk ki, hogy olyan HTTP kéréseket lehessen küldeni a szerver felé amiben definiáljuk, hogy mennyi ideig tartson a kérés kiszolgálása, és ezalatt a processzor hány

[1] `https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/`
[2] `https://nodejs.org/en/`

Vanilla Kubernetes klaszter erőforrásai			
Csomópont	Darabszám	Virtuális CPU-k száma	Memória (Gb)
Master	1	4	16
Worker	3	1	8
Gardener menedzsment klaszter (Garden + Seed) erőforrásai			
Csomópont	Darabszám	Virtuális CPU-k száma	Memória (Gb)
Master	1	2	8
Worker	2	2	8
Gardener Shoot klaszter erőforrásai			
Csomópont	Darabszám	Virtuális CPU-k száma	Memória (Gb)
Worker	3	1	2

4.2. táblázat. *Teszt klaszterek erőforrásai*

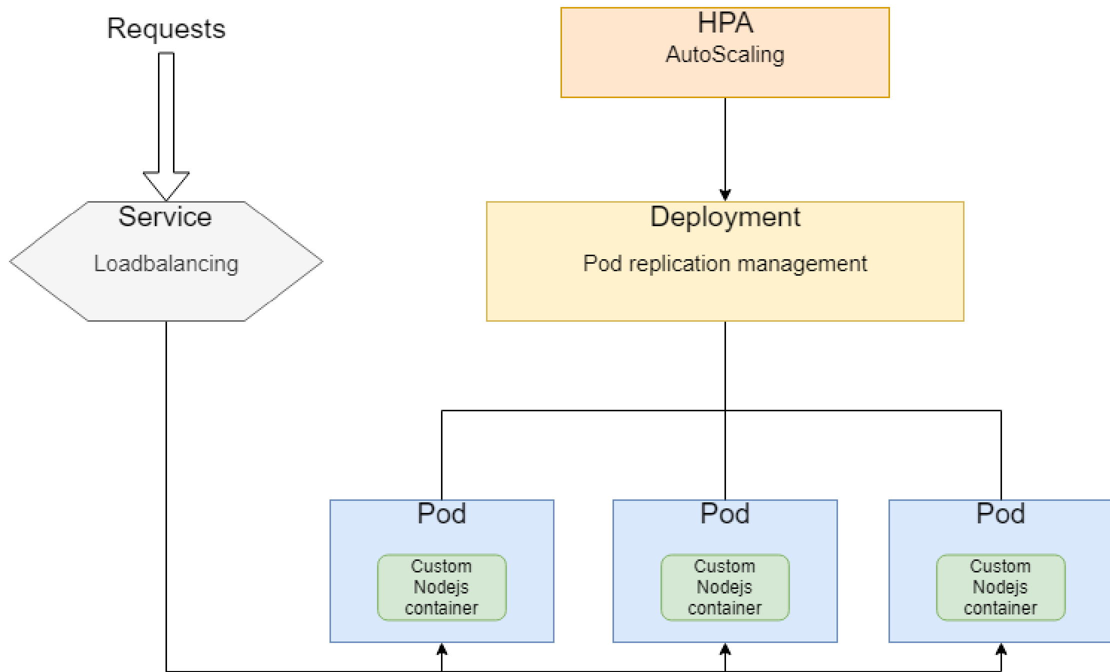
százalékát használja a szerver. Erre azért volt szükség, hogy kiszámítható legyen az alkalmazásunk működése, és releváns méréseket tudjunk végezni.

A szerver logokat készít minden kérésről amit kiszolgál, ehhez a kérésekben megadunk egy azonosítót. A logok minden kérés azonosítóhoz a kérés kiszolgálásának kezdetét és befejezését tartalmazzák.

A teszt alkalmazás felépítését a 4.1 ábra mutatja. A Node.js szervert konténerizáltuk és egy-egy Deploymentet készítettünk belőle a két klaszterben. Ezen kívül egy Service-t is csináltunk, ami a Deploymentben lévő Podok között terheléselosztást végez. Ezen a Service-en keresztül küldtük a kéréseket. Vanilla esetén a Master node egy portjára továbbítottuk a Service megfelelő portját, így a Master node IP címére tudtuk küldeni a kéréseket. A Shoot klaszter esetén a Service automatikusan kapott egy url-t amin keresztül elérhetővé vált, így ebben az esetben erre küldtük a kéréseket. Egy HPA segítségével a Deploymentet automatikusan skálázódóvá tettük, 50%-nál nagyobb CPU terhelés esetén skálázódik az alkalmazás. A skálázási tartomány 1-től 10-ig van beállítva, azaz a terheléstől függően minimmum egy, maximum tíz Pod van a rendszerben. Az alkalmazásnak küldött kérésekben beállítottuk, hogy egy kérés kiszolgálási ideje fél másodperc legyen, és közben a Pod-hoz rendelt CPU erőforrás mennyiség körülbelül 60%-át használja ki.

4.3. Mérések

Kétféle mérést végeztünk a klasztereken. A mérések során HTTP kéréseket küldtünk a Kubernetes Service-en keresztül az alkalmazásunknak, feljegyeztük a kérés sorszámát, valamint a kérés elküldésének idejét. Az alkalmazásunk által készített logok alapján megnéztük, hogy adott sorszámú kérést mikor szolgált ki. A kiszolgálás időpontjának és a küldés időpontjának különbsége adja adott kérésre a válaszidőt,



4.1. ábra. *A teszteléshez használt alkalmazás felépítése*

amit minden elküldött kérésre kiszámoltunk.

Az első típusú mérésnél olyan kérésekkel terheltük a rendszert, melyek egy valós szolgáltatásban is előfordulhatnak. Minden mérés húsz egymást követő kéréssorozatból állt. A mérés minden kéréssorozata azonosan 90 másodpercig tartott és ezeknek a sorozatoknak a küldési sebességet [kérés/másodperc] az alábbiak egyike szerint változtattuk:

1. Sorozatonként lineárisan növeltük a küldési sebességet.
2. Sorozatonként exponenciális jelleggel növeltük a sebességet.
3. „Dombszerűen”, azaz a sorozatok első felében lineárisan növeltük, majd a második felében lineárisan csökkentettük a sebességet.

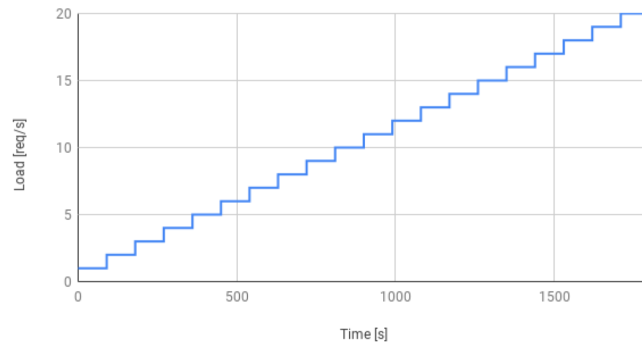
A 4.2 ábra mutatja a tervezett méréseket küldési sebesség - idő diagramon. Az ábrákon látható terhelésfajtákat mind a Shoot klaszteren, mind a három vanilla klaszterbeállításon lemértük. A méréseket automatizáltuk és minden mérést négyszer végeztünk el, majd ezeket aggregáltuk.

A másik típusú mérésnél terheletlen rendszernek küldtünk adott sebességgel kétezer kérést. Az alábbi küldési sebességekkel mértük a rendszer válaszát: 2, 3, 4, 5, 6 [kérés/másodperc].

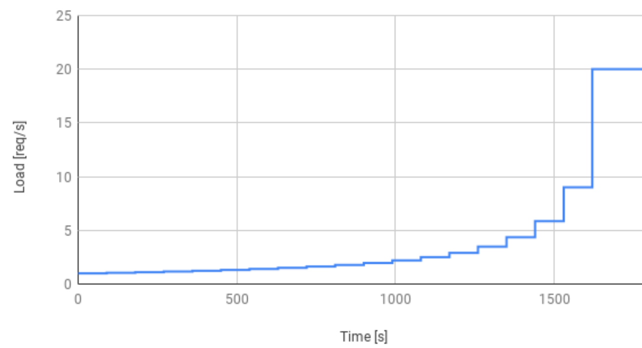
Ezeket a sebességeket megfeleltettük a kihasználtságnak. Az alkalmazás szerver másodpercenként két kérést tud kiszolgálni, ezért a sebesség és kihasználtság között

az alábbi összefüggés áll fent: szerver kihasználtság $= \frac{\text{küldési sebesség}}{2}$. Így a fenti sebességeknek az 1-szeres, 1,5-szeres, 2-szeres, 2,5-szeres és 3-szoros szerver kihasználtság feleltethető meg.

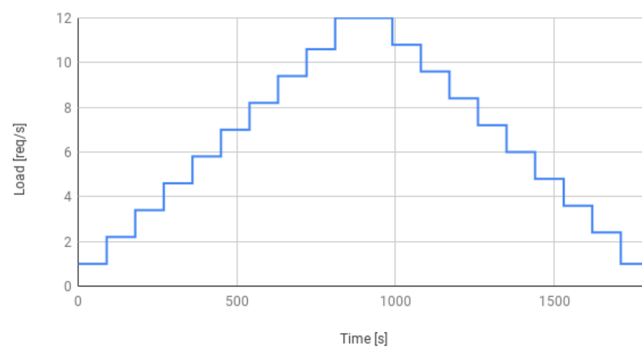
Minden kihasználtság esetén a válaszidőkből kiszámoltuk, hogy a kérések hány százaléka lépte túl a megengedett legnagyobb válaszidőt (terheletlen szerver válaszidejének négyszerese). Az eredmények alapján meghatároztunk egy kis intervallumot, ahol a kritikus kihasználtság lehet, ezt követően az intervallum határokkal ismét elvégeztük a mérést. Azt vizsgáltuk, hogy ismert terhelésekre adott válasz alapján lehet-e következtetni a kritikus kihasználtságra.



(a) *Sorozatonként lineáris sebességnövelés*



(b) *Sorozatonként exponenciális jellegű sebességnövelés*



(c) *A sorozatok első felében lineáris sebességnövelés, majd lineéris sebességcsökkentés*

4.2. ábra. *Tervezett mérések küldési sebesség - idő diagramjai.*

5. fejezet

Eredmények

Az mérések kiértékelését többféle diagrammal végeztük, ezek közül kiválasztottuk azokat, melyek a legjobban jellemzik a különböző mérések közötti különbségeket. Erre a célra a lineáris, exponenciális és „dombszerű" méréseknél a kommulatív eloszlásfüggvény (Commulative Distribution Function - CDF) és egy késleltetés-idő diagramot választottunk.

A CDF diagramon az X tengely értékei a mért késleltetési idők, az Y tengely értékei pedig megadják, hogy az összes kérés hány százaléka volt adott késleltetés alatt.

A késleltetés-idő grafikonon a kérések küldési ideje szerint ábrázoljuk az adott kéréshez tartozó késleltetés nagyságát, illetve megjelenítjük emellett a rendszerben lévő Podok számát is. Ezáltal ábrázolni tudjuk, hogy a rendszer miként skálázódik a terhelés változásával.

Mivel mindegyik mérést négyszer végeztük el, ezért a diagramokon minden értékhez látható a mérések átlaga, illetve, az adott adatnak a négy mérés közüli minimuma és maximuma. A késleltetés-idő grafikonon a rendszerben lévő Podok száma az adott pillanatban is a négy mérésnek az átlagát mutatja.

Az egységugrás típusú méréseknél az X tengelyen láthatóak a kölünböző kihasználtságok, az Y-on pedig, hogy az adott kihasználtság esetén az összes kérésnek hány százaléka volt a 4.3 fejezetben meghatározott érték felett. Az értékek szintén a négy mérés átlagos eredményét mutatják.

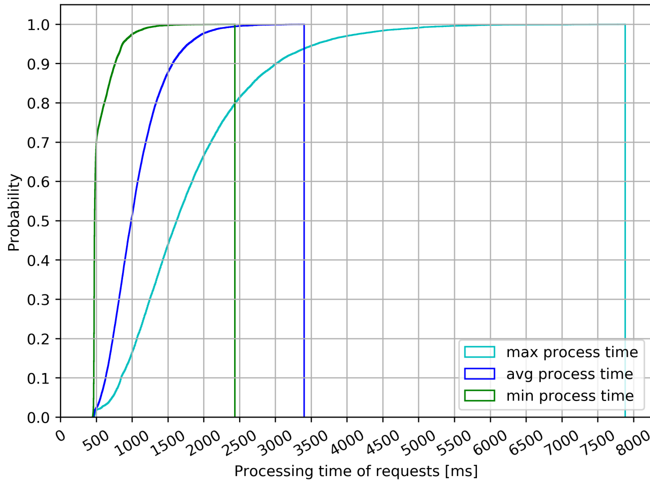
5.1. Lineáris mérések

Az 5.1 diagramokon látható, hogy a négy klaszter beállítás nagyon hasonló átlagos értékeket eredményezett. A kérések 90%-a mind a négy esetben 1,5 másodperc alatt volt, ami felhasználó szempontból megfelelő teljesítménynek számít. Eltérések csak a szélső értékekben mutatkoznak, ezek közül a Vanilla3-hoz tartozó 5.1d diag-

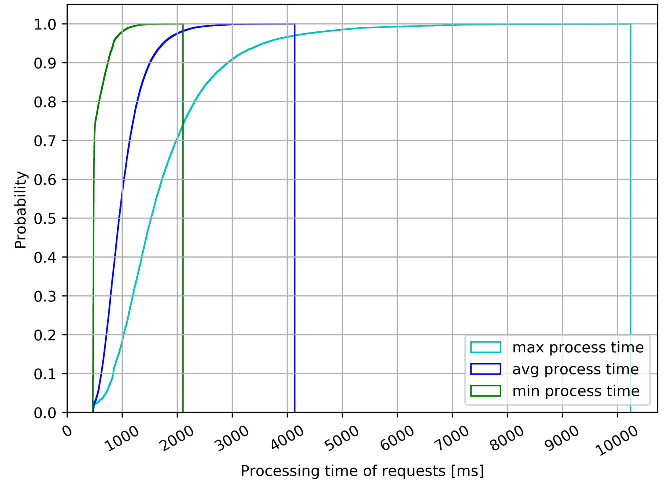
ramon látható maximális 24 másodperces érték a legmagasabb, viszont ez csak a kéréseknek egy elhanyagolható részében jelentkezik, így nem befolyásolja jelentősen a felhasználói élményt.

A hasonló teljesítmény a késleltetés-idő diagramokon is jól megfigyelhető. Az 5.2, 5.3 és 5.4 mind nagyon hasonló működést mutatnak. Eltérés itt is a Vanilla3 esetében figyelhető meg. Az 5.5 grafikonon látható, hogy a mérés elején van egy nagyon magas kilengés, utána viszont az előbbiekkel megegyezően teljesít a rendszer. Az egyéb diagramokat vizsgálva kiderítettük, hogy a kilengést az okozta, hogy a megismételt mérések közül egyik esetben a rendszer túl hamar skálázott vissza, így olyan mértékben megnőtt a késleltetés, hogy az aggregált értékeket is jelentősen befolyásolta. Ez a túl hamar visszaskálázó (a 4.1 táblázatban látható HPA leskálázási késleltetés) tulajdonság megfigyelhető a teljes diagramon. Alapvetően ez nem jelent problémát a rendszer teljesítményére nézve, viszont a túl gyakori Pod leállítás és indítás feleslegesen terhelheti a rendszert.

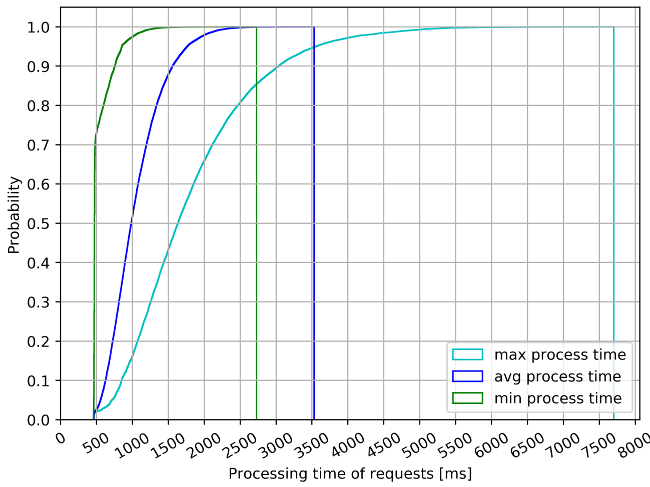
Az eredményeken elvárható volt, hogy hasonlóan teljesítsenek, mert a leglassabb beállításoknál is 1 percenként gyűjt adatokat a metrics szerver (a 4.1 táblázat metrika gyűjtési periódus értékei), így mivel kérés sorozatok másfél percesek és nem változik drasztikusan a sebesség két kérés sorozat között, a rendszernek van ideje lekérdezni a terhelést és alkalmazkodni hozzá (a 4.1 táblázatban a HPA Pod szinkronizálási periódus értékek).



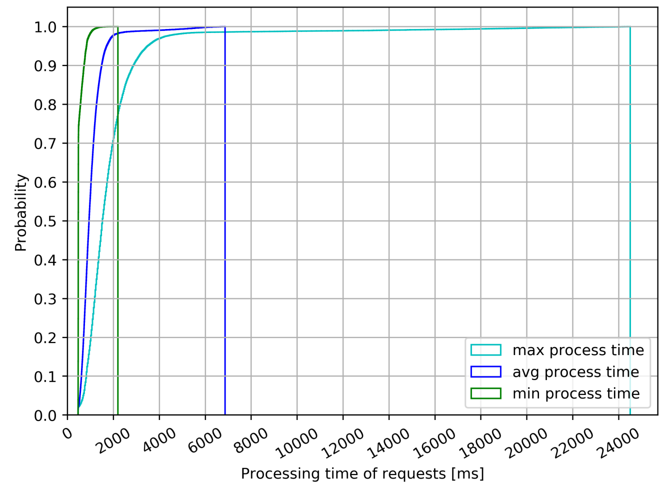
(a) *Shoot klaszter*



(b) *Vanilla1*

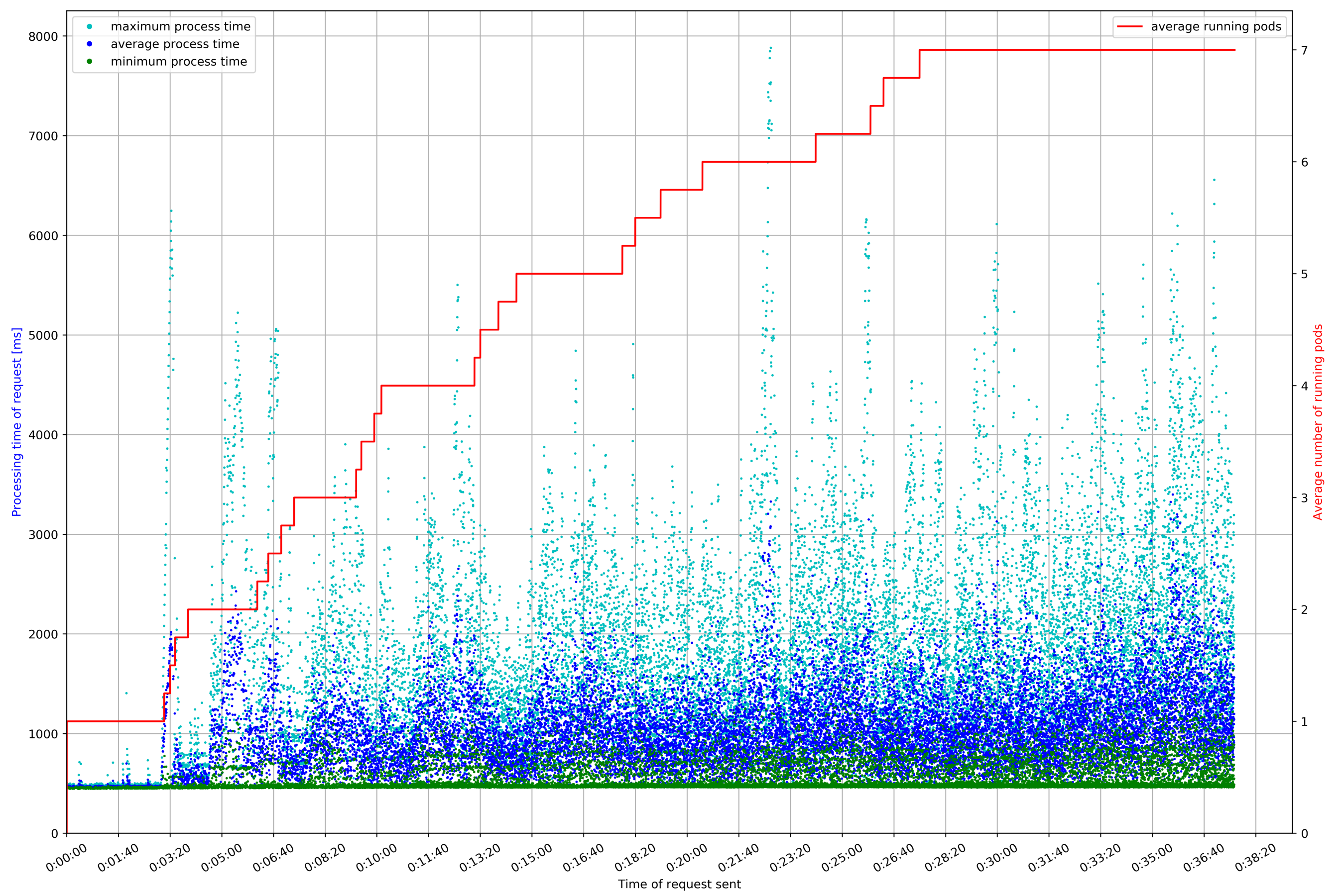


(c) *Vanilla2*

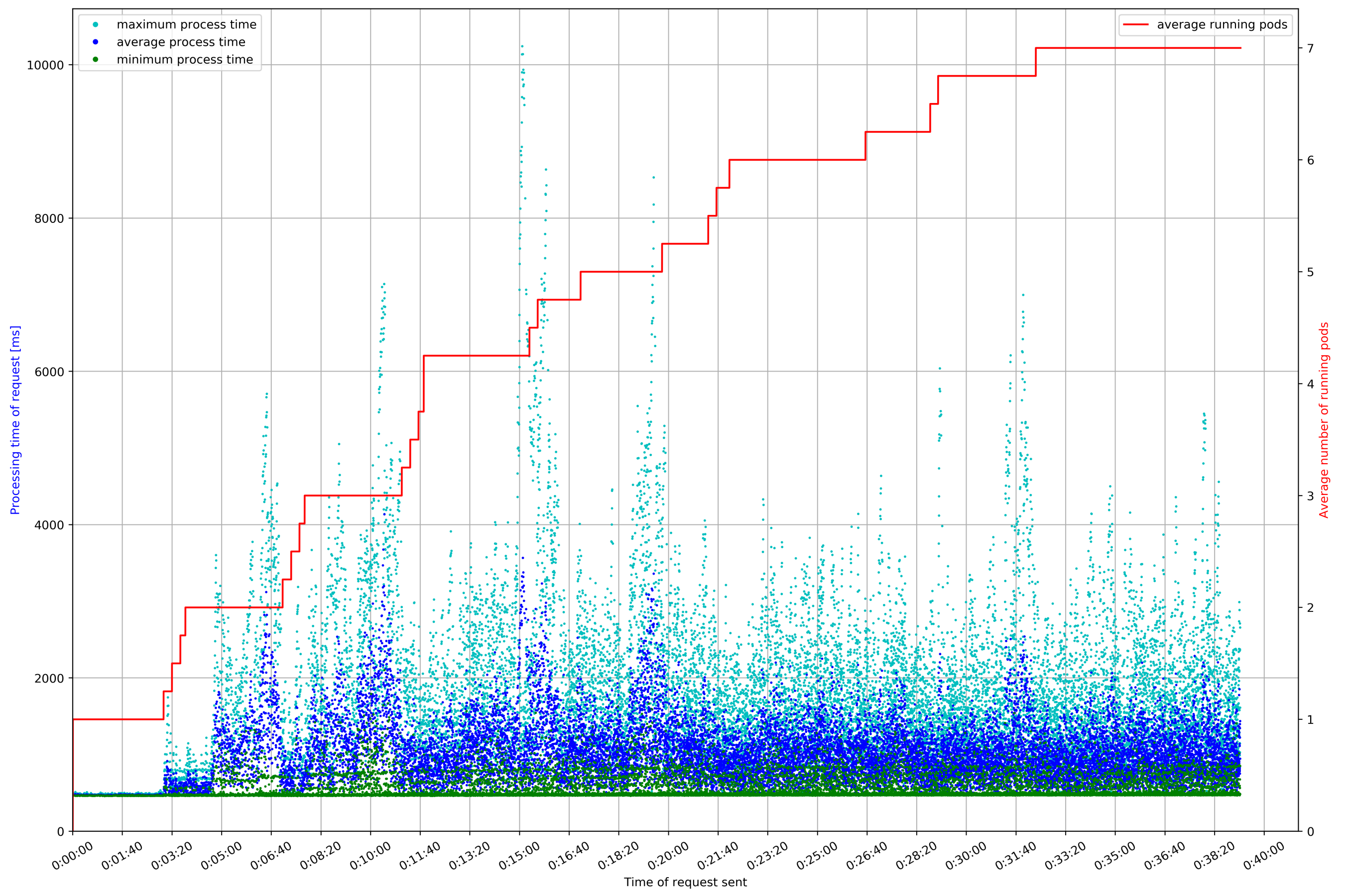


(d) *Vanilla3*

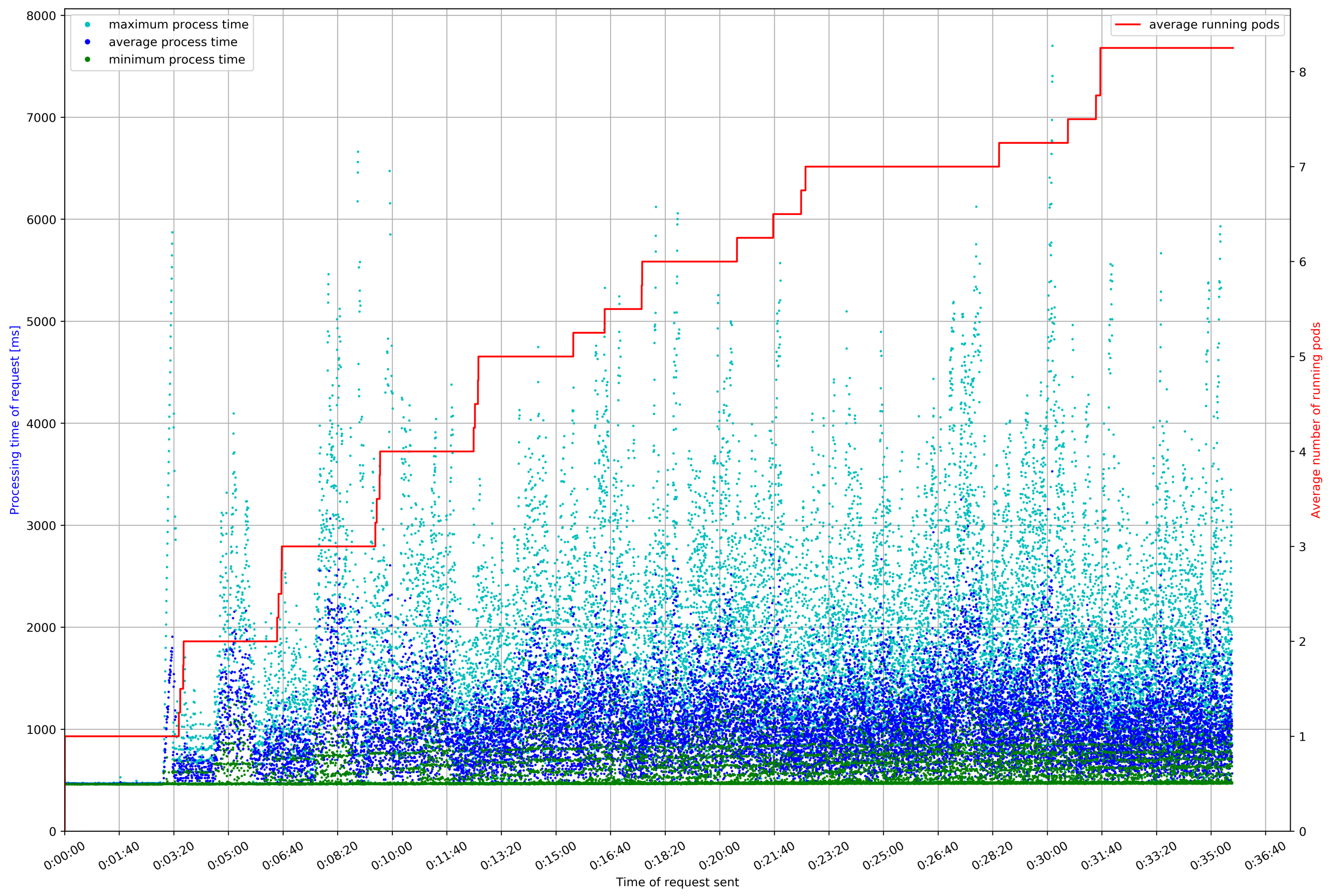
5.1. ábra. *CDF diagramok lineáris sebesség növelés esetén*



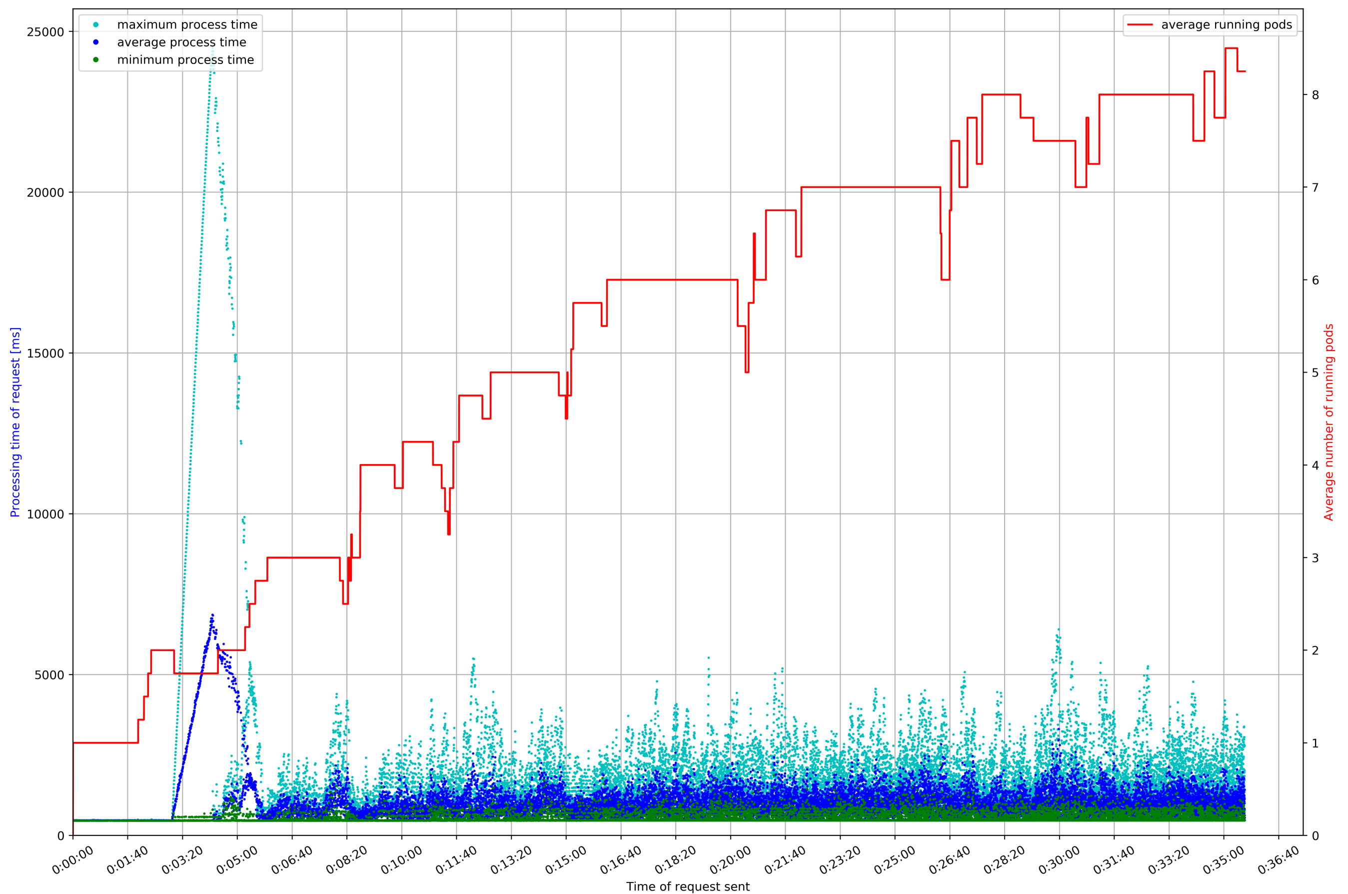
5.2. ábra. *Shoot klaszter*



5.3. ábra. *Vanilla1*



5.4. ábra. *Vanilla2*



5.5. ábra. *Vanilla3*

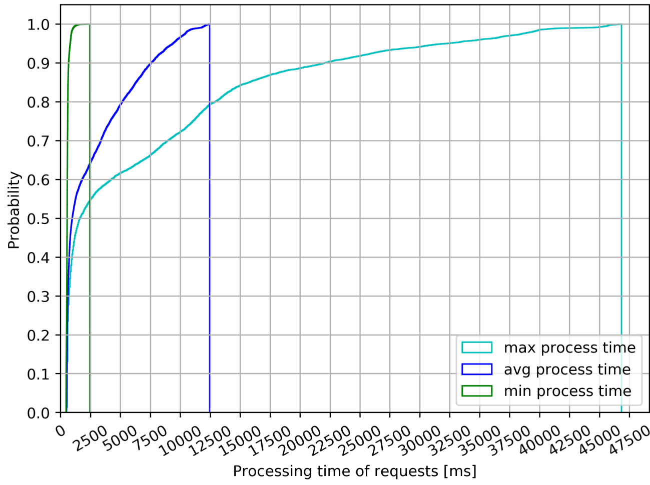
5.2. Exponenciális mérések

A lineáris teszttel ellentétben az exponenciális mérések esetén nagy különbségek figyelhetők meg az egyes klaszterbeállítások esetén. A CDF diagramok az 5.6 ábrán láthatók, megfigyelhető, hogy az átlagos válaszidőt nézve a Vanilla1-hez képest a Vanilla2, ami a Shoot klaszternek megfelelő beállításokat tartalmazza, lényegesen jobb teljesítményt nyújt. Míg Vanilla1 esetén a kérések 45%-a van körülbelül 2 másodperces válaszidőn belül, Vanilla2 esetén ez a szám 50%. Vanilla1 esetén a kérések 10%-ának egy percnél hosszabb a válaszideje, ez jelentősen rontja a felhasználói élményt. Ezzel ellentétben Vanilla2 beállításnál a kérések átlagos válaszidejét nézve nincs ilyen kérés.

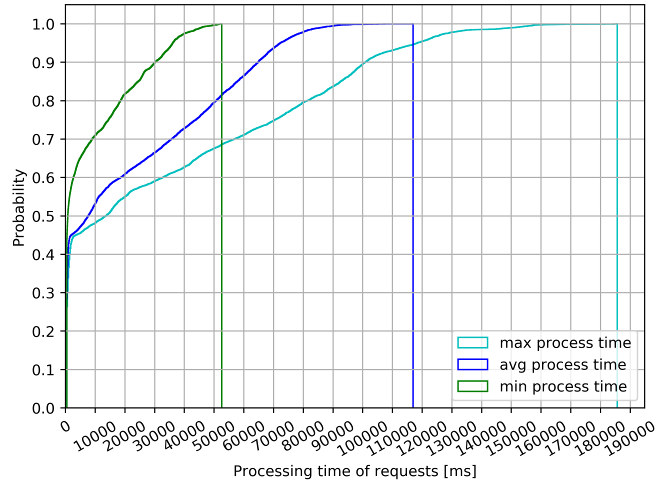
Az exponenciális mérésen a vanilla Kubernetes klaszter beállításai közül a Vanilla3 eredményei a legjobbak. Ennek oka, hogy ebben a beállításban a Podok erőforrás-

használatát gyakrabban kérdezi le a metrics szerver és a HPA Pod szinkronizációs periódusát, valamit a fel- és leskálázási késleltetést is alacsonyra állítottuk. Ezáltal dinamikusan tud alkalmazkodni a hirtelen változásokhoz. A beállításnál használt pontos értékeket a 4.1 táblázat tartalmazza. Az 5.10 ábrán látható, hogy a HPA többször is leskálázza a Podok számát, majd nagyon hamar újra felskálázza, mert megnő a terhelés. Ez a negatív hatása a gyors alkalmazkodóképességnek, amint kisebb lesz a terhelés, azonnal leskálázódik, ezáltal egy Podon nagyobb lesz a kihasználtság, ami felskálázást indukál. Ebben az esetben ez felesleges erőforrás pazarlás. A beállítások változtatásával eliminálni lehet ezt a problémát, ám ez a rendszer gyors és dinamikus alkalmazkodási képességét rontja.

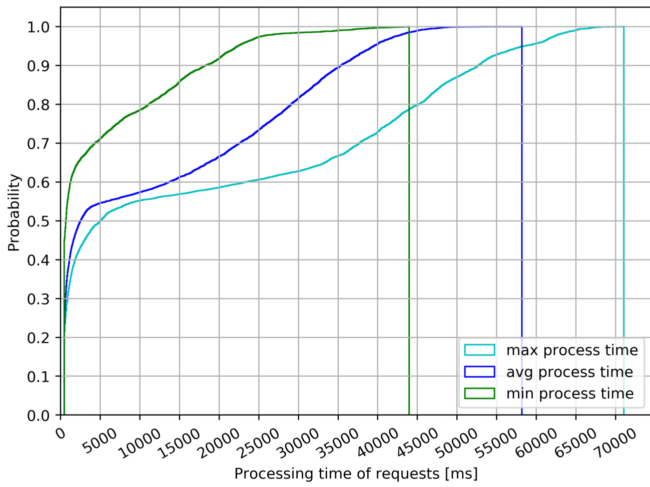
A Shoot klaszter késleltetése a legkisebb ebben a mérésben, azonban az 5.7 ábrán látható, hogy a kérések küldése időben jobban el van osztva. A teljes mérés ebben az esetben nagyjából negyven percig tartott, míg a többi esetben nagyjából harminc percig. Ez az 5.8, 5.9 és 5.10 ábrákon látható. Nem tudtuk megtalálni az okát, hogy miért voltak jobban szétosztva a kérések ebben az esetben, úgyhogy a Shoot klaszteren végzett mérés eredményeit nem tudjuk hitelesen összehasonlítani a vanilla klaszter eredményeivel.



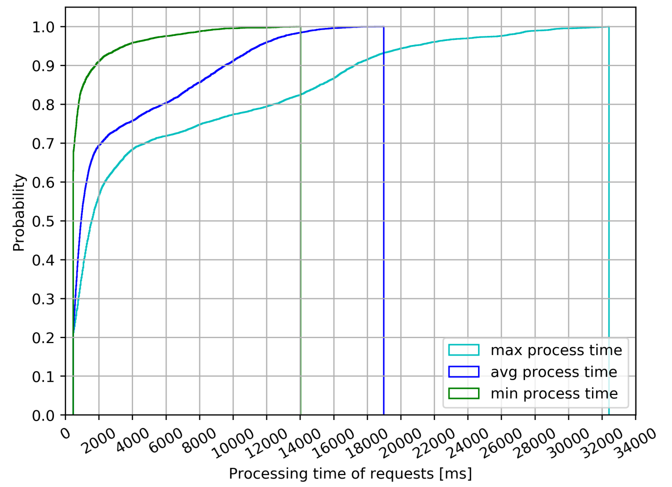
(a) *Shoot klaszter*



(b) *Vanilla1*

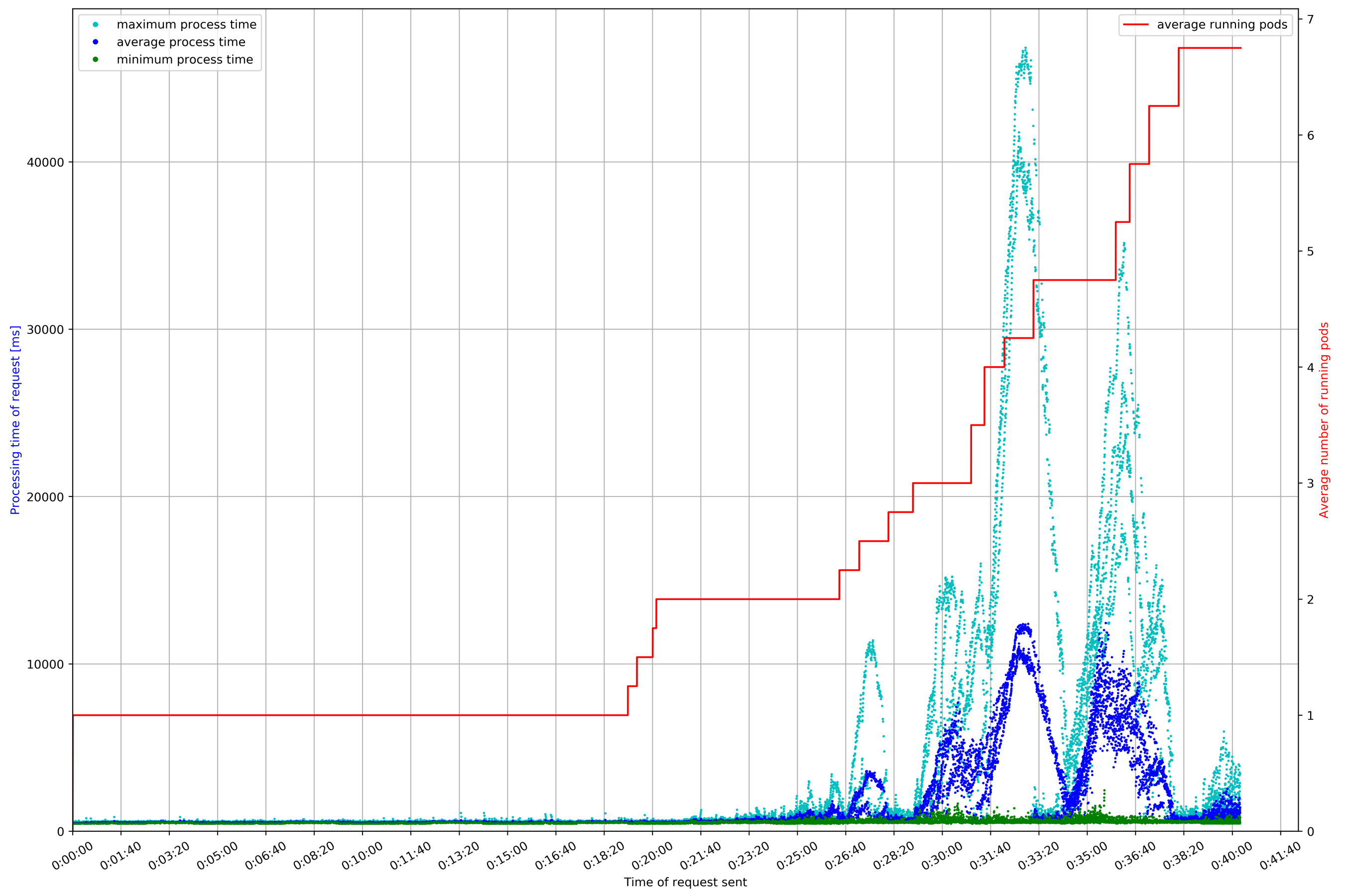


(c) *Vanilla2*

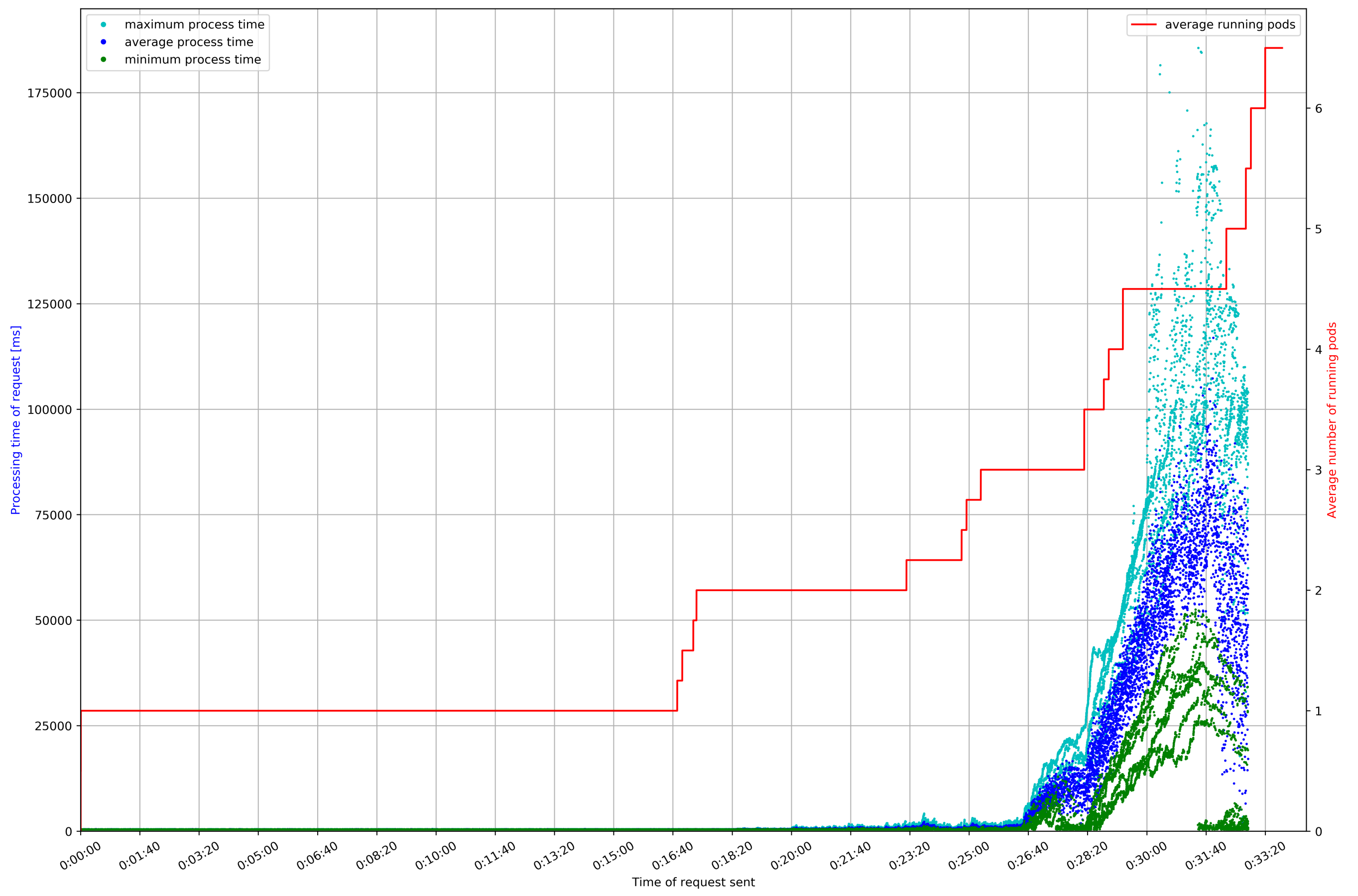


(d) *Vanilla3*

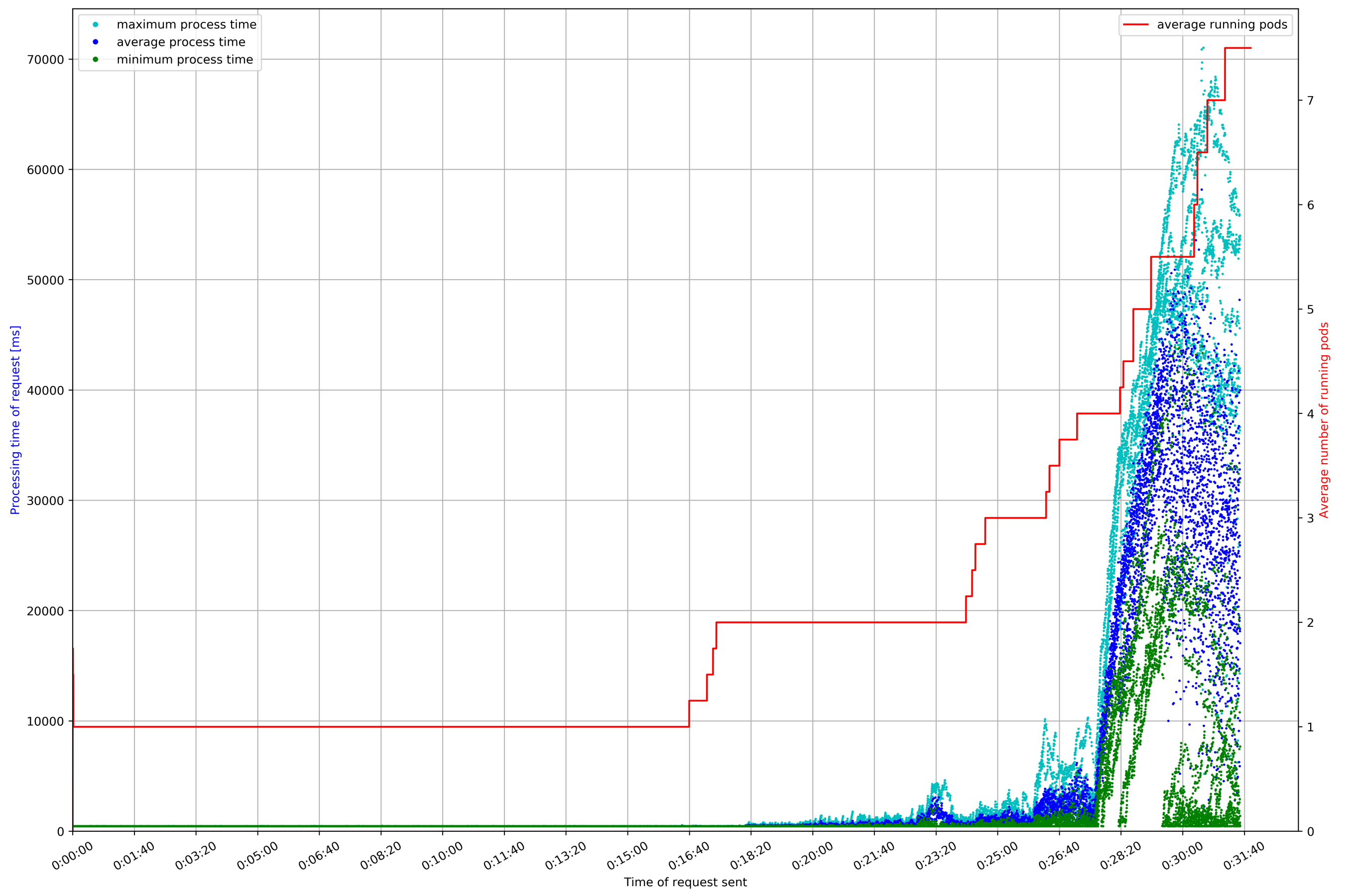
5.6. ábra. *CDF diagramok exponenciális sebesség növelés esetén*



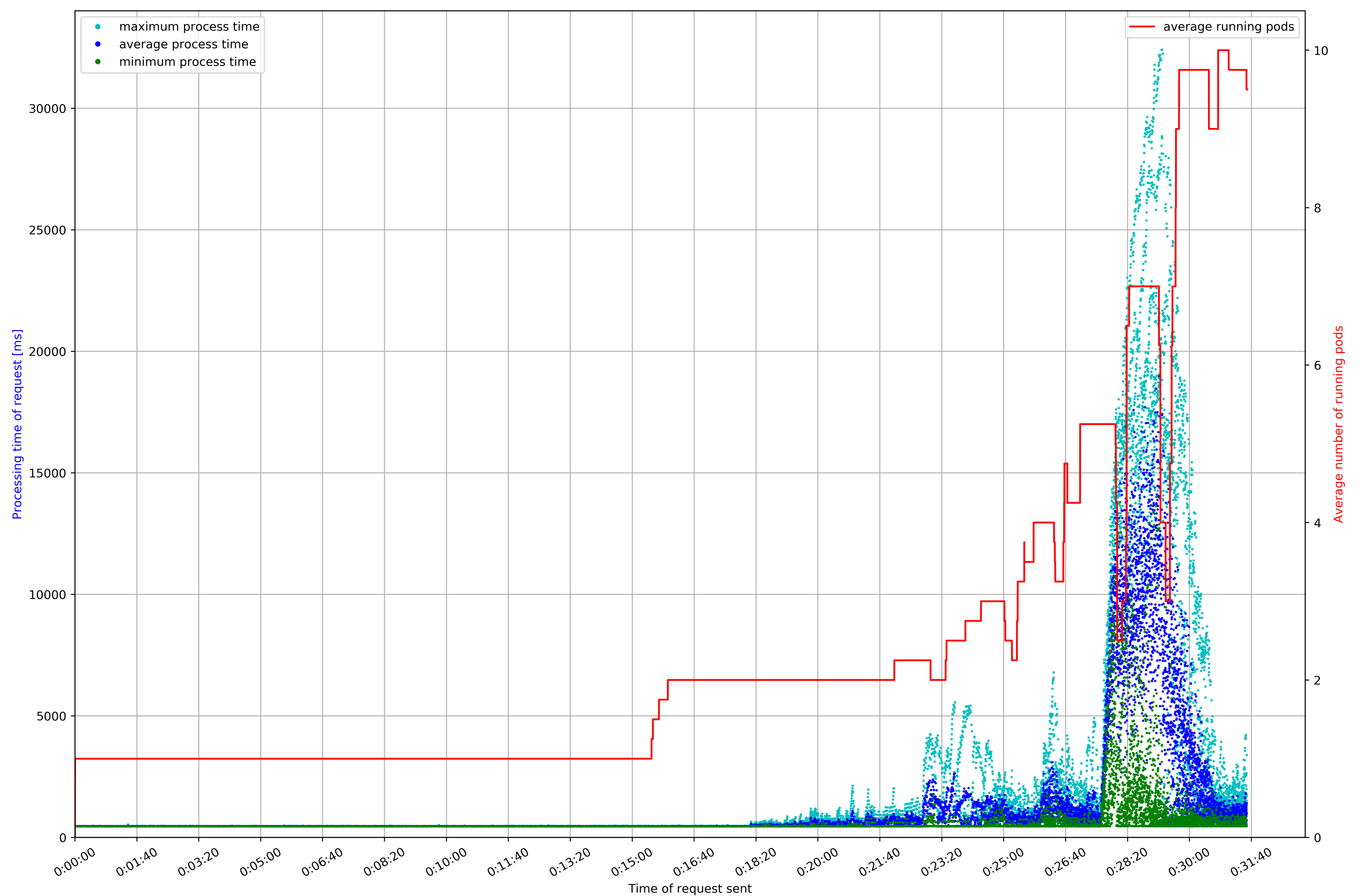
5.7. ábra. *Shoot klaszter*



5.8. ábra. *Vanilla1*



5.9. ábra. *Vanilla2*



5.10. ábra. *Vanilla3*

5.3. „Dombszerű" mérések

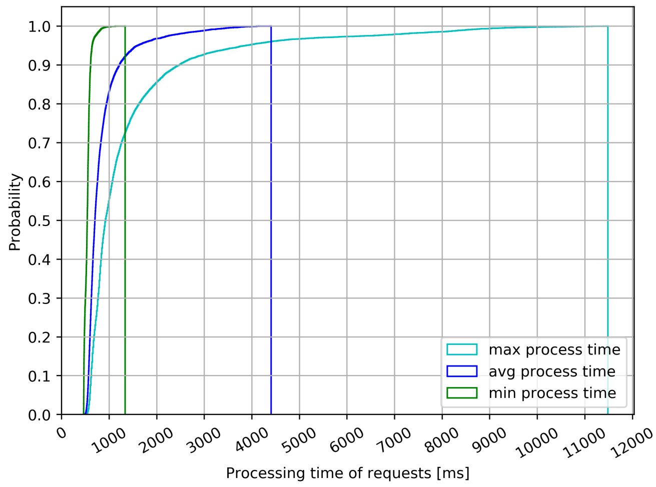
Az 5.11 ábrán szereplő CDF diagramokon látható, hogy ennél a mérésnél is a Vanilla1 szerepelt a legrosszabbul. Az 5.11b diagramról leolvasható, hogy a kérések 90%-a 3 másodperc alatt volt, miközben a Vanilla2 esetén (5.11c) ez az érték 1,5, Vanilla3 (Az 5.11d) esetében pedig körülbelül 1,3 másodperc.

Az 5.13 és 5.14 grafikonokon látható, hogy a Vanilla1 és a Vanilla2 nagyon hasonlóan viselkedik skálázás szempontjából. Ez azonban felveti a kérdést, hogy miért rosszabbak akkor a Vanilla1 eredményei. Egyéb diagramok vizsgálatával arra jutottunk, hogy a Vanilla1 beállításokkal elvégzett 4 darab „dombszerű" mérés közül 3 nagyon hasonló volt a Vanilla2 eredményeihez, viszont az egyiknél jelentős késleltetés növekedést figyeltünk meg a mérés elején, ami nagyban befolyásolta az aggregált eredményt. A megnövekedett késleltetés okára nem sikerült rájönnünk, ehhez továb-

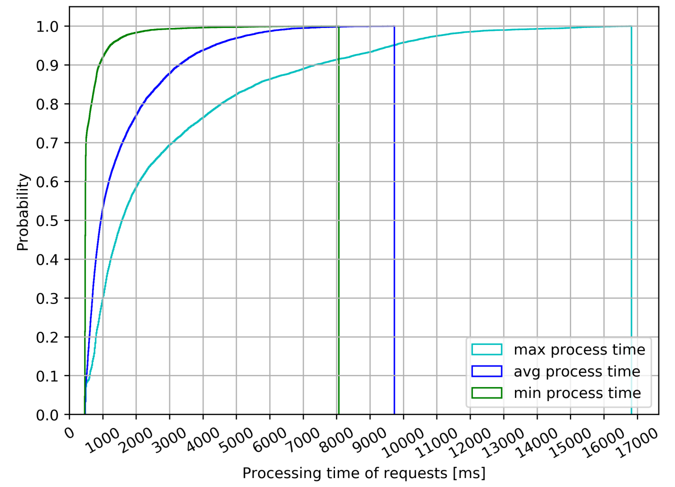
bi mérésekre lett volna szükség.

Az 5.15 diagram jól mutatja a Vanilla3 gyors leskálázási idejének a hasznosságát. A Vanilla2 esetében a terhelés csökkenésével a Podok száma csak hosszú idő után csökkent le a megfelelő szintre. Mivel a Vanilla2 és Shoot klaszter beállítasai azonosak, így ugyanez a tulajdonság figyelhető meg a Gardener esetén is. Ezekhez képest a Vanilla3-nál a terhelés csökkenésével arányosan csökken a Podok száma. A Vanilla1 hasonló skálázási tulajdonságokat mutat, azonban nem olyan gyors mint a Vanilla3 esetén. A gyors leskálázódás akkor lehet hasznos, ha fontos a klaszter erőforrásainak minél jobb kihasználása, például egy másik alkalmazás is van a klaszterben, ami így hamarabb jut erőforráshoz.

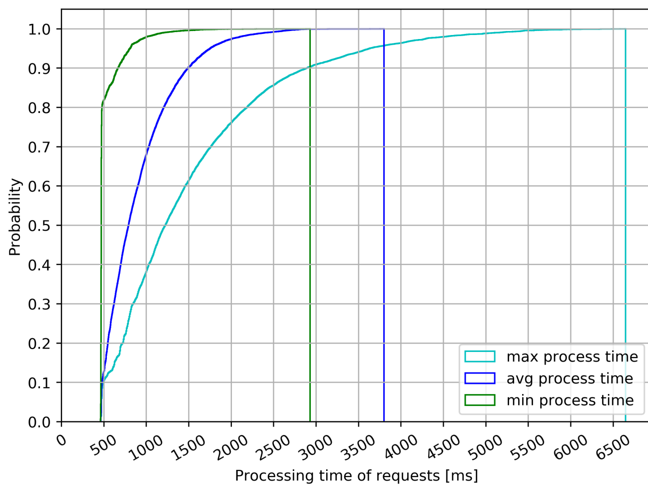
Az exponenciális méréshez hasonlóan, az 5.15 diagramon látható, hogy a Shoot klaszter esetében a kérések időben jobban el vannak osztva, így itt sem tudjuk hitelesen összehasonlítani az eredményeket a vanilla Kubernetes klaszteren végzett mérésekkel. Az eltérés okára ebben az esetben sem sikerült választ találni.



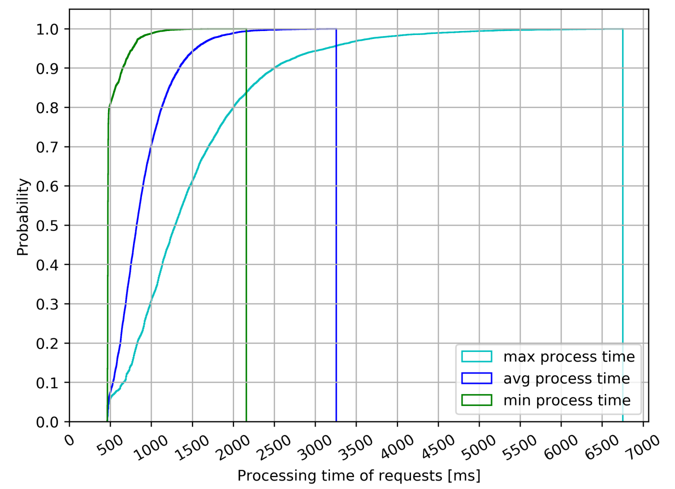
(a) *Shoot klaszter*



(b) *Vanilla1*

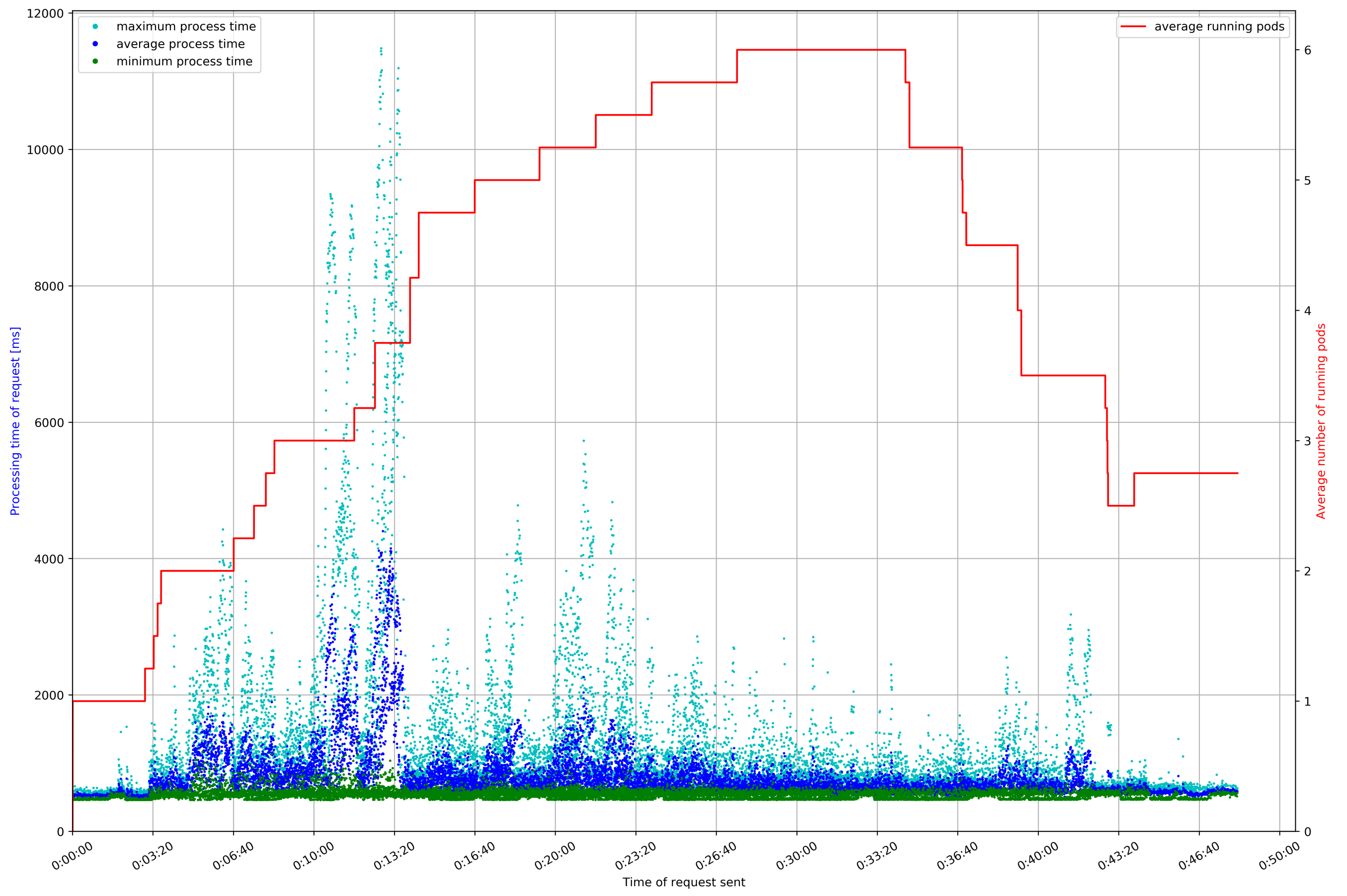


(c) *Vanilla2*

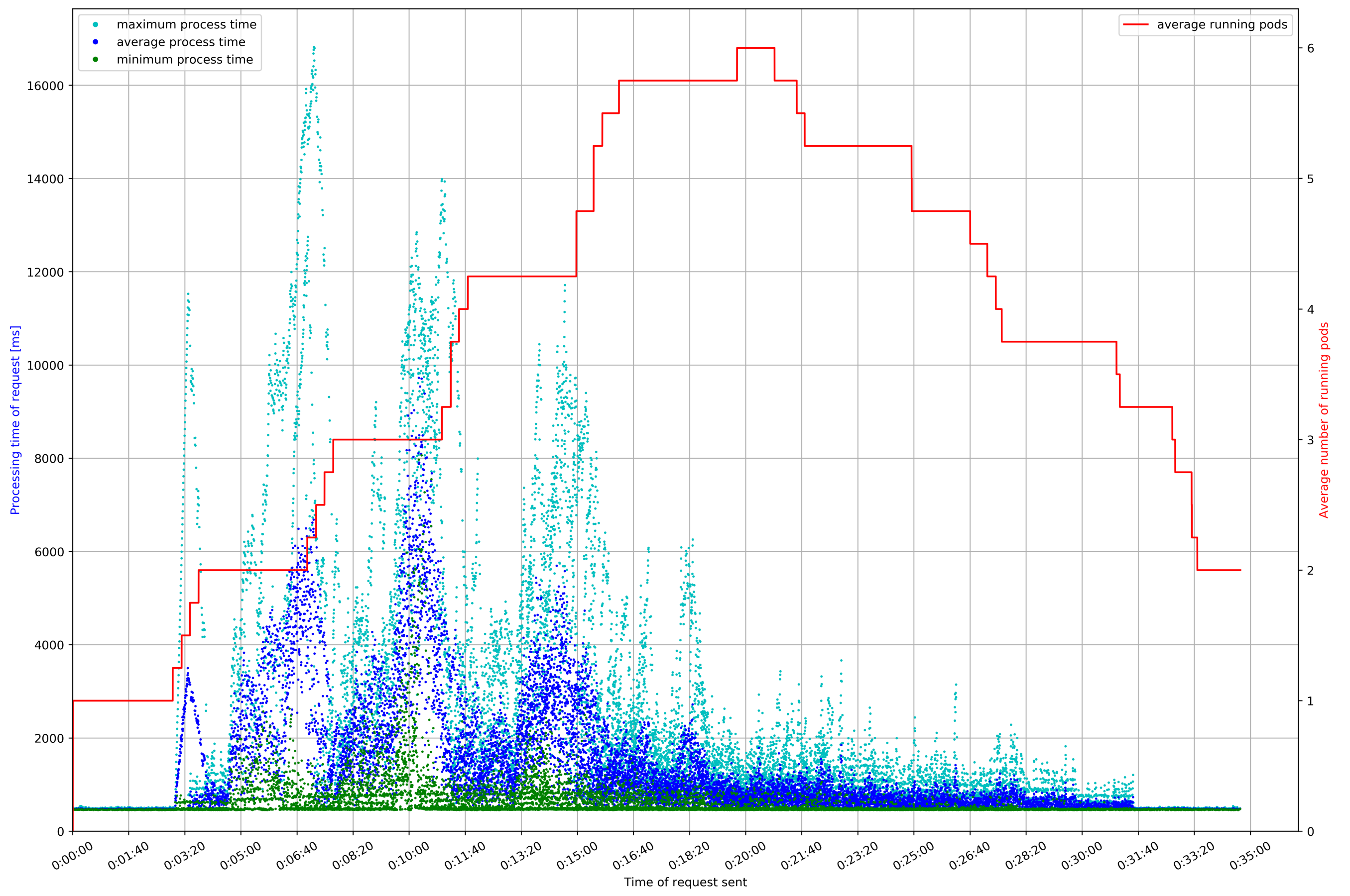


(d) *Vanilla3*

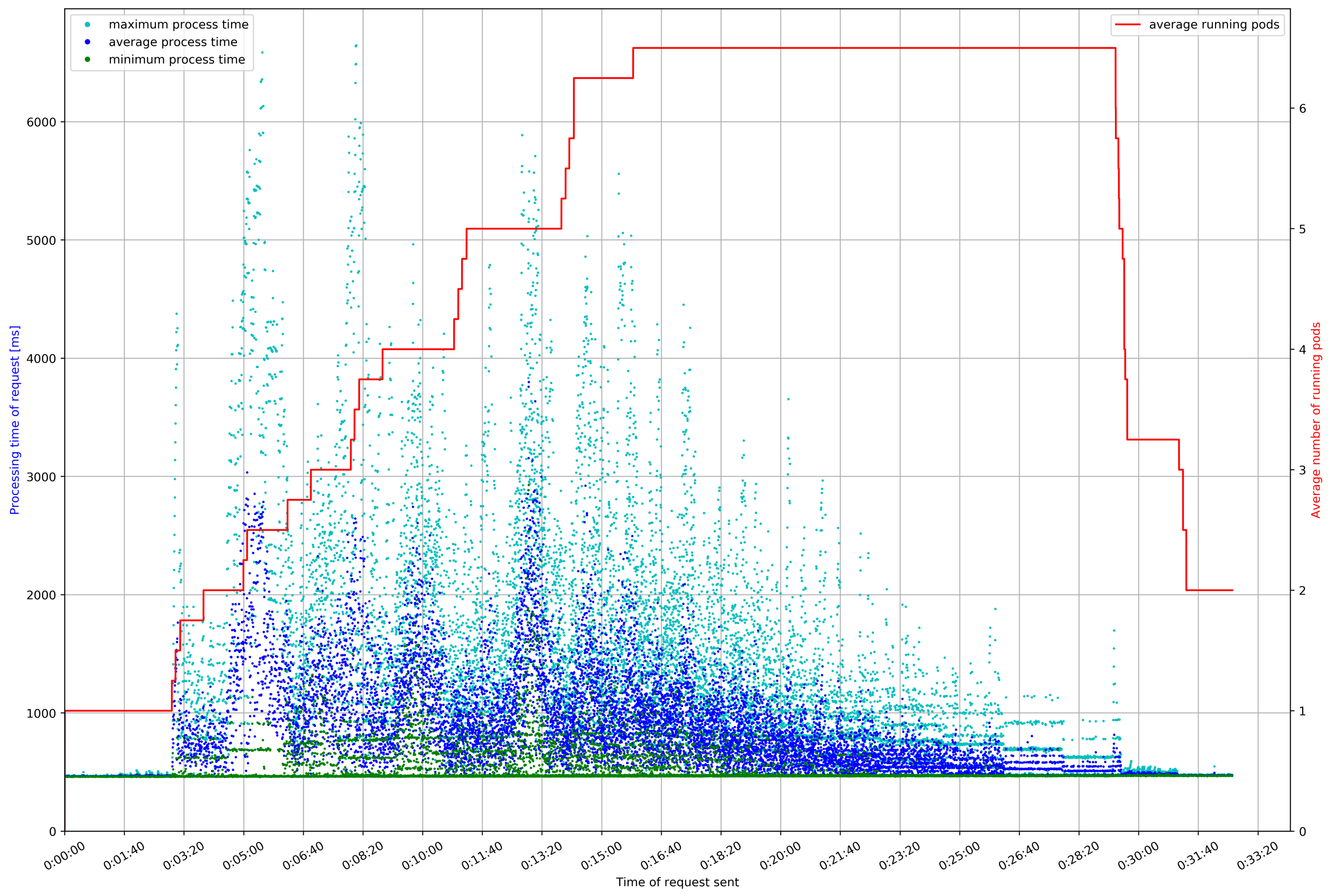
5.11. ábra. *CDF diagramok teve sebesség növelés esetén*



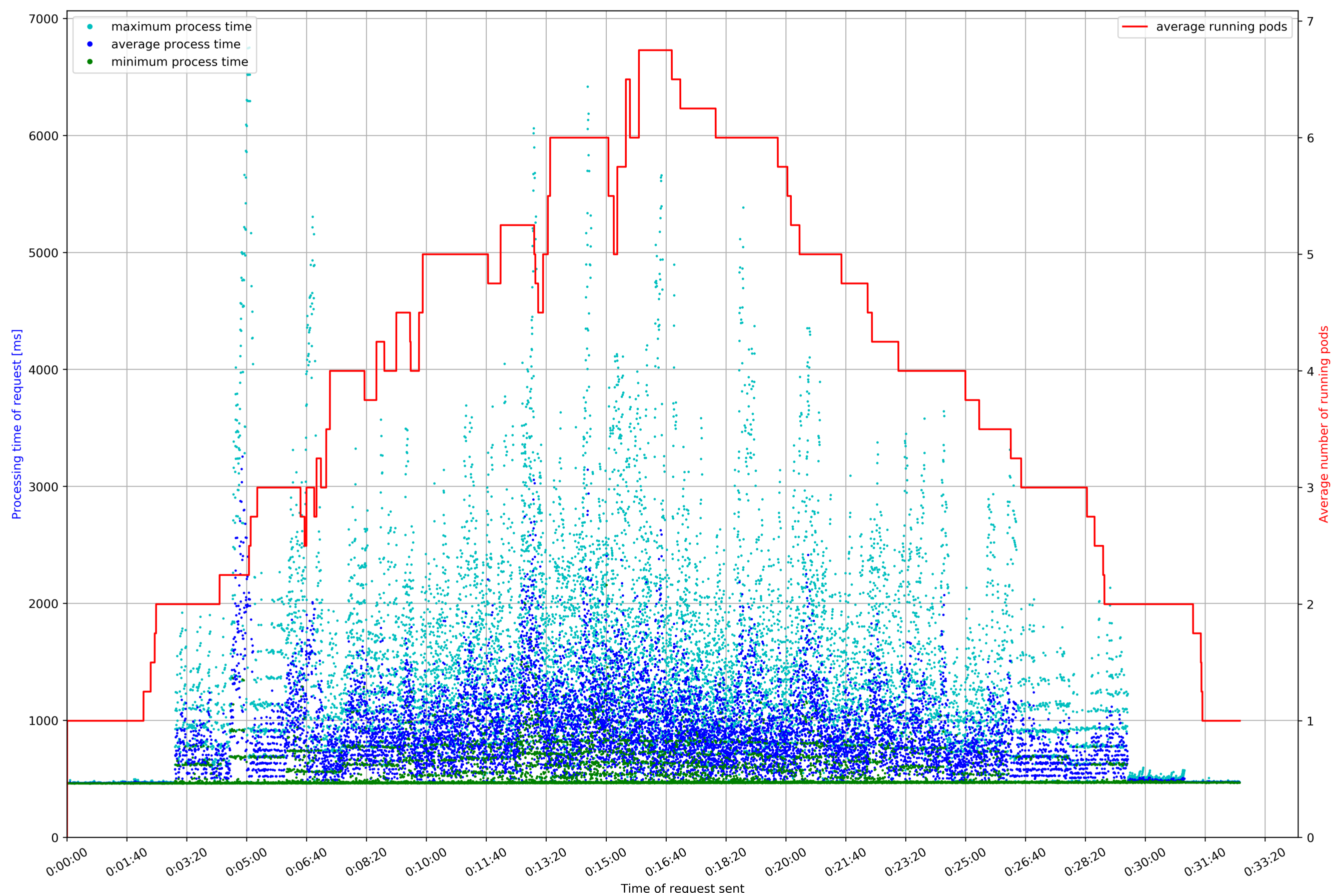
5.12. ábra. *Shoot klaszter*



5.13. ábra. *Vanilla1*



5.14. ábra. *Vanilla2*



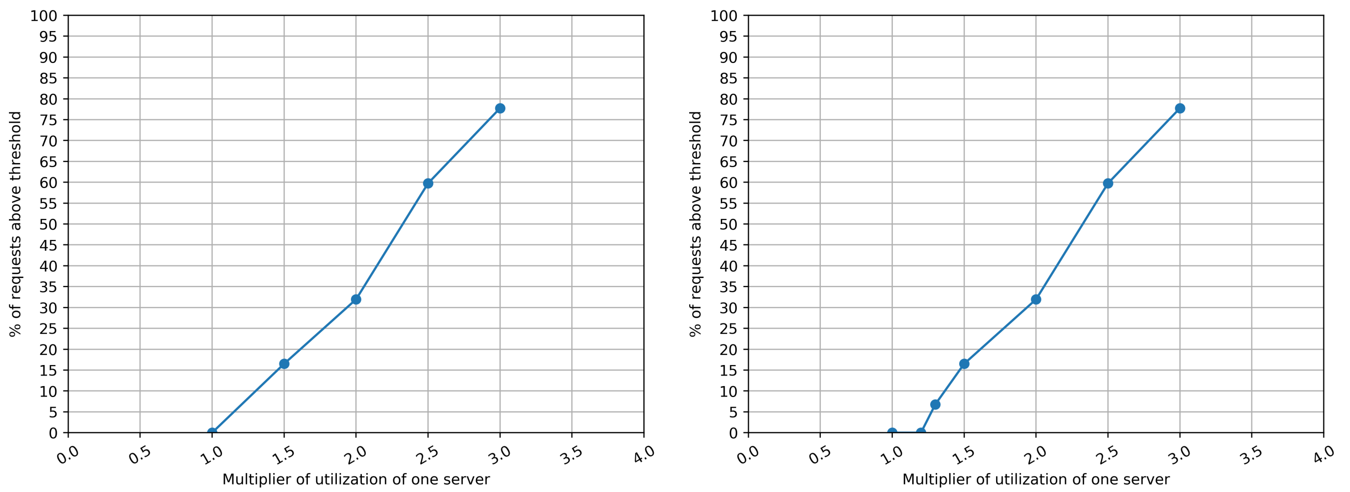
5.15. ábra. *Vanilla3*

5.4. Egységugrás mérések

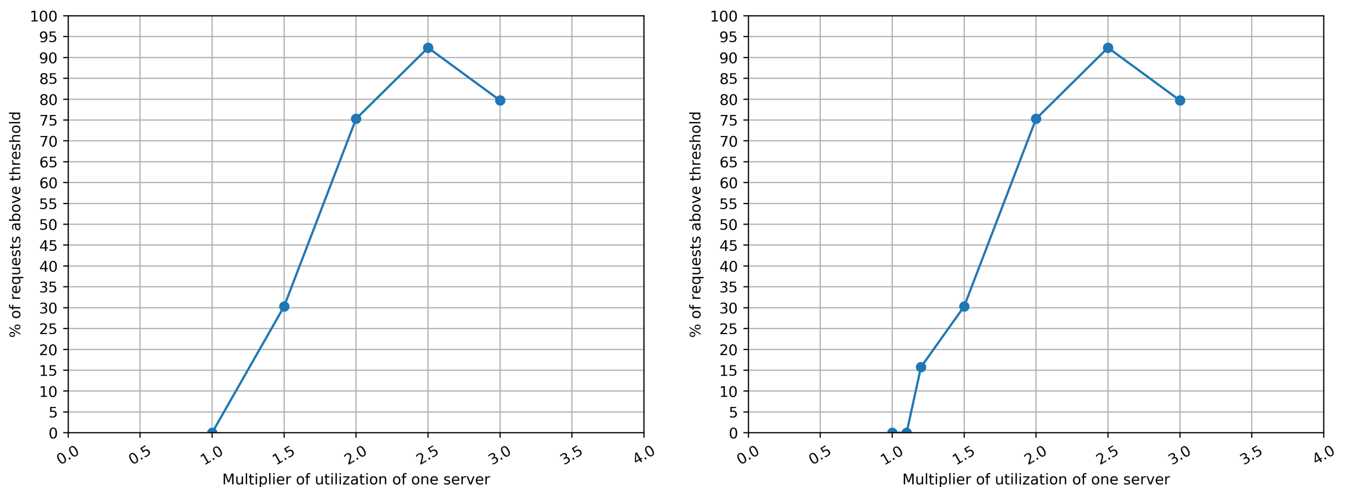
Az öt különböző kihasználtság alapján a kritikus kihasználtságot egy egyszerű módszerrel becsültük. Megnéztük, hogy az eredmények alapján melyik két egymást követő mért kihasználtságot összekötő egyenes metszi a 10%-os értéket. Ezen egyenes és az $y = 10$ egyenes metszéspontjának X koordinátájához legközelebbi két, egy tizedesjegy pontosságú érték lett a következő két kihasználtság amivel a mérést újra elvégeztük.

A Shoot klaszter eredménye az 5.16 ábrán látható. Az öt kihasználtság lemérése alapján, ami az ábra bal oldalán látható, a kritikus kihasználtságot 1,2 és 1,3 közöttire becsültük. Ezt a két terhelést hozzávéve kaptuk a jobb oldali képet, amin látszódik, hogy a keresett terhelés ezek alapján a mérések alapján valahol 1,3 és 1,5 között van. Ebben az esetben becslésünk nem volt elég pontos.

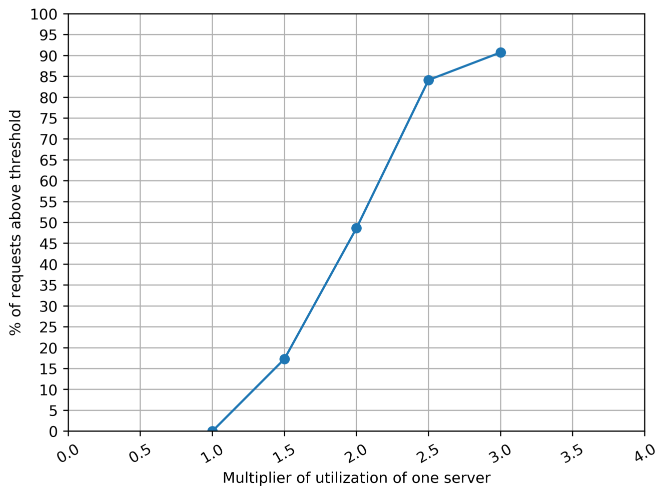
A Vanilla1 beállításokkal végzett mérést az 5.17 ábra mutatja. A kritikus kihasználtságra sikerült becslést adnunk, mégpedig 1,1 és 1,2 között. Azonban a két határ esetén nagy a különbség a küszöb felett lévő kérések számában. Míg 1,1 esetén nincs kérés a küszöb felett, addig 1,2 esetén már a kérések 15%-a a megengedett küszöb felett van a mérések szerint. A Vanilla2 és Vanilla3 beállítások esetén mért eredmények a az 5.18 és az 5.19 ábrán mutatjuk be. Előbbinél 1,2 és 1,3, az utóbbinál pedig 1,5 és 1,6 voltak a becsléseink. A becsült intervallumok itt sem tartalmazták a kritikus kihasználtságot a hét mérést összegezve.

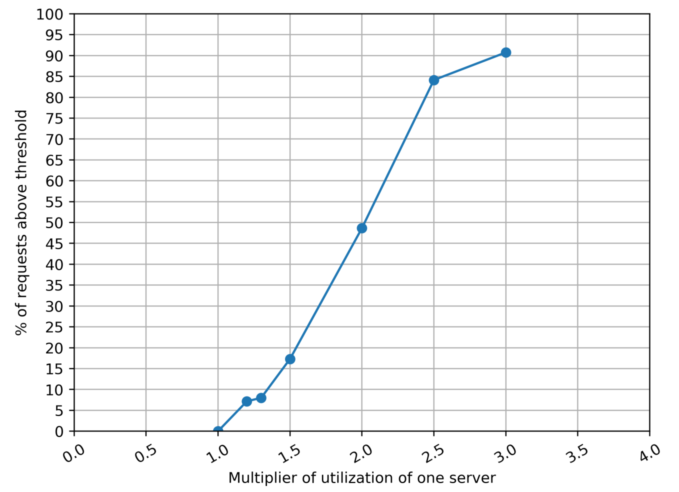


5.16. ábra. *Shoot klaszter*

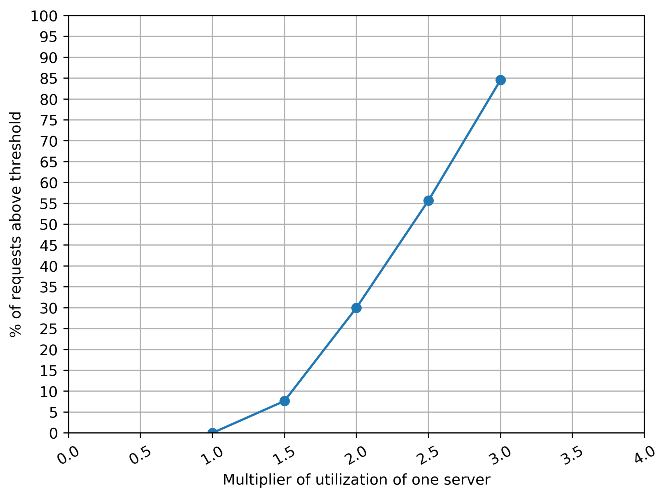


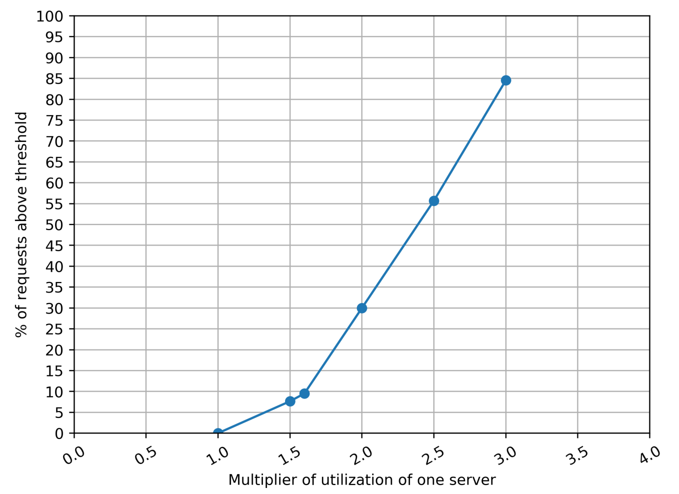
5.17. ábra. *Vanilla1*





5.18. ábra. *Vanilla2*





5.19. ábra. *Vanilla3*

6. fejezet

Konklúzió

Az elvégzett mérések alapján arra a következtetésre jutottunk, hogy a Gardener architektúramódosítása, azaz a vezérlő sík külön klaszterbe emelése, nem változtat szignifikánsan az AutoScaling teljesítményén a klaszternek. A teljesítményt sokkal inkább a 4 fejezet elején bemutatott HPA és metrics szerver beállítások befolyásolják jelentősen. Az egyik legfontosabb beállítás a terheltségi metrikák lekérdezésének gyakorisága, mivel ha ez lassú, akkor a rendszer is lassan tud alkalmazkodni a megnövekedett terheléshez, ami magas válaszidőkhöz és rossz felhasználói élményhez vezet.

Gardener esetén ezeknek a beállításoknak a módosítását a Shoot klaszter felhasználója nem tudja elvégezni, mivel a Seed klaszterben lévő vezérlő síkon található a controller manager, ahol a HPA beállításokat lehet elvégezni, illetve a metrics szerver is, viszont ezekhez a Shoot klaszterből nem lehet hozzáférni. Ettől eltekintve a Gardener egy jól konfigurált, általánosan jól teljesítő felhasználói klasztert szolgáltat. Az általunk mért terhelésekre jól reagált, hátránya, hogy egyes komponensek nehezen vagy egyáltalán nem konfigurálhatók, extrém terheléshez emiatt nehezen igazítható.

Ha több klaszterre van szükség, egyértelműen a Gardener a jobb megoldás, mivel klaszterek indítása nagyon gyorsan lehetséges, és sok komponenst előre telepít a rendszerbe. Több klaszter esetén figyelembe kell venni, hogy minden kontrol sík a Seed klaszterben lesz, így biztosítani kell, hogy megfelelő mennyiségű erőforrás álljon rendelkezésre. Ez publikus felhő szolgáltatás igénybevétele esetén költséges lehet.

Abban az esetben, ha a terhelés ingadozó, hirtelen változó, vagy jó erőforrás-kihasználást kell biztosítani (több szolgáltatás fut a klaszterben), akkor a vanilla Kubernetes jobb megoldás lehet, mivel a HPA és a metrics server is tetszőlegesen konfigurálható. Nagy hátránya viszont a telepítés nehézsége, mivel még automatizált eszközök használatával (kubeadm) is nagyon sok konfigurációt és komponens telepítést manuálisan kell elvégezni. Emiatt is érdemes lehet a Gardenert választani,

ha több klaszterre van szükség.

A kritikus kihasználtság vizsgálata alapján elmondható, hogy közelíteni tudtuk, de jó becslést csak egy esetben sikerült adnunk rá a mérések alapján. A lépésközök finomításával pontosabb eredmény érhető el. Jelen munkánk során a mérésnél nyugalmi állapotban egy Pod volt a rendszerben, azonban a mérni kívánt kihasználtságoknak megfelelő küldési sebességek meghatározásával a módszer adaptálható több Pod esetére is. Ennek a módszernek a segítségével, több mérést elvégezve az eredmények alapján felállítható egy modell, amivel előre lehet jelezni a magas válaszidejű kérések számát különböző kihasználtságok esetén. Ennek a modellnek a felállítása folytatása lehet jelen munkánknak.

A dolgozatunkban bemutatott eredmények alapjául szolgálhatnak további vizsgálatoknak a Kubernetes HPA működésével kapcsolatban és megalapozhatnak egy olyan algoritmus létrehozását is, mely a skálázást a válaszidőtől, vagyis a felhasználói élménytől teszi függővé.

Irodalomjegyzék

[1] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[2] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[3] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):10, 2016.

[4] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[5] Betty Junod. CONTAINERS AS A SERVICE (CAAS) AS YOUR NEW PLATFORM FOR APPLICATION DEVELOPMENT AND OPERATIONS. `https://blog.docker.com/2016/02/containers-as-a-service-caas/`, 2016. [Online; accessed 23-October-2018].

[6] Mike Roberts. Serverless Architectures. `https://martinfowler.com/articles/serverless.html`, 2018. [Online; accessed 23-October-2018].

[7] Ann Mary Joy. Performance comparison between linux containers and virtual machines. In *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*, pages 342–346. IEEE, 2015.

[8] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. Performance engineering for microservices: research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 223–226. ACM, 2017.

[9] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. Modelling performance & resource management in kubernetes. In *2016 IEEE/ACM*

9th International Conference on Utility and Cloud Computing (UCC), pages 257–262. IEEE, 2016.

[10] Emiliano Casalicchio and Vanessa Perciballi. Auto-scaling of containers: The impact of relative and absolute metrics. In *Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on*, pages 207–214. IEEE, 2017.

[11] Justin Ellingwood. An Introduction to Kubernetes. `https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes`, 2018. [Online; accessed 24-October-2018].

[12] John Fink. Docker: a software as a service, operating system-level virtualization framework. *Code4Lib Journal*, 25:29, 2014.

[13] Andreas Herz. Architecture :: Gardener documentation. `https://gardener.cloud/030-architecture/`. [Online; accessed 25-October-2018].

[14] Gardener setup scripts. `https://github.com/gardener/landscape-setup`. [Online; accessed 25-October-2018].