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

Madárhang azonosító és riasztó felhő-natív rendszer

TDK dolgozat

Készítette:

Torma Kristóf
Pünkösd Marcell

Konzulens:

Dr. Maliosz Markosz
Dr. Simon Csaba

2020

Tartalomjegyzék

Kivonat

Napjainkban a mezőgazdaságban egyre elterjedtebbek a dolgok internetére (Internet of Things – IoT) épülő megoldások, ezek viszont nagy mennyiségű adatot generálnak, amelyek feldolgozása tradicionális rendszerekkel nehézkes. Erre a problémára tud megoldást nyújtani egy jól skálázódó felhő-natív adatfeldolgozó és elemző rendszer, amelynek tervezése és megvalósítása több kihívást is rejt magában. Kártevő madarak hang alapján történő gyors és automatikus azonosítása fontos feladat, ugyanis például a seregélyek akár több tízmillió forint kárt is képesek okozni egy nagyobb szőlőbirtoknak. Dolgozatunkban seregélyek hang alapján történő azonosítása és riasztása a kitűzött cél, ezzel segítve a szőlősgazdákat. Az ilyen rendszerek esetében fontos elvárás a közel valós idejű reakció, különben nem képes hatékonyan támogatni a megfigyelni vagy felügyelni kívánt folyamatokat. Ezért a rendszer komponenseinek telepítése és elhelyezése során biztosítani kell, hogy egy adatpont átfutási ideje és az egyes komponensek válaszideje is bizonyos keretek között maradjon. A rendszer tervezése során további lehetőségeket vetnek fel az az edge cloud megoldások. Dolgozatunk keretein belül egy ilyen rendszer tervezését és fejlesztését mutatjuk be. Ehhez elkészítettük a rendszer architektúrájának tervét, a komponensek közötti interfészeket. A rendszer kidolgozása során számos általános problémát megoldó komponens készült el, valamint több, a seregélyek hangját felismerő elemet is felhasználtunk. Ezen az elkészült rendszeren méréseket végeztünk, hogy annak változó méretű és jellegű terhelés alatti működését felmérjük. Az elvégzett méréseket és azok eredményeinek analízisét automatizáljuk, felgyorsítva ezek értelmezését. A tervezés és fejlesztés során cél kizárólag nyílt forráskódú és szabadon elérhető komponensek, valamint szolgáltatófüggetlen Kubernetes technológiák használata. Így a rendszer felhő szolgáltatók között gyorsan és könnyedén hordozható marad, de komolyabb nehézségek nélkül telepíthető saját Kubernetes fürtbe is.

Abstract

Nowadays many smart agricultural solutions leverage the novel mechanisms of Internet of Things (IoT), which in turn generate large volume of data that cannot be processed efficiently with legacy data processing methods. This shortcoming can be successfully addressed by cloud native data processing and analytics systems, but due to their complexity, the design and deployment of such systems pose their own challenges. Fast and automatic recognition of pest birds is important as they can cause great financial harm to larger vineyards. In our work we focused on the detection and deterrence of Sturnus Vulgaris, one such pest bird. The IoT systems we discuss require tight, near instant response times, otherwise they cannot efficiently support the processes they are supposed to monitor or control. To ensure this, during installation and scheduling of each component the round-trip time of datapoints and the response time of each component must be kept within certain limits. When planning such systems edge cloud solutions give developers tools to achieve this. As a part of our paper we discuss the design and development of such systems. To do this we designed and developed such a system from the ground up and measured its behaviour under loads of varying size and nature. We automated these measurements and their analysis to speed up further research. During design and development, we had a goal to exclusively use free and open source components as well as a provider-independent Kubernetes distribution. By doing this our system can be moved between cloud providers with relative ease and it may also be deployed to a self-hosted Kubernetes cluster quite easily.

1. fejezet

Bevezetés

A felhő alapú technológiák elterjedésével egyre növekszik az ilyen megoldások iránti igény. Az ebben rejlő lehetőségek kihasználására képes alkalmazások fejlesztése egyedi és új kihívásokat jelent. Az ilyen alkalmazások komponensei általában egymástól elválasztottak, lazán kötődnek és állapotmentesek. Ez tipikusan, de nem szükségszerűen mikroszolgáltatás alapú alkalmazás architektúrát, agilis fejlesztési folyamatokat, konténer alapú technológiák használatát jelenti.

Szőlőtulajdonosoknak éves szinten jelentős kárt okoznak a seregélyek, akik előszeretettel választják táplálékul a megtermelt szőlőt. Az ilyen kártevő madarakat elriasztó eszközök általában folyamatos ultrahang kibocsájtásával [1] érik el a kívánt hatást. Ezekhez képes egy intelligens, célzottan működő rendszer a környezetre kisebb ráhatással is elérhet hasonló eredményeket, hiszen nincs szükség a többi állatot is zavaró ultrahang kibocsájtására, ha a kártevők nincsenek jelen. Emellett az Internet of Things és a felhő adta lehetőségeket kihasználva a szőlőültetvények számára az általunk készített rendszer további hozzáadott értéket is hordoz.

Az Internet of Things eszközökről általánosságban elmondható, hogy olyan mindennapi használati tárgyak vagy ipari eszközök, amelyeket hálózati kapcsolattal és számítási kapacitással is felruháztak. Az említett hálózati kapcsolat lehet többek között egyes eszközök közötti, vagy akár egy felhőben lévő rendszerhez is kapcsolódhatnak a készülékek. Nagy számú eszközpark esetén megoldandó probléma a készülékek állapotának megfigyelése és központi irányítása is.

Azért, hogy a riasztás effektív legyen elengedhetetlen a relatív alacsony körbefordulási idő a rendszer és az egyes eszközök között. Ahhoz, hogy megtudjuk egy adott rendszerben ez az érték miként alakul méréseket kell végezzünk. Mikroszolgáltatás alapú rendszerek esetében az is érdekes információ lehet, hogy az egyes mikroszolgáltatások mennyi idő alatt képesek egy adatpontot feldolgozni, hogy reagálnak nagy terhelésre és mennyire skálázhatóak, valamint ezeknek milyen hatása van az egész rendszerre.

Célunk egy seregélyeket hang alapján felismerni és elriasztani képes rendszer megtervezése és fejlesztése volt, amely kizárólag nyílt komponenseket használ fel. Kiemelten fontos volt, hogy az általunk elkészített rendszerbe későbbi fejlesztések során könnyedén illeszthetők legyenek más típusú adatok és kimenetek. A rendszer architektúráját ezen

szempontok és a korábban ismertetett követelményeket figyelembe véve készítettük el. A rendszer elkészülte után megterveztünk és elvégeztünk méréseket, amiknek a célja a rendszer nagy számú szenzor melletti viselkedésének vizsgálata és az egyes komponensek skálázásának az adatok körülfordulási idejére gyakorolt hatásának elemzése volt. Ezek alapján ajánlásokat fogalmazunk meg a saját rendszerünkre, valamint hasonló rendszerek fejlesztésére vonatkozóan.

2. fejezet

Cloud-native rendszerekről

2.1. Felhő

A felhő alapú számítástechnikában [2] a felhasználó elől elrejtve, a szolgáltató erőforrás halmazán elosztva megvalósított szolgáltatásokat értjük, amit jellemzően virtualizációs technológiára építenek. Négy fő szolgáltatási modellt különböztetünk meg: SaaS [3] (Software as a Service, Szoftver, mint Szolgáltatás, például: Office 365), FaaS [4] (Function as a Service, Függvény, mint Szolgáltatás, például: Amazon Lambda), PaaS [5] (Platform as a Service, Platform, mint Szolgáltatás, például: Oracle Cloud Platform) és IaaS [6] (Infrastructure as a Service, Infrastruktúra, mint Szolgáltatás, például: Microsoft Azure).

2.2. Kubernetes

A Kubernetes egy Go nyelven írt, nyílt forráskódú konténer orkesztrációs platform, amely képes konténerek automatikus konfigurációjára, skálázására, valamint bizonyos hibák automatikus elhárítására is. Több konténer technológiát támogat, köztük a Dockert is. Nagyon népszerű, gyors fejlesztés alatt álló projekt. Emiatt felhasználói és programozói interfésze gyakran változik, megkövetelve a ráépülő megoldásoktól a hasonló sebességű fejlesztést. Jól definiált interfésze miatt sok ráépülő, azt kiegészítő projekt létezik. A Kubernetes kiváló keretet nyújt mikroszolgáltatás alapú alkalmazások fejlesztésére.

Egy Kubernetes klaszterben két típusú hosztgép lehet. Mindkettőből lehet több darab, de legalább egy-egy példány kötelező. Egy Kubernetes klaszter [7] legalább egy Masterből és Workerből épül fel, ezek egy-egy feladatot ellátó komponensekből állnak. A Kubernetes Master felelős a klaszterben lezajló folyamatok irányításáért, a Slave-ek vagy más néven Worker-ek, valamint az alkalmazások állapotának nyilvántartásáért. Egy Master node számos komponensből áll, ezek a Master egy-egy feladatáért felelnek. Több Master node futtatása esetén - úgynevezett multi-master mode - csak az API Server [8] és az etcd [9] komponensekből jön létre több példány, a többiből egyszerre csak egy példány lehet aktív. A Kubernetesben különböző objektumtípusokat definiáltak, ezek közül a legfontosabbakat bemutatjuk a következőkben (Pod, Deployment, Service és Ingress).

A Pod [10] egy vagy több konténert összefogó logikai hoszt. Egy podon belül lévő konténerek osztoznak a hálózaton és a háttértáron, valamint azonos a futtatási specifikációjuk. Ez azt jelenti, hogy az egy podon belüli konténerek localhost-on keresztül megtalálják egymást, valamint van lehetőség, hogy a konténerekben futó alkalmazások lássák a másik konténerben futó folyamatokat. Egy Kubernetes rendszerben a Pod a legkisebb egység, amit futásra lehet ütemezni.

A Deployment [11] segítségével deklaratívan leírható egy alkalmazást felépítő podok és azok kívánt állapota. Lehetőség van megadni, hány replikát hozzon létre a rendszer. Hasonló módon lehet a podok állapotát frissíteni vagy egy korábbi állapotra visszatérni. Egy Deploymentben lehetőség van több pod definiálására, ami az alkalmazás komponensek lazább csatolását teszi lehetővé.

A podok bármikor törlődhetnek, valamint új példány jöhet belőlük létre [12]. Minden pod saját IP címmel rendelkezik, viszont szükség van valamilyen módszerre, aminek segítségével nyomon lehet követni, hogy egy pod által nyújtott szolgáltatás milyen címen érhető el. Erre a problémára nyújt megoldást a Service, ami absztrakciót jelent a podok felett.

Néhány fontosabb szolgáltatás, melyet a Kubernetes nyújt:

- Horizontális skálázás,
- Konfiguráció és szenzitív adatok menedzsmentje,
- Háttértár orkesztráció,
- Szolgáltatások név alapján történő felderítése.

Ingress [13] erőforrás segítségével klaszteren belüli Service erőforrást lehet azon kívülre kiszolgálni. Ennek módját az Ingress erőforrás határozza meg, amelyet az Ingress Controller nevű klaszter szintű objektum szolgál ki.

2.3. Mikroszolgáltatások

A mikroszolgáltatás vagy mikroszolgáltatás [14] szoftverarchitektúra egy alkalmazás architektúra, amelynek segítségével a komplex, skálázható alkalmazások fejlesztése egyszerűbb, valamint a kódbázis növekedésével átlátható marad. Ezt úgy éri el, hogy az alkalmazást kisebb, önálló komponensekre bontja, ezeket lazán, tipikusan REST API-val, vagy üzenet sorok (message queue) segítségével illeszti egymáshoz. Az architektúra alkalmazása esetén felmerülnek bizonyos problémák [15], mint például az egyes szolgáltatásoknak meg kell egymást találniuk, vagy több példány futtatása esetén megoldandó a terheléselosztás [16] is. Az így fejlesztett alkalmazások kiválóan illeszkednek a felhő alapú rendszerekhez, például a Kuberneteshez. Mikroszolgáltatás architektúra esetében figyelni kell az úgynevezett rejtett monolitra, amikor a fejlesztett alkalmazás viselkedéséből adódóan igazából nem mikroszolgáltatás alapú. Például, ha minden mikroszolgáltatás függ egytől, akkor az adott architektúra rejtett monolit jellegű. Pozitív tulajdonsága a mikroszolgáltatásoknak,

hogy jól definiált API-k mellett az egyes szolgáltatások a saját adatszerkezetüket a nekik legmegfelelőbb módon képesek kezelni, nincs szükség kompromisszumokra az egész alkalmazást figyelembe véve.

2.4. Internet of Things

Az Internet of Things (IoT), vagy magyarul a Dolgok Internete mindennapi eszközök szenzorokkal és internet kapcsolattal ellátását [17] jelenti. Ezek az eszközök képesek automatikusan adatok gyűjtésére és azok továbbítására. Gyakran előfordul, hogy több típusú szenzorból gyűjtött adatok egy felhő [18] rendszerben kerülnek összegzésre és feldolgozásra, ugyanis at IoT eszközök tipikusan alacsony számítási kapacitással és fogyasztással rendelkeznek, ami az egyes eszközök árát is alacsonyan tartja. Jó példa az IoT-ben rejlő lehetőségekre egy olyan okos termosztát, amely az online elérhető időjárás előrejelzések és a felhőben tárolt felhasználói profil alapján előre melegíti vagy hűti az általa irányított lakást.

2.5. Mesterséges Intelligencia

Az elmúlt években, ahogy a technológia fejlődött, a mesterséges intelligencia és gépi tanulás eszköztárát segítségűl hívó megoldások is egyre inkább eljterjedtek. Ezek a megoldások lehetőséget adnak arra, hogy olyan problémákat is hatékonyan megoldjunk, amelyekhez eddig nagyon komplex algoritmusok vagy emberi beavatkozás volt szükséges. A mesterséges intelligencia önmagában egy nagyon nagy és szerteágazó tudományág [19]. Ennek egy kis szeletét, a gépi tanulást használtuk mi fel.

A gépi tanulás témaköre olyan rendszerekkel foglalkozik, amelyek valamilyen bemenet (tapasztalat) alapján valamilyen belső tudást képes alkotni és felhasználni. Ezt a gyakorlatban úgy kell elképzenlni, hogy egy adott tanuló adat alapján egy meghatározott algoritmus szerint matematikai modellt építenek. Ezt a folyamatot leegyszerűsítve ”tanulásnak” nevezzük. Ezt a modellt felhasználva a mesterséges intelligencia képes, eddig számára ismeretlen bemenetre is becsléseket alkotni és döntést hozni [20]. Mindezt anélkül, hogy erre explicit programozva lenne.

A munkánk célja nem a mesterséges intelligencia modellek finomítása, testreszabása, hanem a madárhang azonosítás komplex feladatának minden egyes komponensét integráltan kezelő rendszer hatékony, felhő-natív megvalósítása. Egy adott hangminta felismerére a technológia mai állása szerint az egyik legelterjedtebb megoldás a gépi tanulási módszer alkalmazása, a mi munkánkban ezért megjelent a gépi tanulás is, mint felhasznált technológia. Ugyanakkor ezt a módszertant nem mi alkalmaztuk (nem mi alkottuk meg a modellt, nem mi tanítottuk be), hanem egy kollégánk által elért eredményekre támaszkodtunk [21]. A kollégánkkal egyeztetve, a már megvalósított modellből kiindulva (lásd a Classification Service és Model Service mikroszolgálatásokat a 3.1 ábrán) mi készítettük az interfészeket, valamint megvalósítottuk a kapcsolatot a többi komponenssel. A fenti szempontok miatt ebben a részben nem fejtjük ki bővebben a mesterséges intelligencia tematikáját, az arra

kíváncsi olvasó az ebben az alfejezetben hivatkozott publikációkban megfelelő mélységű referenciákat találhat.

3. fejezet

A madárhang azonosító és riasztó felhő-natív rendszer ismertetése

A rendszer tervezését olyan alapvető felelősségekkel kezdtük, mint adatok fogadása, ezek tárolása, valamint továbbítása. A fejlesztési folyamat során ezektől haladtunk az egyre specifikusabbak fele, mint a mesterséges intelligencia által használt modellek tárolása és dinamikus frissítése. Ennek a folyamatnak eredményeképp készítettük el a a 3.1 ábrán látható architektúrát. A rendszer minden komponense egy-egy mikroszolgáltatás. Ezt a tervezési folyamat egy korai lépésében döntöttük el. E döntés mögött két indokunk volt. Az első a fejlesztés során a kooperáció könnyebbsége, hiszen a mikroszolgáltatások interfészeinek egyeztetése után azok egymástól függetlenül fejleszthetők, ezzel a kooperációt is elősegítve. Emellett ettől a döntéstől azt vártuk, hogy amennyiben a rendszer tervezése során követjük a korábban leírt ajánlásokat, a komponensek skálázhatósága jobb lesz, így a nagyobb terhelés esetén nem lesz érzékelhető növekedés a válaszidőben. Ebben a fejezetben szeretnénk bemutatni a hasonló, már létező ilyen megoldásokat és az általunk tervezett architektúrát a beérkező adatok útját bejárva.

3.1. Felhő-natív IoT rendszerek áttekintése

Számos a piacon is elérhető felhő alapú IoT szolgáltatás érhető el. Ezek közül egy az Azure IoT Hub [22], ami biztonságos kommunikációt biztosít az eszközök és a felhőben futó komponensek között. Ezek lehetnek saját fejlesztésű szoftverkomponensek vagy más Azure szolgáltatások, például Azure Stream Analytics [23], ami adatfolyamok gyors feldolgozására képes szolgáltatás, de számos más lehetőség van. Gyakorlatilag minden népszerű publikus felhőszolgáltatónál léteznek hasonló tudású szolgáltatások, ezek működése is tipikusan igen hasonlít egymásra. Ilyen előre elkészített szolgáltatások használatának előnye, hogy a fejlesztői és az ehhez tartozó mérnöki munkát már elvégezték és a szolgáltatás teljesítményével szemben bizonyos garanciát is gyakran vállalnak.

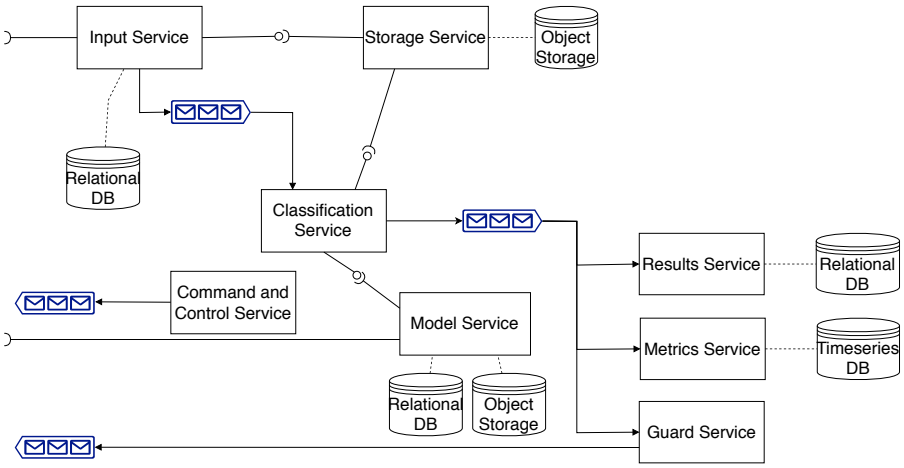
Fontos figyelembe venni viszont azt is, hogy az ilyen szolgáltatások használata esetén gyakran igen nehéz szolgáltatók között mozogni, különböző szolgáltatóktól keverni szolgáltatásokat pedig ennél is nehezebb. Ezt a jelenséget, amikor a szolgáltató a saját

termékcsaládjában próbálja tartani az előfizetőit vendor lock-innek hívjuk, ami az egyik legnagyobb visszatartó ereje a felhő alapú technológiák adopciójának az adatbiztonság és a kontroll elvesztése után [24].

Ezt elkerülhetjük nyílt forráskódú, valamint saját fejlesztésű komponensek használatával, viszont ennek a megközelítésnek hátránya, hogy a rendszert üzemeltetni, támogatni a fejlesztőnek kell. Egy ilyen nyílt forráskódú platform a Kubernetes is.

Kubernetes segítségével olyan IoT eszközöket támogató rendszerek [25] készíthetők, amik tradicionális alkalmazásoknál jobban skálázódnak, költséghatékonyabbak. Mesterséges intelligencia alkalmazása egy ilyen rendszerben egy újszerű, érdekes probléma, aminek nem egyértelmű a megoldása. Dolgozatunkban erre teszünk kísérletet.

3.2. Felhő-natív rendszerelemek ismertetése



3.1. ábra. A felhő-natív rendszer architektúrája

A rendszerbe két típusú adat érkezhet. A kihelyezett eszközök által felvett hangadatok a felhő rendszerbe az Input Service-hez kerülnek be, ahol a beérkezett hangfájl és az azt kísérő üzenet formátumának validálása után az alapvető metaadatok – többek között a küldő eszköz azonosítója, beérkezés dátuma, valamint az egész felhő rendszerben használt egyedi azonosító – eltárolódnak. Ezt követően az állomány továbbításra kerül a Storage Service-nek, ahol egy objektumtárban kerül eltárolásra. Ez azért előnyös, mert ha egy mintát szeretnénk többször is feldolgozni, a feldolgozott mintákat meghallgatni validálás céljából, vagy az AI Service-ben használt modellt a rendszer felhasználója pontosítani szeretné a rendelkezésre álló valós mintákkal, akkor lehetőség van azt lekérni egyéb metaadatai mellett az egyedi azonosító segítségével. Erre viszont csak limitált ideig van lehetőség, ugyanis az eltárolt hangfájlok beállítható idő múlva – alapértelmezés szerint egy hét – törlésre kerülnek, hogy a beérkezett minták által elfoglalt lemezterületet kordában

tartsuk. A fájl sikeres tárolását követően az Input Service egy üzenetsorra beküldi a minta azonosítóját, ezzel jelezve a feldolgozó komponensnek az új minta beérkezését és azt, hogy milyen azonosítót használhat a Storage Service-től lekérdezéskor.

Az üzenetsoron az AI Service példányai fogadják az üzenetet, ezen komponens felelőssége a minták klasszifikálása többek között seregély, traktor és egyéb zaj kategóriákba. Még mielőtt megkezdhetné a feldolgozást, a Model Service-nél ellenőrzi, hogy elérhető-e újabb modell, amivel dolgozni tud. A Model Service képes több különböző típusú – beleértve a felhő rendszerben és az IoT eszközön található mesterséges intelligenciák által használtakat – modellt tárolni, a modellek verzióját követni, valamint visszaadni REST API-n keresztül. Az éppen használt modell friss állapotának eldöntésére lekéri az alapértelmezett modell metaadatait, amiket összevet a lokálisan meglévő modell metaadataival. Amennyiben valóban elérhető újabb modell, letölti azt és onnan betölti további használatra. E mechanizmus által lehetőség van a rendszer által használt modellek dinamikus frissítésére és cseréjére, valamint a Model Service-ben az alapértelmezett modell korábbira visszaállításával akár korábbira visszatérés is könnyedén lehetséges. A modell ellenőrzése után az AI Service-ben található mesterséges intelligencia elvégzi a hangfájl klasszifikációját. Ennek eredményét a minta azonosítójával egyetemben egy másik üzenetsoron publikálja.

Az üzenetsorra érkező adatokat három mikroszolgáltatás fogadja, de az üzenetsornak hála újabb ilyen komponens illesztése a meglévő a rendszerhez könnyedén lehetséges. A Results Service feladata eltárolni minden egyes minta adatait úgy, hogy azok egyesével lekérdezhetők legyenek. Ennek a mikroszolgáltatásnak a segítségével lehetséges harmadik fél által gyártott szoftverekhez csatolni az általunk fejlesztett rendszert. A Metrics Service felelőssége idősoros adatbázisba illeszteni az eredményeket. Az idősor adatbázisból statisztikai lekérdezések segítségével betekintést nyerhetünk a szőlőben a seregélyek mozgásába, tendenciáiba akár egy harmadik fél által fejlesztett dashboarddal. Az utolsó mikroszolgáltatás, ami megkapja a klasszifikációk eredményeit a Guard Service. Ez a mikroszolgáltatás üzenetsoron küld egy riasztási parancsot annak az eszköznek, ahonnan seregélyként detektált minta érkezett. Az egyes komponensek közötti kommunikáció alapvetően HTTP feletti REST API-k segítségével történik, kivéve az előzőekben említett helyeken, ahol üzenetsorral. Előbbit olyan esetekben alkalmaztuk, amikor az adott interfész külső alkalmazások felé elérhető, nagy mennyiségű adat átvitele történik, vagy fontos a szinkron kommunikáció. Jó példa a Storage Service interfésze, ugyanis itt fontos tudni, hogy mikor áll készen a mikroszolgáltatás a minta kiszolgálására, valamint egy tipikus JSON formátumú üzenethez képest nagy mennyiségű, bináris adat átvitelét jelenti. Természetesen üzenetsorra is megvalósítható lenne ez a viselkedés, viszont jelentősen bonyolultabb üzleti logikát igényelt volna a megvalósítás, jelentősebb előnyt viszont nem hozna. Az egyes mikroszolgáltatások közötti üzenetsorok AMQP (Advanced Message Queuing Protocol) felett zajlanak. Emellett akkor döntöttünk, amikor több komponensnek is szüksége van egy művelet eredményére – mint például az AI Service kimenetére reagáló három mikroszolgáltatás –, vagy nem szükséges az adott üzenetre válasznak érkeznie. Ilyen az Input Service által az AI Service-nek küldött értesítés is. Az interfészek és üzenetek tervezése során kiemelten figyeltünk arra, hogy minden üzenetsoron és REST API-n átvitt

adat lehetőleg minél kisebb legyen, csak a szükséges adatok kerüljenek továbbításra. Ezzel is növelve a kommunikáció hatékonyságát és kihasználva a HTTP és üzenetsor előnyeit minimális többletköltség mellett.

A minták fogadásában nem vesz részt a Command and Control Service, felelőssége az egyes eszközök és azok szenzorainak állapot vezérlése, nyomon követése. A kihelyezett IoT eszközök az állapotukat üzenetsoron jelzik. Egy ilyen üzenet például lehet jelzés, hogy az eszköz valamilyen hibás állapotba került és javítást vagy cserét igényel, esetleg egy szenzorról nem képes adatot olvasni, kikapcsolni készül vagy épp most kapcsolt be. Egy másik üzenetsoron hallgatják a parancsokat és végrehajtják azokat. Ilyen parancs lehet a mesterséges intelligencia által használt modell frissítése és a szenzor vagy eszköz kikapcsolása. Az irányító üzenetek küldését REST API segítségével lehet kiváltani. A Command and Control Service indulásakor nem ismeri a rendszerben futó eszközöket, azokat az első üzenetük fogadásakor jegyzi fel egy saját relációs adatbázisba, ahol követi az állapotukat. Azokat az eszközöket, amik öt perce nem küldtek státusz üzenetet, a mikroszolgáltatás hibás állapotba lépteti a belső állapotreprezentációjában. Az egy hétnél tovább el nem érhető eszközöket automatikusan kikerülnek az adatbázisból kivéve, ha a felhasználó API hívás segítségével permanensnek nem jelölte azt.

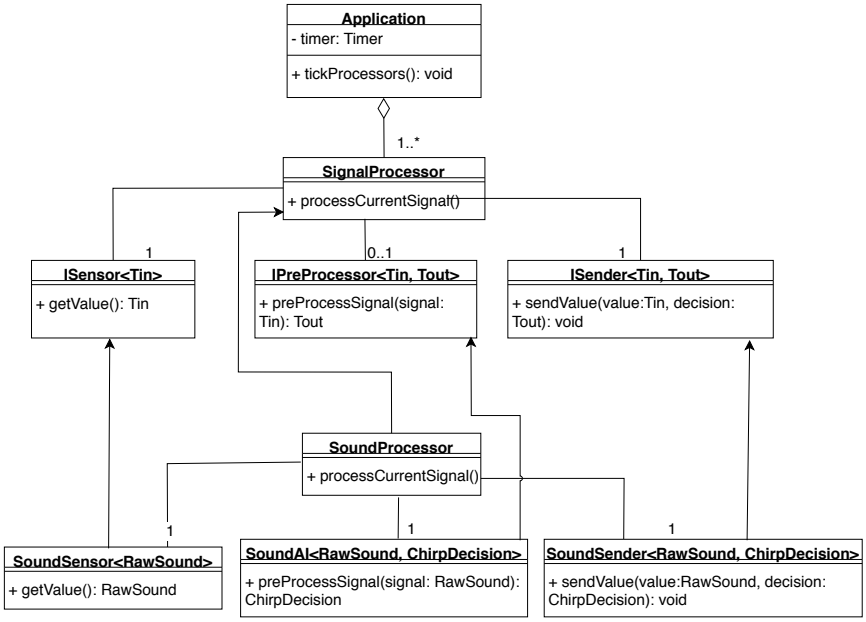
Az eszközök és a felhőben lévő komponensek közti üzenetsoros kommunikáció MQTT (Message Queue Telemetry Transport) protokollon történik, ami mellett annak flexibilis téma (topic) architektúrája miatt döntöttünk. Lehetőség van az egyes üzenetsorok, vagy másnéven témák, struktúrálására és ezáltal egyszere több témára üzenet küldésére, valamint feliratkozásra. Ezt a Command and Control Service esetében ki is használtuk, ugyanis az egyszere iratkozik fel minden eszköz státuszüzeneteire, valamint a struktúrált témáknak köszönhetően minden vezérlőüzenetet csak a céleszköz kap meg, ami csökkenti a szükséges hálózati forgalmat és az eszközök által feldolgozandó üzenetek számát.

3.3. Internet of Things eszköz ismertetése

Az IoT eszköz architektúrájának tervezése során a legfontosabb szempont az új szenzorok bevezetése volt. Emiatt úgy döntöttünk, hogy az egyes szenzorokat teljesen külön kezeljük. Három alapvető lépést határoztunk meg egy mért adat útja során. Az első a szenzor általi mérés elvégzése, amit az opcionális előfeldolgozás követ. Végezetül, a szoftvernek döntést kell hoznia, hogy az eredményt tovább küldje-e felhőbe, majd a döntést végre is kell hajtsa.

Azért, hogy ez a folyamat minél flexibilisebb legyen, egy külön osztály fogja össze és indítja az egyes lépéseket, valamint a mintavételezéseket egy központi, belső óra vezérli. Itt konkrét, hardveres implementációval nem foglalkoztunk, azt a szoftver tervezése és fejlesztése során elabsztracháltnak tekintettük.

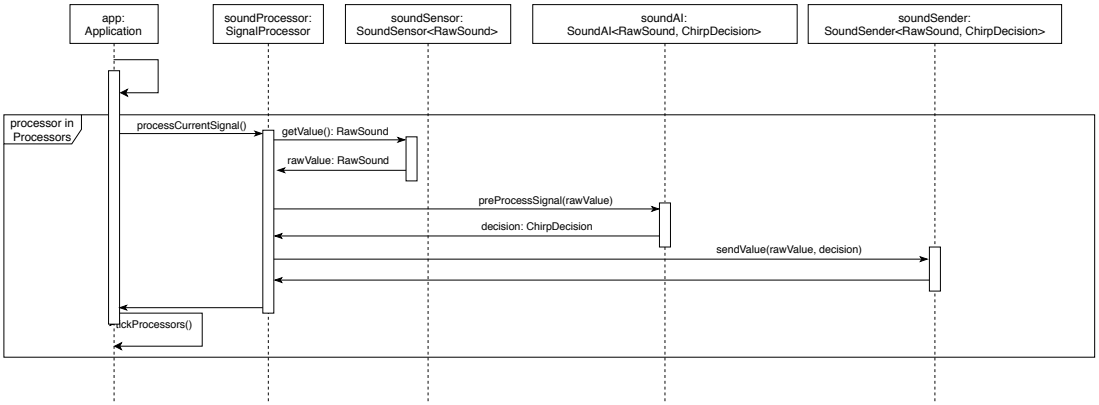
A felvázolt koncepciót a fenti osztálydiagramon valósítottuk meg. Új szenzor felvétele esetén az ISensor, IPreProcessor, ISender interfészek implementálásával és SignalProcessor osztályból leszármazással, valamint ezek az Application osztályba regisztrálásával lehetséges. Új riasztásra képes forgatókönyv esetében az IActuator interfész implementálásával van lehetőségünk a riasztás elvégzésére.



3.2. ábra. Az IoT eszközön futó szoftver osztálydiagrammja

Az Application osztály tartalmazza az eszközök eseményhurkát, ami lehetővé teszi a szenzorok ismételt mintavételét, előfeldolgozását és a minta felhőbe küldését. Az általunk megvalósított viselkedés során egy ütemben a SoundProcessor vezényli a hangminták kezelését. Az ütem elején egy másodperc hosszú hangmintát vesz az előre bekonfigurált mikrofonról, amit egy mesterséges intelligencia bekategorizál többek között traktor, emberhang, méhek és madárcsicsergés kategóriákba. A madárcsiripelés kategóriába sorolt mintákat továbbítja a felhőbe a SoundSender, ahol az előbbiekben leírt folyamat kerül végrehajtásra. Ez a forgatókönyv a 3.3 ábrán látható. Minden harmadik ütem végén küld az eszköz státusz frissítést a felhőnek.

Az eszköz indulásakor az Application osztály a beállított üzenetsoron elküldi, hogy éppen bekapcsolt, hamarosan elkezd adatokat küldeni és képes parancsok végrehajtására. Utóbbi külön szálon történik, emiatt szükség volt a szenzorok offline és online állapotainak szálbiztos kezelésére a SignalProcessor leszármazottjaiban és az ezeket nyilvántartó listának is hasonlóan szálbiztosnak kell lennie. Az offline szenzorok forgatókönyve kikerül az ütemezett forgatókönyvek közül, viszont futó forgatókönyvet az alkalmazás nem szakít meg.



3.3. ábra. Az IoT eszköz egy futásának szekvenciája

4. fejezet

A rendszer teljesítménymérése

Dolgozatunk e fejezetében bemutatunk egy olyan módszertant és eszközöket, amelyek segítségével egy komplex mikroszolgáltatásokból álló rendszerben megtalálhatók az esetleges gócpontok, valamint azok orvoslására javaslat tehető. Célunk egy olyan eljárás kidolgozása volt, amely a mikroszolgáltatás architektúrában fejlesztett szoftverek esetében gyakran előforduló szűk keresztmetszeteket a DevOps eszköztárával agilisan képes felderíteni és iteratívan javítani rajta. Emellett bemutatjuk az ennek kidolgozásához és ellenőrzéséhez felhasznált eszközöket is.

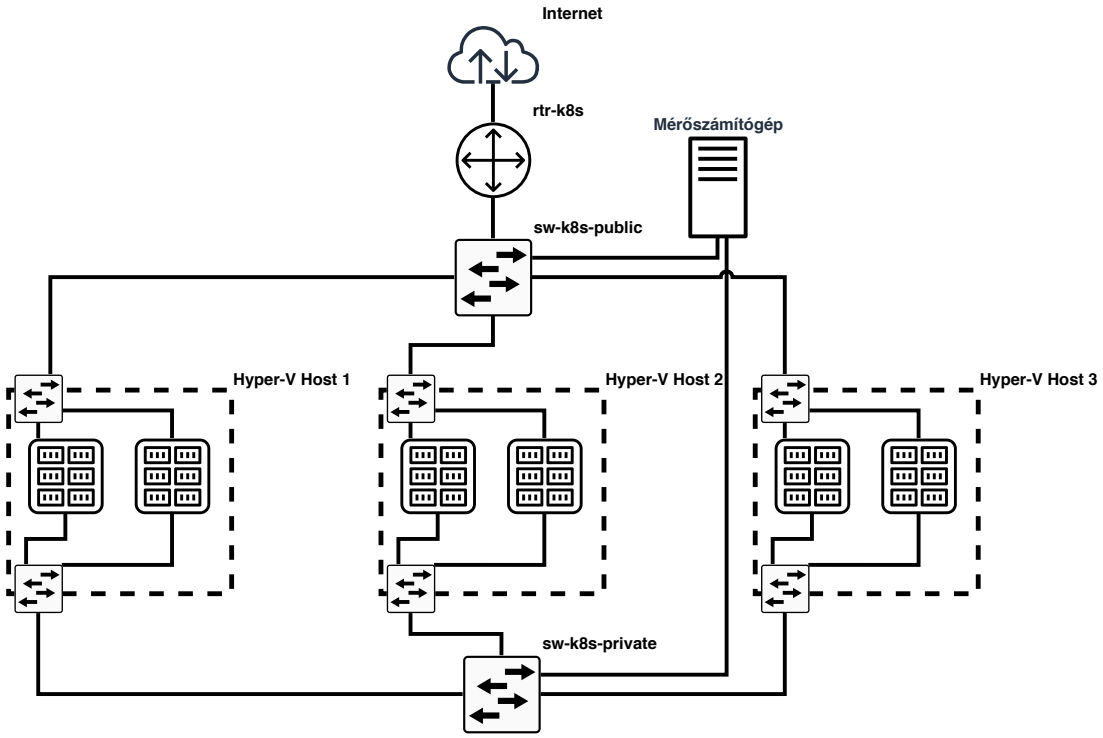
4.1. Mérési környezet ismertetése

A fejlesztés és teljesítménymérés idejére létrehoztunk egy Kubernetes klasztert, amiben futtattuk a rendszert. A klaszterben három Master és ugyanennyi Worker volt, mindegyik virtuális gépen futott, Microsoft Hyper-V hypervisorban. A Mastereknek kettő processzormagja és nyolc gigabájt memóriája volt, a Workereknek négy magot, valamint tizenhat gigabájt memóriát allokáltunk. A klaszter három fizikai gépen futott, mindegyikben Intel Xeon E5-2450L processzor, valamint 64 gigabájt memória volt. Egy Master és egy Worker példányt telepítettünk mindegyik fizikai gépre.

A fizikai hosztgépek két egy gigabites Ethernet hálózaton kerültek összekötésre. Amint az a 4.1 ábrán is látható, az egyik hálózaton a hypervisorok és a virtuális gépek az internetet érték el, ezt csak frissítések és csomagok letöltésére használtuk. Emellett egy fizikailag teljesen különálló hálózat állt a virtuális gépek rendelkezésére a klaszter forgalmának lebonyolítására.

Mivel a fürtnek számos tagja volt, telepítettünk egy külön fizikai gépre egy HAProxy HTTP és TCP terheléselosztót, ami a kéréseket roundrobin algoritmussal osztotta el az egyes node-ok között. Ennek segítségével volt elérhető az egyes mikroszolgáltatások API-ja és az MQTT broker az IoT eszközök számára. Ebben a fizikai gépben egy Intel Core i5 2550K és 16 gigabájt ram volt.

A Kubernetes klaszterben a Calico hálózati implementációt használtuk, ami nem hoz létre virtuális hálózatot a meglévő fölé, hanem az egyes alkalmazások és kompnensek között különálló IP alhálózatok és útvonalválasztás segítségével biztosítja a megfelelő izolációt.



4.1. ábra. A felhő rendszer architektúrája

Emellett NGINX Ingress Controllert használtunk. Ez egy NGINX webszerver konfigurációjával realizálja az Ingress Controller funkcionalitását. A mikroszolgáltatásokat kiszolgáló relációs adatbázisokat egy külön virtuális gépen futó PostgreSQL szolgálta ki, aminek 2 processzormagot és két gigabájt memóriát allokáltunk.

A méréseket futtató számítógépben egy Core i7 3770 és 16 gigabájt ram volt, valamint csatlakozott mind a publikus, mint a magánhálózatra.

4.2. Mérőszoftver ismertetése

Mint azt az előző fejezetben ismertettük a leghosszabb út, amit egy minta és az arra generált válasz megtehet, egy seregélyként kategorizált minta esetében fordul elő. Ebből következik, hogy egy kritikus mérés a beküldött HTTP kérések és az érkező MQTT üzenetek között eltelt idő lehet. Olyan kész mérőeszköz viszont, ami ennek a mérésére képes nem elérhető, ezért egy saját mérőeszköz fejlesztése mellett döntöttünk.

Annak érdekében, hogy a lehető leggyorsabban legyen képes a szoftver kéréseket generálni, indulás után először kigenerálja azok törzsét az Input Service elvárt sémában található eszközazonosító kitöltésével és a séma egyéb mezőit és a multipart kérés más részeit előre beállított sablont érintetlenül hagyásával. Parancssori paraméterként megadható a használni kívánt konkurenciaszám, ennyi külön folyamat fog futni. Az egyes folyamatok a következő lépésben egy saját listában nyilvántartják az aktuális rendszeridőt, elküldenek egy HTTP kérést az előre elkészítettek közül, majd egy másik listába is feljegyzik a rend-

szeridő aktuális értékét. Fontos kiemelni, hogy az egyes folyamatok minden kérés végéig blokkolnak, azaz megvárják, amíg megérkezik a válasz. Ezt a lépést addig ismétlik, amíg a mérést felügyelő folyamat – az előre beállított idő leteltével – le nem állítja őket. Ez alatt és után 600 másodpercig, szintén külön folyamatban, a szoftver fogadja az MQTT-n érkező üzeneteket és feljegyzi azok érkezési idejét egy listába.

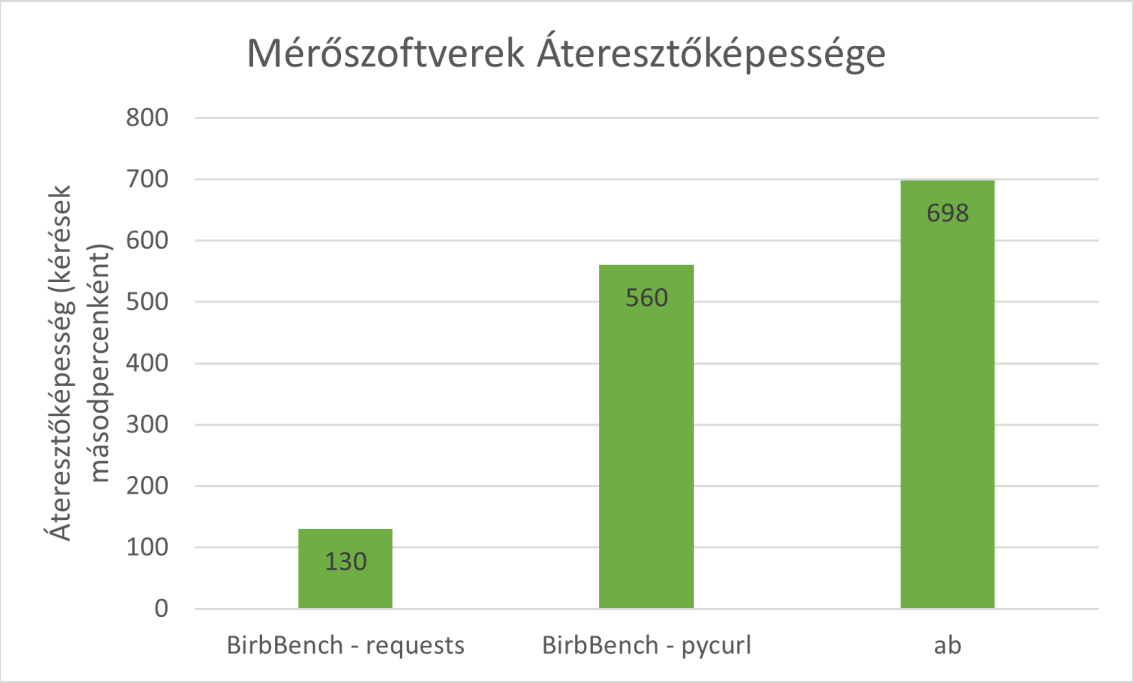
Miután minden folyamat végzett, a különböző HTTP kéréseket generáló folyamatoktól lekérdezi a küldési és fogadási időpontokat tartalmazó listákat és egyesíti azokat egy-egy nagy listába, amik a kérések elküldése szerint kerülnek rendezésre. Ez úgy lehetséges, hogy a szoftver minden kéréshez rendel egy egyedi azonosítót, amit a listában az időbélyeg mellé csatol. Ugyanez teszi lehetővé az MQTT üzenetek HTTP kérésekhez párosítását, ugyanis a mérni kívánt rendszer a bemeneti API-ján elfogad egy egyedi azonosítót, amit felhasznál az MQTT téma összeállítása során. Amennyiben a mérések során minden esetben egyedi azonosítóval látunk el egy-egy kérést, azok végig egyértelműen azonosíthatók maradnak, lehetővé téve az egyes mérőkomponensek külön szálakra bontását.

4.3. Mérőszoftver értékelése

Amennyiben a mérőszoftver segítségével méréseket futtatnánk, a kapott eredmények csekély információt hordoznának egyéb adatok híján, ugyanis bizonytalan lenne, hogy a mérés a mért rendszer vagy a mérőeszköz legjobb teljesítményét mutatja, nem tudjuk melyik teljesítményét mértük ki. A mérőszoftvert Pythonban készítettük, ezért annak teljesítménye nem feltétlen haladja meg a mért rendszerét.

Ezt a problémát megoldandó elkészítettünk egy a rendszert imitáló webes alkalmazást. Az alkalmazást Kotlinban írtuk, amiben a korutinok segítségével könnyű aszinkron, nem blokkoló kód írása. Ennek köszönhetően az alkalmazás minden HTTP kérést aszinkron módon szolgál ki, ezzel emelve az alkalmazás áteresztőképességét. Amennyiben az alkalmazáshoz GET kérés érkezik, az egy nullás karakterrel tér vissza. Ezt a viselkedést a mérő alkalmazás működésének tesztelésére hoztuk létre. Ha az alkalmazáshoz POST kérés érkezik, ellenőrzi, hogy a kérés megfelel-e az Input Service által elvártaknak. Amennyiben nem, 500-as hibakóddal válaszol. Amennyiben a kérés átmegy a validáción, a kérés törzséből beolvassa az eszközt – vagy ez esetben a kérést – azonosító mező értékét és aszinkron módon küldd egy üzenetet az MQTT brókernek továbbításra a megfelelő témára, majd válaszol a kérésre egy húsz karakter hosszú karakterlánccal. Ez utóbbi célja, hogy a HTTP válasz mérete is nagyságrendileg akkora legyen, mint amekkorát az Input Service küldene. Ennek az alkalmazásnak teszteltük a teljesítményét referenciaként az Apache Benchmark (ab) eszközzel.

Az a 4.2 ábrán is látható, hogy az ab (ab oszlop) és az általunk készített szoftver (BirbBench – requests oszlop) nagy a különbség teljesítmény téren. Ezt a különbséget nem tartottuk elfogadhatónak. A gyanú gyorsan az általunk használt Python HTTP kliensre, a requests-re terelődött, ugyanis az teljes egészében Pythonban készült, valamint a PyCurlhöz viszonyítva az összehasonlító teljesítménymérésekben [26] lemarad. Utóbbi egy vékony Python réteg a C-ben implementált libcurl felett. Valóban, a PyCurl használata



4.2. ábra. Mikroszolgáltatások késleltetésének összehasonlítása

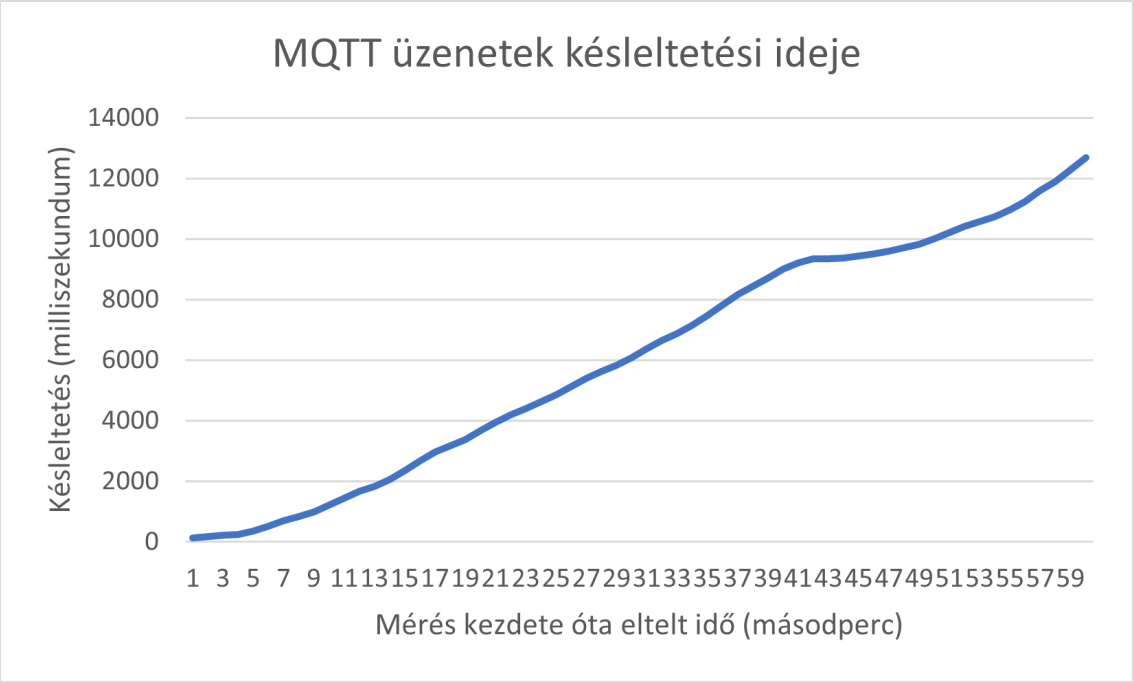
esetén a mérőeszköz által mért áteresztőképesség jobban megközelítette az ab által elértet, de nem érte azt el, viszont ez nem volt célunk. Ezen mérések által van referenciánk arról, milyen karakterisztikákkal rendelkezik a mérőszoftverünk HTTP kliense.

Az MQTT komponens értékelése során is nem várt anomáliát figyeltünk meg. Amint az a 4.3 ábrán megfigyelhető, az MQTT üzenetek késleltetési ideje a HTTP kérésekhez képest a mérés során szigorúan monoton növekedett, ami arra enged következtetni, hogy vagy a broker áteresztőképességénél nagyobb rátával küldi neki az üzeneteket a Kotlinos alkalmazás, vagy a kliens képes túl lassan fogadni az üzeneteket. Előbbire enged következtetni, hogy a broker folyamata a Workeren egy processzormagot teljesen kihasznál.

A fenti hipotézis ellenőrzésére elkészítettünk egy olyan Kotlin programot, ami letárolja az egy másodperc alatt érkezett üzeneteket, majd ezt összehasonlítottuk a Pythonban készült mérőszoftverrel. Az összehasonlítás során Ezek alapján arra következtettünk , hogy az általunk használt Apache Artemis MQTT broker egy példányt használva a számunkra elérhető hardveren körülbelül száz egyedi témára érkező üzenetet képes továbbítani.

4.4. Skálázódási lehetőségek felmérése

Méréseinkkel a rendszer skálázhatóságát szerettük volna megvizsgálni, viszont ehhez tudnunk kellett, mely komponenseket érdemes skálázni, melyek azok a mikroszolgáltatások, amk egy adott kérést a legtovább dolgozzák fel, ezzel a rendszerben szűk keresztmetszetként viselkednek. Mivel a mikroszolgáltatásaink egy távoli Kubernetes klaszterben futnak a tradicionális, fejlesztői eszközökbe épített profilozó használata nem lehetséges. Ezt a problémát hivatottak megoldani az Alkalmazás Teljesítmény Monitorozó [27] eszközök. Számos ilyen eszköz elérhető, mint például a Jaeger, vagy a mikroszolgáltatásokban kelet-



4.3. ábra. MQTT üzenetek késleltetése

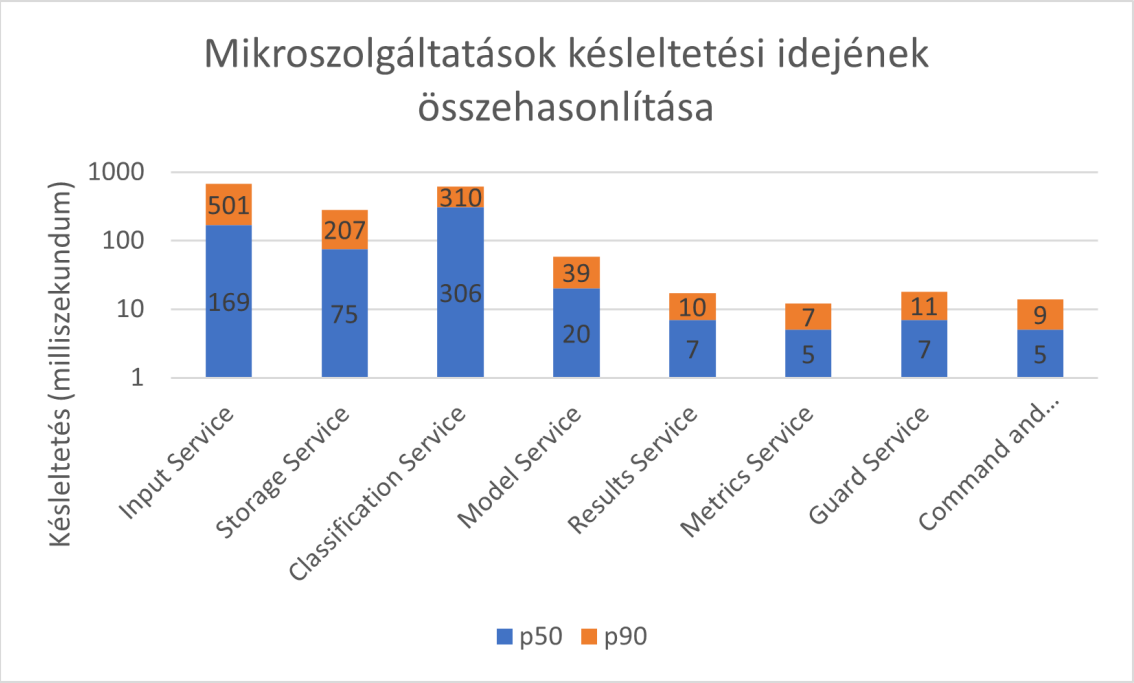
kezett hibákat is gyűjteni képes Sentry. Ezek között a különbség tipikusan a támogatott nyelvek és a szerverben használt technológiákban rejlik, mindegyik képes egy megjelölt kódblokk futási idejét monitorozni. A projekt fejlesztése során Sentryt használtunk az elosztott hibakeresési képességei, valamint elsőrangú Python támogatása miatt.

Ezen vizsgálatok során a módszertanunk az volt, hogy a rendszerbe beküldtünk egyetlen mintát, valamint a korábban ismertetett mérőeszközzel generáltunk nagyobb terhelést a rendszerben, majd a Sentry által aggregált adatokat összevetettük és értkeltük. Minden mérést húsz alkalommal ismételtünk és az eredmények átlagát vettük annak érdekében, hogy egy-egy kiemelkedő mérés ne befolyásolja aránytalanul a kapott eredményeket.

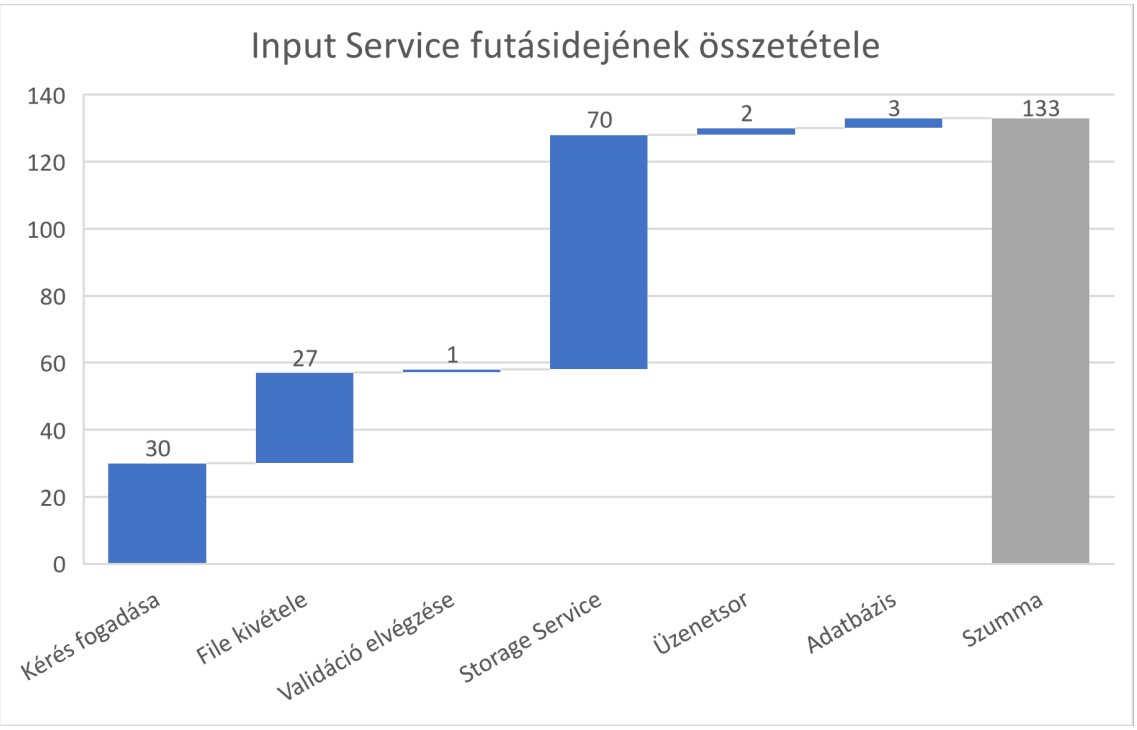
Általánosan elmondható volt a tesztek alapján, hogy a hangmintákkal dolgozó, valamint mesterséges intelligenciát alkalmazó mikroszolgáltatások válaszideje nagyságrendekkel nagyobb, mint a csak szöveges adatokkal dolgozóké. Ennek oka valószínű az Input és a Storage mikroszolgáltatások esetében a HTTP kérések mérete lehet, a Classification Service-nél pedig a mesterséges intelligencia futási idejéből adódhat a magas átfutási idő. Ezt ellenőrizendő finomítottuk a használt tranzakciókat, hogy a kérdéses HTTP kérések során elvégzett műveletek ideje pontosan látszódjon.

Ez alapján (ahogy az a 4.5 ábrán is látható) megállapítható, hogy valóban a kérés fogadása, valamint a kezelése tart sokáig az Input Service-ben, az egyéb műveletek rövid idő alatt lezajlanak. Ennek okán az egyik skálázásra kijelölt komponenspár az Input és Storage Service-ek voltak.

Úgy döntöttünk ezeket közösen érdemes skálázni, hiszen a a 4.5 ábrát összevetve a a 4.6 ábrával is megfigyelhető, hogy a Storage Service esetében a HTTP kérés fogadásával és a hangfájl multipart kérésből kivételével eltöltött idő igen közel van az Input Service-nél tapasztaltakhoz. Ez egy várható eredmény volt, ugyanis mindegyik mikroszolgáltatás



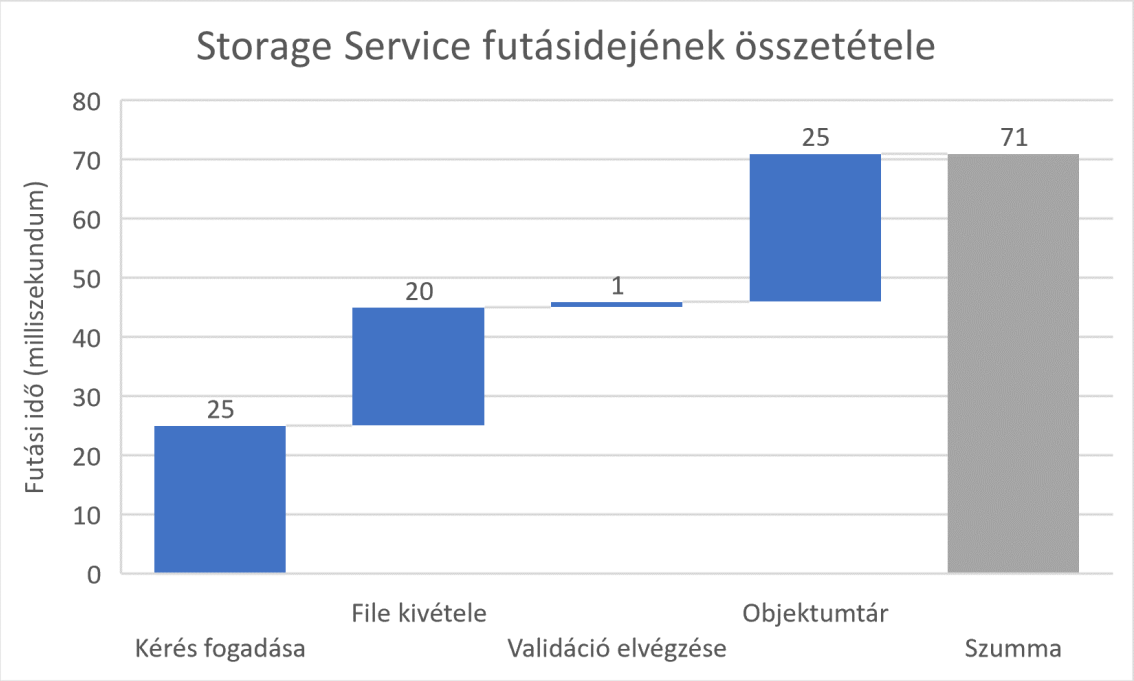
4.4. ábra. Mikroszolgáltatások késleltetésének összehasonlítása



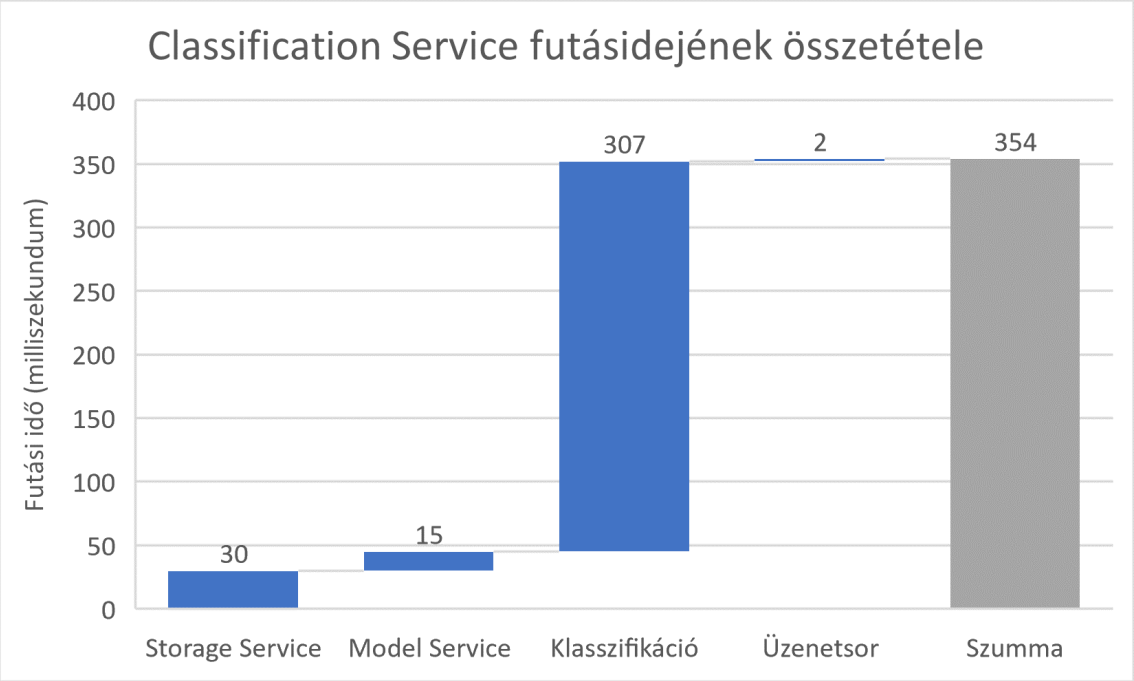
4.5. ábra. Input Service futási idejének összetétele

Python nyelven, a Flask webes keretrendszert használva készült, így ezen műveletek implementációja és belső viselkedése nagyon hasonló.

A Classification Service-t górcső alá véve a korábbi hipotézisünk beigazolódni látszik, ugyanis a minta feldolgozásával eltöltött idő közel 85%-át tölti a mesterséges intelligencia



4.6. ábra. Storage Service futási idejének összetétele



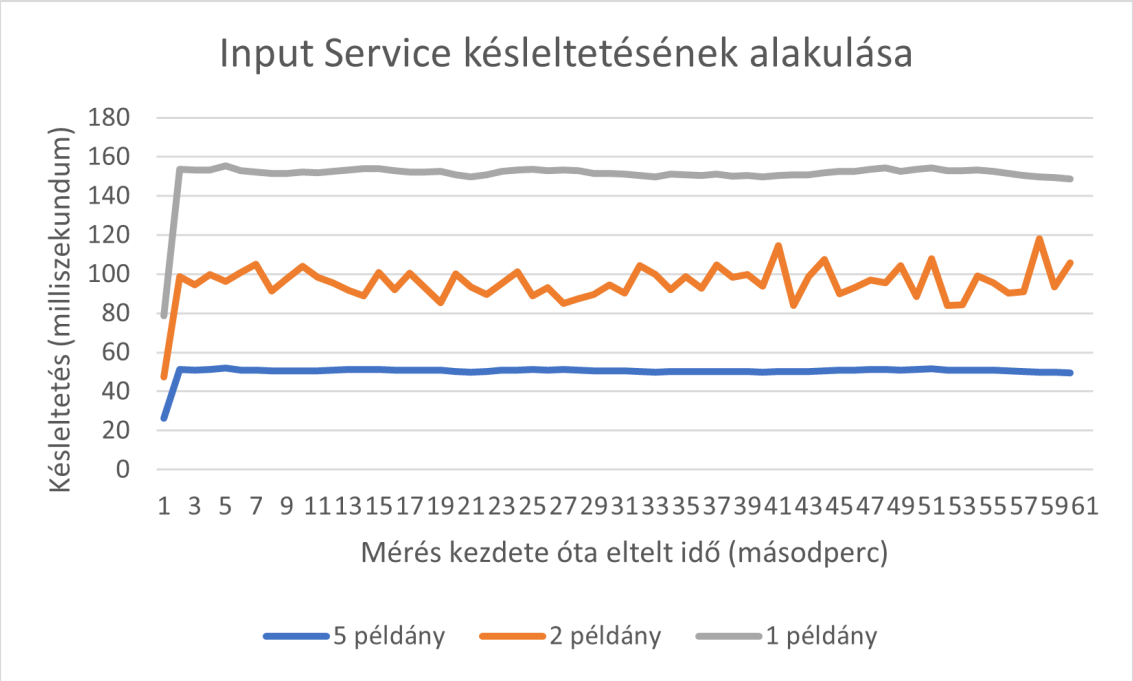
4.7. ábra. Classification Service futási idejének összetétele

futtatásával a mikroszolgáltatás. Ez azt jelenti, hogy nagy számú egyidejű beérkező minta esetén mindenképpen skálázni kell ezt a mikroszolgáltatást is.

A többi mikroszolgáltatás esetében nem tartottuk indokoltnak a skálázás alkalmazását, ugyanis azok nagy számú konkurens kliens esetében is alacsony válaszidővel képesek voltak feldolgozni az egyes kéréseket, így azok a rendszer áteresztőképességét nem korlátozzák.

4.5. Input és Storage Service-ek skálázásának vizsgálata

Száz konkurens kliens esetén az Input Service késleltetése nem növekszik meg lényegesen az egy, vagy húsz konkurens kliensek esetén mérttől, amint azt a 4.8 ábrán is láthatjuk. Ez jó jel a skálázhatóságára tekintve, ugyanis ebből arra következtethetünk, hogy ez a mikroszolgáltatás kiszámíthatóan reagál a változatos terhelésekre. Ez alapján megsejthetjük, hogy amennyiben kettő példányt indítunk a komponensből annak áteresztőképessége közel kétszeresére növekedik. Ehhez viszont ki kell kössük, hogy a Storage Service skálázása is szükséges, ellenben az továbbra is szűk keresztmetszet marad.



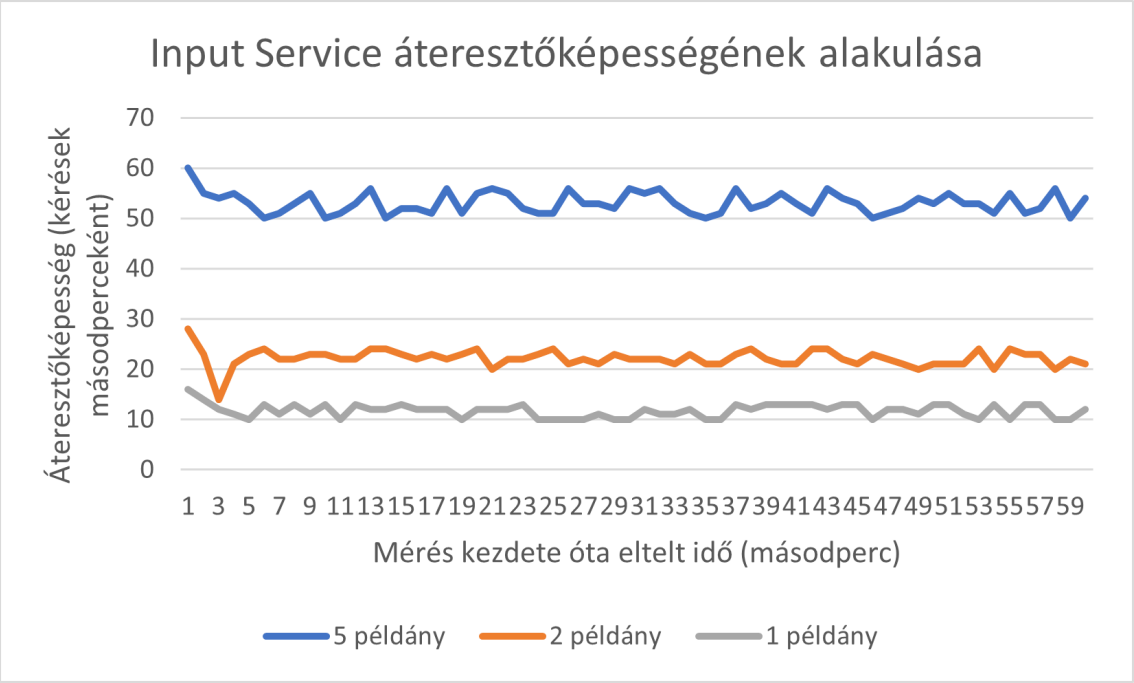
4.8. ábra. Input Service késleltetésének alakulása

A sejtés beigazolódni látszik, ugyanis míg ez a két mikroszolgáltatás egy-egy példányt használtva tíz kérést volt képes feldolgozni másodpercenként, kettő példányt használva húszat, öt esetében pedig negyvennyolcat.

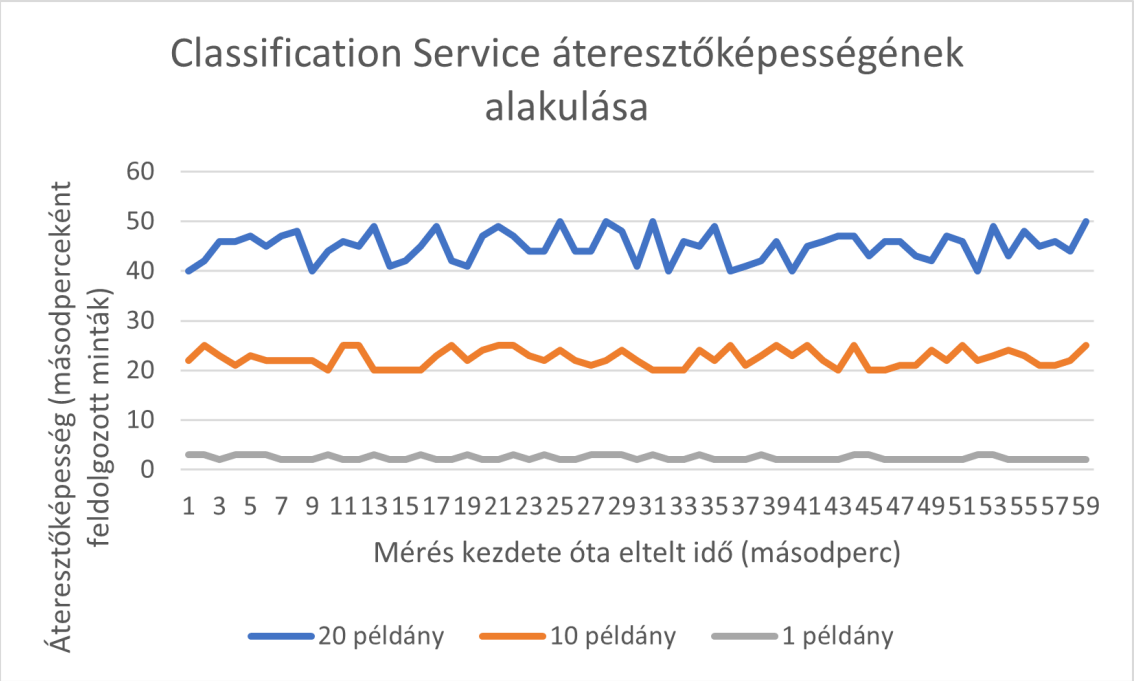
4.6. Classification Service skálázásának vizsgálata

A Classification Service bemeneti interfésze, mivel üzenetsoron kapja a bemeneti adatokat más jellegű, mint az Input vagy a Storage Service-eké. A teljesen aszinkron bemeneti interfészén csak akkor kap adatot, ha épp nem dolgoz fel egy másikat, valamint az aszinkron kimeneti interfésze miatt nem blokkolja másik mikroszolgáltatás sem. Emiatt az üzenetsor egyfajta terheléselosztóként is felfogható.

A fent leírtak alapján azt várjuk, hogy a mikroszolgáltatás skálázása esetén a komponens áteresztőképessége a replikák számával arányosan fog nőni. Amint ez a 4.10 ábrán látható, ez így van, amíg egy példány két mintát képes másodpercenként feldolgozni, tíz példány húszat, húsz példány pedig negyvenet.



4.9. ábra. Input Service áteresztőképességének alakulása



4.10. ábra. Classification Service áteresztőképességének alakulása

4.7. Eredmények értékelése

A fejezetben ismertetett mérések eredményeiből számos konzekvencia levonható a rendszerrel kapcsolatban. Az első, hogy a teljesítmény szempontjából szűk keresztmetszetet képző komponensek jól skálázódnak. A szűk keresztmetszet megszüntetésére vagy enyhítésére viszont egyéb alternatívák is láthatók. Az Input és Storage Service-ek vizsgálata

esetén érdemes lehet meggondolni azok összevonását, ugyanis ezzel a lépéssel az Input Service válaszideje érdemben csökkenthető. Az eredeti 130 milliszekundum válaszidőből ezzel megtakarítható a kérés fogadására és a file kivételére fordított idő, ezáltal körülbelül 50 milliszekundummal kevesebb lenne az adott mikroszolgáltatás futási ideje. Ezen túl a komponens skálázásával jól növelhető az áteresztőképessége.

A Classification Service esetében javasoljuk a Kubenetesben egyedi metrika alapján történő automatikus skálázás alkalmazását, ugyanis egy esetleges rövid idő alatt nagy számú beérkező hangminta esetében nehéz előrejelezni, hogy hány példány lenne azt képes gyorsan feldolgozni. A skálázáshoz szükséges egyedi metrikákat kellene megfogalmazni, például az üzenetsoron egységnyi idő alatt beérkező üzenetek számát, és ez alapján lehet a skálázási logikát programozni. Egy ilyen megoldás nem csak a szűk keresztmetszet problémáját kezeli, hanem elkerüli a klaszter erőforrás pazarálását is, mert az automatikus skálázó logika terhelésmentes időszakban leállítja a felesleges példányokat. Ennek a skálázódásnak implementációjánál viszont figyelni kell arra, hogy az új példányok az első minta feldolgozása során le kell kérjék az aktuális modellt a Model Service-től, ami késleletetésben és adatforgalomban mérhető extra költséget jelent.

4.8. A teljesítménymérési módszer alkalmazása általános rendszerekre

A dolgozatunkban bemutatott módszer jól felhasználható más hasonló mikroszolgáltatásokra épülő szoftverek esetében is.

Első lépésként valamilyen Alkalmazás Teljesítmény Monitorozó megoldás használata és integrálása kulcsfontosságú. Ezek segítségével megállapítható, hogy mely mikroszolgáltatások vizsgálata szükséges skálázási szempontból, melyek jelenthetik a szűk keresztmetszetet. Természetesen a skálázás előfeltétele, hogy a mikroszolgáltatásaink állapotmentesek legyenek. A szűk keresztmetszetet alkotó mikroszolgáltatások azok lehetnek, melyek legalább egy nagyságrenddel nagyobb idő alatt dolgoznak fel egységnyi kérést a rendszerben, ezek lehetnek érdemesek további vizsgálatra.

Második lépésként meg kell vizsgáljuk a kérdéses komponensek miként viselkednek skálázás esetében. Ezt terheléstesztek segítségével érhetjük el. Amennyiben egy komponens nem skálázódik jól, annak refaktorálására lehet szükség.

Az itt leírt tesztek segítségével elég adatot gyűjthetünk ahhoz, hogy javaslatokat tegyünk a rendszerben esetlegesen szükséges változtatásokra.

5. fejezet

Összefoglalás

Dolgozatunkban egy olyan mikroszolgáltatásokra épülő felhő-natív megoldást mutattunk be, amely képes seregélyek hangjának felismerésére és a madarak automatizált elriasztására. Megmutattuk, hogy a rendszer teljesítménye szempontjából kritikus komponensek jól skálázódnak. A tervezési és megvalósítás mérnöki folyamata egy, a DevOps szemléletbe illő, általunk kidolgozott módszert követett.

Az elkövetkező időszakban szándékunkban áll megtervezni és implementálni az előző fejezetben javasolt módosításokat, valamint azok hatékonyságának értékelését is el szeretnénk végezni.

Úgy gondoljuk, hogy a rendszerben számos további lehetőség rejlik, amit érdemes lenne kiaknázni, ezért további funkciókkal szeretnénk bővíteni mind az IoT eszközön futó szoftvert, mind a felhő-natív rendszert. Egy elemzés alatt levő bővítési irány további szenzor adat integrálása, például időjárási adatok alapján javítani a seregélyek detekciójának pontosságát.

6. fejezet

Köszönetnyilvánítás

Köszönjük a konzulenseinknek a segítőkész hozzáállását és útmutatását Dr. Maliosz Markosznak és Dr. Simon Csabának. Köszönjük szüleinknek és a barátainknak a támogatást.

A dolgozatban ismertetett eredmények a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Kar Balatonfüredi Hallgatói Kutatócsoport szakmai közössége keretében jöttek létre a régió gazdasági fejlődésének elősegítése érdekében. Az eredmények létrehozása során figyelembe vettük a balatonfüredi központú Rendszertudományi Innovációs Klaszter által megfogalmazott célkitűzéseket, valamint a párhuzamosan megvalósuló EFOP 4.2.1-16-2017-00021 pályázat támogatásával elnyert „BME Balatonfüredi Tudáscentrum" térségfejlesztési terveit.

A kutatás az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósult meg (EFOP-3.6.2-16-2017-00013, Innovatív Informatikai és Infokommunikációs Megoldásokat Megalapozó Tematikus Kutatási Együttműködések).

Irodalomjegyzék

[1] Ghazi Mahjoub, Mark K. Hinders, and John P. Swaddle. Using a “sonic net” to deter pest bird species: Excluding european starlings from food sources by disrupting their acoustic communication. *Wildlife Society Bulletin*, 39(2):326–333, 2015.

[2] Cloud computing: A complete guide. `https://www.ibm.com/cloud/learn/cloud-computing`. Megtekintve 2020-02-27.

[3] What is saas? software-as-a-service defined. `https://www.infoworld.com/article/3226386/what-is-saas-software-as-a-service-defined.html`. Megtekintve 2020-02-21.

[4] What is function as a service (faas)? `https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/`. Megtekintve 2020-02-23.

[5] Saas vs paas vs iaas: What’s the difference and how to choose. `https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/`. Megtekintve 2020-02-21.

[6] What is infrastructure-as-a-service? everything you need to know. `https://www.techradar.com/news/what-is-infrastructure-as-a-service`. Megtekintve 2020-02-21.

[7] Nodes. `https://kubernetes.io/docs/concepts/architecture/nodes/`. Megtekintve 2020-02-27.

[8] kube-apiserver. `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/`. Megtekintve 2020-02-20.

[9] Data model. `https://etcd.io/docs/v3.4.0/learning/data_model/`. Megtekintve 2020-02-20.

[10] Pods. `https://kubernetes.io/docs/concepts/workloads/pods/pod/`. Megtekintve 2020-02-21.

[11] Deployments. `https://kubernetes.io/docs/concepts/workloads/controllers/deployment/`. Megtekintve 2020-02-20.

[12] Service. https://kubernetes.io/docs/concepts/services-networking/service/. Megtekintve 2020-02-21.

[13] Ingress. https://kubernetes.io/docs/concepts/services-networking/ingress/. Megtekintve 2020-02-25.

[14] Chris Richardson. Introduction to microservices. https://www.nginx.com/blog/introduction-to-microservices/. Megtekintve 2020-02-10.

[15] S. Hassan and R. Bahsoon. Microservices and their design trade-offs: A self-adaptive roadmap. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 813–818, 2016.

[16] Y. Niu, F. Liu, and Z. Li. Load balancing across microservices. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 198–206, 2018.

[17] Nour E. Oweis, Claudio Aracenay, Waseem George, Mona Oweis, Hussein Soori, and Vaclav Snasel. Internet of things: Overview, sources, applications and challenges. In Ajith Abraham, Katarzyna Wegrzyn-Wolska, Aboul Ella Hassanien, Vaclav Snasel, and Adel M. Alimi, editors, *Proceedings of the Second International Afro-European Conference for Industrial Advancement AECIA 2015*, pages 57–67, Cham, 2016. Springer International Publishing.

[18] M. Aazam, I. Khan, A. A. Alsaffar, and E. Huh. Cloud of things: Integrating internet of things and cloud computing and the issues involved. In *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences Technology (IBCAST) Islamabad, Pakistan, 14th - 18th January, 2014*, pages 414–419, 2014.

[19] Seema Singh. Cousins of artificial intelligence. https://towardsdatascience.com/cousins-of-artificial-intelligence-dda4edc27b55. Megtekintve 2020-10-27.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[21] Nagy Kristóf. Tömeges gép-gép kommunikáció mezőgazdasági alkalmazása. https://diplomaterv.vik.bme.hu/hu/Theses/Tomeges-gepgep-kommunikacio-mezogazdasagi. MSc Diplomaterv, BME-VIK, 2020.

[22] Mi az azure iot hub? https://docs.microsoft.com/hu-hu/azure/iot-hub/about-iot-hub. Megtekintve 2020-03-25.

[23] Mi az az azure stream analytics? https://docs.microsoft.com/hu-hu/azure/stream-analytics/stream-analytics-introduction. Megtekintve 2020-03-25.

[24] Justice Opara-Martins, Reza Sahandi, and Feng Tian. Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. *Journal of Cloud Computing*, 5(1):4, Apr 2016.

[25] H. Chen and F. J. Lin. Scalable iot/m2m platforms based on kubernetes-enabled nfv mano architecture. In *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1106–1111, 2019.

[26] Microbenchmark of different python http clients. `https://github.com/svanoort/python-client-benchmarks`. Megtekintve 2020-05-24.

[27] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar. Application-level performance monitoring of cloud services based on the complex event processing paradigm. In *2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8, 2012.