Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati rendszerek és szolgáltatások Tanszék

Tóth Vince

KUBERNETES ALAPÚ SKÁLÁZHATÓ AUTOMATIZÁLT FOLYAMATMENEDZSMENT KERETRENDSZER NAGYLÉPTÉKŰ 5G-V2X SZIMULÁCIÓKHOZ

Diplomaterv

KONZULENS

Dr. Bokor László

BUDAPEST, 2022.05.29

Tartalomjegyzék

1 Bevezetés ... 8
2 Szimulációs rendszer és folyamat ... 9
2.1 Artery szimulátor ... 9
2.1.1 SUMO Városi forgalom szimulátor ... 10
2.1.2 TraCI interfész ... 12
2.1.3 Omnet++ Hálózat Szimulátor ... 12
2.1.4 Veins ... 14
2.2 Szimulációs folyamat ... 15
2.3 Folyamatok automatizálása ... 15
3 Kubernetes felhő alapú technológia ... 17
3.1 Konténerizáció ... 18
3.2 Konténer Orkesztráció ... 19
3.3 Kubernetes ... 20
3.3.1 Kubernetes komponensek ... 20
3.3.2 Kubernetes futtató környezet lehetőségei ... 23
3.4 Szimulációk konténerizált futtatása ... 24
3.5 Kubernetes alapú szimulációk előnyei ... 25
4 Összetett automatizált folyamatmenedzsment rendszer tervezése ... 28
4.1 Rendszer architektúra ... 28
4.2 Komponensek ... 30
4.2.1 GitLab ... 30
4.2.2 CI/CD Pipeline ... 30
4.2.3 Konfigurációs szerver és vezérlő backend szolgáltatás ... 31
4.2.4 Konfigurációs szerver Frontend szolgáltatás ... 31
4.2.5 Monitoring ... 32
4.2.6 Szimulációs szolgáltatás ... 33
4.2.7 Sidecar Konténer ... 34
4.2.8 Eredmény gyűjtő szolgáltatás ... 34
4.2.9 Eredmény megjelenítő frontend szolgáltatás ... 35
4.2.10 Támogató komponensek ... 35
4.3 További rendszer tervek ... 35

4.3.1 Ütemező Funkció 35
4.3.2 Szimulációk megfeleltetése Kubernetes Deployment-ekre 37
5 Kubernetes alapú felhő infrastruktúra létrehozása 39
5.1 Virtualizáció 39
5.2 Virtuális Host a Kubernetes számára 39
5.3 Kubernetes rendszer 41
5.3.1 Telepítési módok 41
5.3.1.1 K3s 41
5.3.1.2 RKE 42
5.3.2 Kubernetes node-ok típusai 44
5.4 Rancher 45
5.4.1 Rancher telepítése 46
5.4.2 Rancher kiegészítők 47
5.4.3 Elosztott tárhely megoldás alkalmazása 48
6 Összetett Kubernetes alapú rendszer megvalósítása 49
6.1 Menedzsment backend szolgáltatás 49
6.1.1 Adatmodell és API 49
6.1.2 Funkciók implementálása 51
6.1.2.1 Klaszter állapot vizualizálás 51
6.1.2.2 Szimulációk kezelése 51
6.2 Eredmény feldolgozó backend szolgáltatás 54
6.2.1 Modell és API 55
6.2.2 Szimuláció eredmény feldolgozás 56
6.3 Konfigurációs és eredmény megjelenítő felület frontend alkalmazás 58
6.3.1 Alkalmazott könyvtárak 58
6.3.1.1 Styled-components 58
6.3.1.2 React-Redux 58
6.3.1.3 React-Toastify 59
6.3.1.4 Kész applikáció 59
6.4 Alkalmazások konténerizálása és Kubernetes integrációja 64
6.4.1 Java backend konténerizálás 64
6.4.2 Python Django backend konténerizálás 64
6.4.3 JavaScript SPA konténerizáció 65
6.4.4 Kubernetes integráció 66

6.4.5 Kubernetes integráció kihívásai ... 68
7 5G-V2X szimuláció tervezése és megvalósítása a rendszer teszteléséhez ... 69
8 Szimuláció CI/CD ... 73
9 Szimulációs folyamat automatizált működése ... 75
10 Telepítés ... 79
11 Összefoglalás ... 80
Eredmények értékelése ... 80
Továbbfejlesztési lehetőségek ... 81
Köszönetnyilvánítás ... 82
Szójegyzék ... 83
Ábrajegyzék ... 86
Irodalomjegyzék ... 89
Függelékek ... 91
Rke cluster.yaml ... 91
Gitlab CI/CD yaml: ... 91
Python backend Dockerfile: ... 92
JavaScript frontend Dockerfile: ... 93
Java backend Dockerfile: ... 93
Python results_maker.py ... 94

HALLGATÓI NYILATKOZAT

Alulírott **Tóth Vince**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 05. 29.

...…………………………………………….

Tóth Vince

Kivonat

A modern közlekedés egyik új technológiai kihívása a V2X kommunikáció, és annak kihasználása a forgalom működésének javításához. Az új szolgáltatások, amiket a C-ITS és ADAS vezetést segítő alkalmazások nyújtanak segíthetnek a közlekedési balesetek számának csökkentésében, a károsanyag kibocsátásának csökkentésében, vagy a közlekedők kényelmének javításában. A fejlesztések segítésére és támogatására, a V2X témakörű hálózati és forgalmi szimulátorokat használnak a fejlesztők, amik a valóságot próbálják hűen utánozni azért, hogy az új alkalmazások hatékonyságáról információkat kapjanak. Ez a módszer költséghatékony és gyors. Nem kíván valós beavatkozást a forgalomba és növeli a forgalmi tesztelésbe kerülő alkalmazások biztonságát a virtuális tesztelések segítségével.

Az ilyen szimulátor rendszer gyakran összetett, nagy erőforrásigényű szoftvereket alkalmaznak. Ezeket az igényeket pedig lehetséges modern technológiai megoldásokkal jobban kielégíteni. Az új felhő alapú rendszerek, jelenthetik a megoldást a még pontosabb és komplikáltabb szimulációk felé. A szimulációk infrastruktúráját a felhőben végtelenségig lehet erőforrással támogatni. A modern támogató funkciókkal pedig a fejlesztők munkáját lehet könnyebbé tenni. A munkám során a feladatom a megtervezése, és az implementálása egy olyan rendszernek, ami modern felhő alapú infrastruktúrába ültet egy összetett V2X szimulációs rendszert. A forgalom szimulátort az SUMO szolgáltatja, a hálózat szimulálásáért pedig az Artery keretrendszer a felelős. A szimulációs rendszert a Kubernetes felhő technológia segítségével felhő architektúrába integrálom, konténerizációt támogató eszközökkel és az eredményeket automatikusan kiértékelő modullal. Ezáltal a fejlesztők munkáját megkönnyítő modern szolgáltatás lesz, a keretrendszerből.

Abstract

One of the new technological challenges of modern transportation is V2X communication and its use to improve the operation of traffic. The new features of C-ITS and ADAS driving assistance applications can help reduce the number of traffic accidents, reduce emissions, or improve driver comfort. To help and support improvements, developers are using V2X-themed network and traffic simulators that try to realistically mimic reality to get information about the effectiveness of new applications. This method is cost-effective and fast. It does not require real intervention in traffic and increases the security of applications that are tested for traffic through virtual testing.

Such a simulator system often uses complex, resource-intensive software. And these needs can be better met with modern technological solutions. New cloud-based systems can be the solution to even more accurate and complicated simulations. The infrastructure of the simulations in the cloud can be extended with resources indefinitely. And modern support features make it easier for developers to work.

In my work, my task is to design and implement a system that implements a complex V2X simulation system into modern cloud-based infrastructure. The traffic simulator is provided by the SUMO, and the Artery framework is responsible for simulating the network. I integrate the simulation system into a cloud architecture using Kubernetes cloud technology, with tools to support containerization and a module that automatically evaluates the results. This will be a modern service to facilitate the work of developers with the framework.

.

1 Bevezetés

A modern közlekedés új technológiai innovációja a kooperatív járművek rendszere. Ez számos környezetvédelmi, közlekedésbiztonsági és kényelmi funkció fejlődésével kecsegtet, mindemellett pedig az egyik legfontosabb támogató technológiája lehet az önvezető járműveknek. Ezeket a rendszereket és alkalmazásokat szimulációs környezetben is vizsgáljuk, a költségek optimalizálására és a tesztelési folyamatok gyorsítására. Az egyik ilyen szimulációs rendszer a több alkalmazásból álló összetett szimulátor az Artery nyílt forráskódú rendszer, ami komplex megoldást ad a V2X technológiák tesztelésére és fejlesztésére.

Ez a szimulátor rendszer mikroszkopikus felbontással képes szimulálni a forgalomban résztvevő járműveket és a köztük folyó kommunikációt, ezáltal nagyfokú realitást lehetségessé téve. Maga a szimulátor bárki számára elérhető és futtatható saját környezetben, ugyanakkor a nagymértékű szimulációk futtatására is képes ipari méretekben. Maga a szimulátor egyszerű monolitikus rendszerként fut, de van lehetőségünk konténerizált megoldás használatára, ami motivációként szolgál a jelen munka elkészítéséhez.

A dolgozatban összefoglaljuk a már említett szimulátor rendszer felhő alapú skálázható és automatizált keretrendszerének tervezését és implementálását. A keretrendszer célja lesz a fejlesztő munkájának támogatása, hogy az elkészült szimulátor verzió automatikusan tesztelhető legyen, párhuzamosított konténerizált formában, egy felhő architektúrán.

2 Szimulációs rendszer és folyamat

A szimulációs rendszer maga is több alkotóelemből áll össze mégis alapvetően monolitikus felépítésűnek mondható. A továbbiakban a rendszer részeit megemlítve felvázoljuk a szimulátor szerkezetét, majd a szimulációs folyamat leírása után megmutatjuk a folyamat automatizálásában rejlő lehetőségeket és előnyöket.

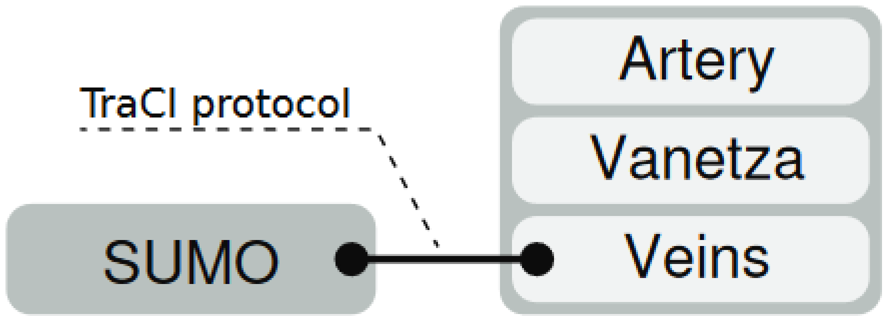
2.1 Artery szimulátor

A rendszer komponensei egyenként is hasznos szimulációs eszközök és megvan a saját rendeltetésük azonban a közös használatukkal elérhetünk egy új megközelítésű valóságos szimulációt a V2X szimulációkhoz.[1]

Az szimulációk a V2X kommunikáció által elérhető új alkalmazások, ADAS (Advanced Driving Assistance Systems) -ok kifejlesztését segítik , illetve az IEEE 802.11p protokoll család viselkedését vizsgálják. Az előbbit azért, mert így költséghatékony módon tudnak jövőbeli alkalmazásokat tesztelni és fejleszteni. Az utóbbit pedig azért, hogy a kommunikációs képességeket protokoll szinten vizsgálva tudják megmondani annak hatását az applikációkra.[2]

Az Artery V2X szimulációs keretrendszer célja, hogy a már elterjedt VANETs (Vehicular Ad Hoc Networks) rendszereket vizsgáló programok hatékonyságát növelje, mivel ezek nagyon komplikált rendszerré tudnak válni egy bizonyos méret fölött. Ezek gyorsan létrejövő önszerveződő lokális hálózatok, amik gyakran változnak, a környezetük is változik, és a résztvevők is változnak. Emiatt nagyon komplex rendszert alkotnak.[2]

Az Artery keretrendszer a Veins (Vehicles in Network Simulation) -re épül és egészíti azt ki. Javítva annak képességeit az VANET alkalmazások analizálását tekintve, ezáltal több különböző alkalmazást egy szimulációban lehetővé téve, több különböző képességű járműtípussal. [2]



1. ábra Artery moduláris rendszer [2]

Ezt a kiegészítést egy háromtagú moduláris rendszerrel teszi, amiben a Veins köti össze a SUMO szimulátort az Omnet++ hálózati szimulátor által megvalósított 802.11p rádiókommunikációs réteggel. A Vanetza a Artery rendszerhez készített implementációja az ETSI (European Telecommunications Standards Institute) ITS-G5 hálózati és szállítási réteg béli protokolloknak.[2]

2.1.1 SUMO Városi forgalom szimulátor

A SUMO (Simulation of Urban Mobility), szimulátor egy nyílt forráskódú, mikroszkopikus forgalom szimulátor. Az egyes járművek lépéseit explicit módon számolja, a megfelelő forgalmi modellek szerint, diszkrét időben. Léteznek folyam alapú makroszkopikus szimulátorok is, amelyek más tulajdonságokkal bírnak, de számunkra a SUMO a legideálisabb rendszer. A Német Repülésügyi hivatal (German Aerospace Center- DLR) hozta létre, ideális a részletes modellezéshez. Lépésenként különböző modellek alapján számolja a járművek sebességét, és sokféle statisztikai adatot tud szolgáltatni.[3]

A mikroszkopikus tulajdonsága miatt képesek lehetünk teljes kontrollált szimulációkat létrehozni benne, mivel a szimuláció menete során minden egyes lépésről pontos információkat nyerhetünk ki, mielőtt tovább haladna a program. Más típusú programok esetén időközönként vehetnénk mintát egy szimulátor futásának állapotáról, amivel pontatlanabb adatokat kapnánk.

A másik, ami miatt fontos a mikroszkopikus tulajdonság az a szimulátorban résztvevő járművek vezérlése. Az ADAS applikációk, amik befolyásolják a járművek viselkedését minden lépésben közbe tudnak avatkozni és vezérelni a járműveket.

A SUMO szimulátor valós térkép fájlokat tud feldolgozni, amiket később szerkeszthetünk a saját kedvünkre. Ezáltal lehetőséget kapunk a valós forgalmi környezetekben való tesztelésre. Akár meglévő V2X infrastruktúrával felszerelt környezetek, helyszínek viselkedését is figyelhetjük a szimulátorban. Ennek jelentőséget növeli az ilyen okos városok elterjedése[4].

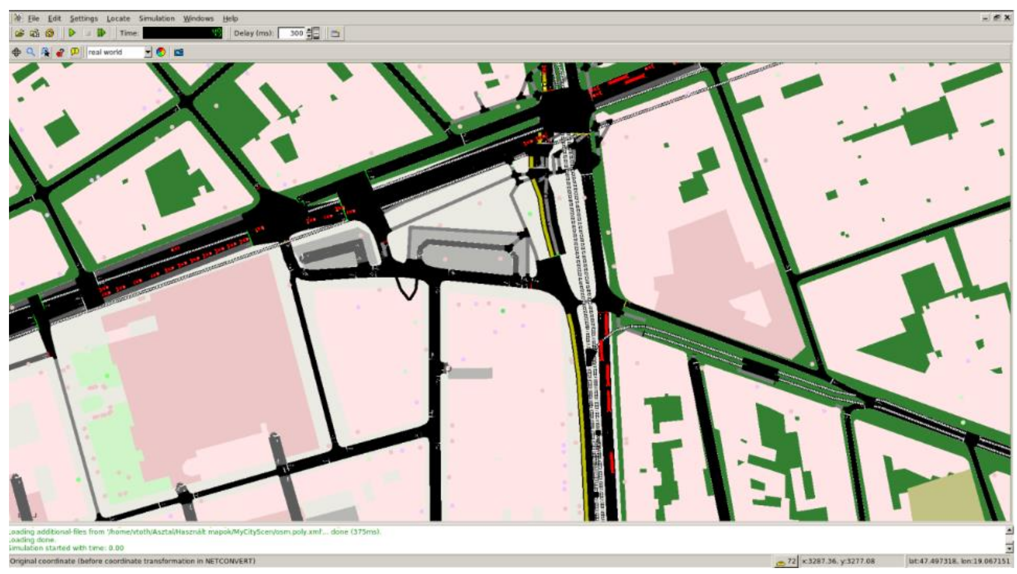
Emellett a forgalom generálására is ad módot. Lehet véletlenszerűen generált, illetve előre megadott útvonalakat is használni, ezeket az útvonalakat külön fájlban kell megadnunk. Ezeket gyakran kombinálni kell azért, hogy megfelelően valósághű szimulációkat kapjunk. A térképek kézzel történő kijavítása is gyakran ajánlott, a kiszámíthatatlan helyezetek elkerülése végett. Az ilyen utómunkák után viszont nagyon valóságos viselkedést tudunk elérni a valóságból hozott helyszíneken.[3]

2. ábra Szerkesztett autópálya térkép[3]

A különböző típusú járműveket is szerepeltethetünk, ezáltal a vizsgált alkalmazások száma is megnő, helyet kaphatnak tömegközlekedést vagy más típusú járműveket segítő applikációk. A térképadatok felhasználásával képes valós földrajzi koordinátákat is szolgáltatni. Ennek jelentősége más típusú szimulátorokban kap helyet igazán, ahol esetleg valós eszközöknek kell helyadatokat szolgáltatni, például HiL(Hardwere in the Loop) szimulátorokban.[5], [6]

A térképek tartalmazhatják a környezet épületeit, amiket magassággal ellátva a rádiókommunikációs szimulációt szolgáltató programban ugyanúgy felhasználhatunk a jelterjedés vizsgálatánál, hiszen a rádiós kommunikáció egyik legnagyobb tényezője a környezet és annak tulajdonságai.[5]

Mindemellett grafikus interfésszel is rendelkezik, ami demonstrációs szempontból sem utolsó, illetve egy-egy jármű viselkedését vizuálisan felmérve javul a rapid-prototyping sebessége, mivel az egyértelmű hibák kiszűrése gyorsabbá válik, adatok analizálása helyett. Ez a mód későbbi fázisokban nem lesz hasznos, a nagy erőforrásigénye miatt. Ezért csak speciális esetekben alkalmazandó.



3. ábra SUMO grafikus interfész[3]

2.1.2 TraCI interfész

A Veins szimulátor képességeit csak úgy lehet kihasználni, ha a járművek viselkedését szimuláló programokkal, forgalom szimulátorokkal együtt tudjuk megtenni. A TraCI (Traffic Control Interface) egy olyan közbülső interfészt szolgáltat, ami képes összekötni a különböző mobilis hálózati node-ok és a forgalmi szimulátorok járműveinek mozgását. Ez így már teljessé teszi a szimulátorunkat, mivel összeköti a SUMO forgalomszimulátort és a Veins, illetve Artery keretrendszert.[7]

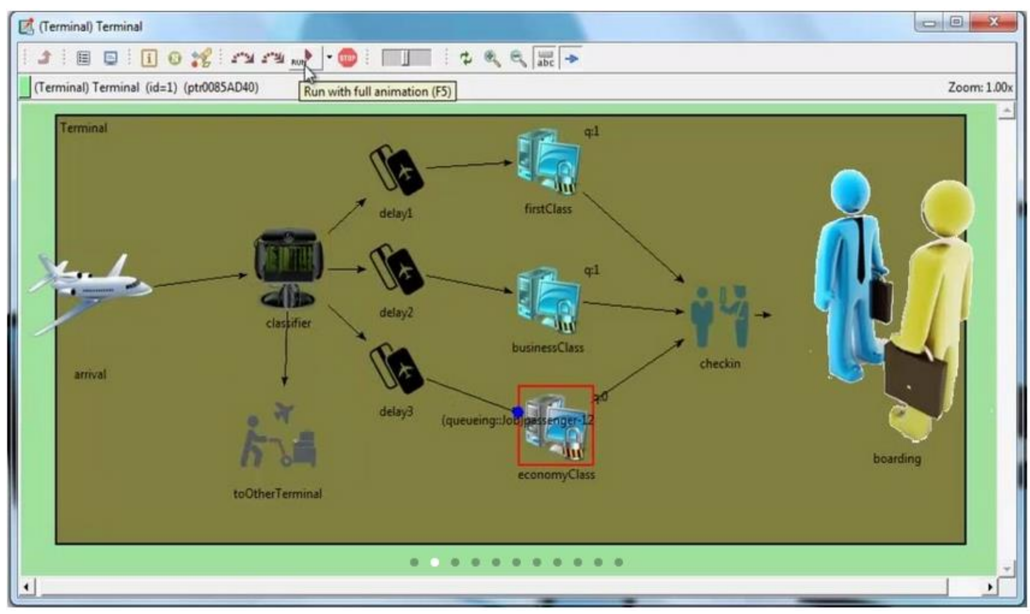
2.1.3 Omnet++ Hálózat Szimulátor

Az Omnet++ egy C++ alapú szimulációs könyvtár és keretrendszer. Tartalmaz egy integrált fejlesztő és vizualizációs futtató környezetet. A felépítésére jellemző a modularitás és a kiterjeszthetőség. A konkrét hálózati rendszerek összetevőit más, ráépülő könyvtárak és rendszerek adják.[8]

Esetünkben az INET Framework az egyik legrégebbi és legelterjedtem szimulációs modellgyűjtemény az, ami az Omnet++ rendszerbe épül. Az INET az Omnet++-vel együtt

fejlődött. Az első verziókban az Internet Protocol stack, tehát az IPv4, TCP, és UDP protokollokat valósította meg. Azonban ez később kiegészült sok más protokollal is.[9]

Az így kiegészített Omnet++ egy összetett hálózati szimulációs rendszer, ami teljes protokoll stacket átívelően képes a kommunikációt szimulálni egészen a jelterjedési paraméterektől kezdve a magasabb szintű protokollok viselkedéséig. Mindezt mikroszkopikusan lépésről lépésre.

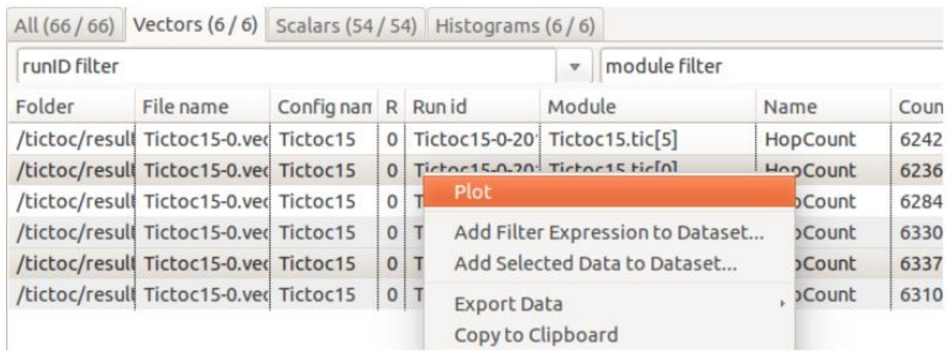


4. ábra Omnet++ grafikus felület

A szimulációban node-okat definiálunk amire különböző technológiákat megvalósító adóvevőket tehetünk. Ezek a node-ok lehetnek statikusak, de mozgó node-okat is definiálhatunk.[10]

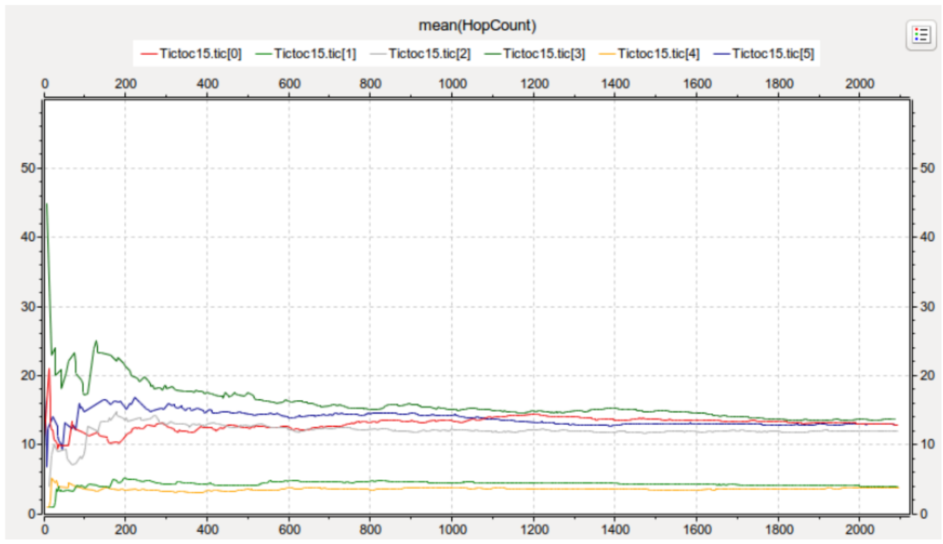
A megfelelő szimulációs konfigurációkat a különböző leíró ned kiterjesztésű fájlok és C++ nyelvű kódrészletek adják. A rendszer maga úgy lett szegmentálva kódban, mint ahogy a valóságban is elkülönülnek a rétegek egymástól és a különböző entitások. Az üzenetek végighaladnak a C++ osztályokon mint a protokollok egyes rétegein.[11]

A szimulációk kimeneteleit az Omnet++ által szolgáltatott vektor és skalár információk alapján is értékelhetjük. Ezek a szimuláció során mért adatok megfelelői, amiket az Omnet saját grafikus felületén is megvizsgálhatunk.



5. ábra Omnet++ eredmény vizualizációs felület [12]

Ezzel a módszerrel az Omnet++ beépített környezetét használva kaphatunk rövid idő alatt feldolgozott és vizualizált eredményeket. Ezek fontos adatokat szolgáltathatnak az alkalmazások kiértékeléséhez.



6. ábra Vizualizált szimulációs eredmény[12]

2.1.4 Veins

A Veins egy szimulációs modellkönyvtár, az Omnet++ keretrendszerhez. A feladata támogatni a kutatókat a járműkommunikációs rendszerek szimulációjával. Lehetnek ezek VANETs rendszerek vagy ITS(Intelligent Transportation Systems) komponensek.[13]

Ezt teszi azáltal, hogy egy szofisztikált modellt ad az IEEE 802.11 MAC layer komponensek számára. Ezt használja fel az Artery keretrendszer az ETSI ITS-G5 protokoll szimulációjához. A Veins olyan moduláris felépítésű rendszer ami egyaránt alkalmas gépjárművek vagy más közlekedési eszközök, esetleg gyalogosok szimulációját végző rendszerek alapjául szolgálni. [13]

2.2 Szimulációs folyamat

A folyamatot az elemek összekapcsolása és kooperatív futtatása adja. Az első lépés a SUMO szimuláció elindítása. Ez ekkor betölti a térképet, és a járműveket amiket majd a forgalomban szimulálni fogunk. Az Omnet++ szimulátor a felsorolt szimulációs modell könyvtárakkal kiegészítve (INET,Veins,Vanetza,Artery) a hálózati szimulációt a megfelelő node-okkal szimulálja. A TraCI interfészen keresztül a SUMO-ban szimulált járművekről kap információt az Omnet++ ami ezáltal képes a node-ok mobilitását szimulálni. Az Artery az adott ADAS(Advanced Driving Assistance Systems) applikációt futtatva pedig szintén a TraCI interfészen keresztül tudja befolyásolni és kontrollálni SUMO-ban szimulált járművek viselkedését. A szimuláció léptetése során az Omnet++ egy másodpercben szimulálja a hálózat viselkedését, majd lépteti a SUMO-szimulációt.

2.3 Folyamatok automatizálása

Az Artery szimulátor futtatása során masszív szimulációkat hozhatunk létre. Az Omnet++ saját konfigurációs fájlja megengedi, több térkép és más konfigurációs fájl illetve paraméter használatát. Például különböző térképeken, vagy különböző generált forgalmakkal lehet szimulálni az alkalmazásaink viselkedését. Így árnyaltabb képet kapva az ADAS alkalmazás hatásáról, eredményeiről.

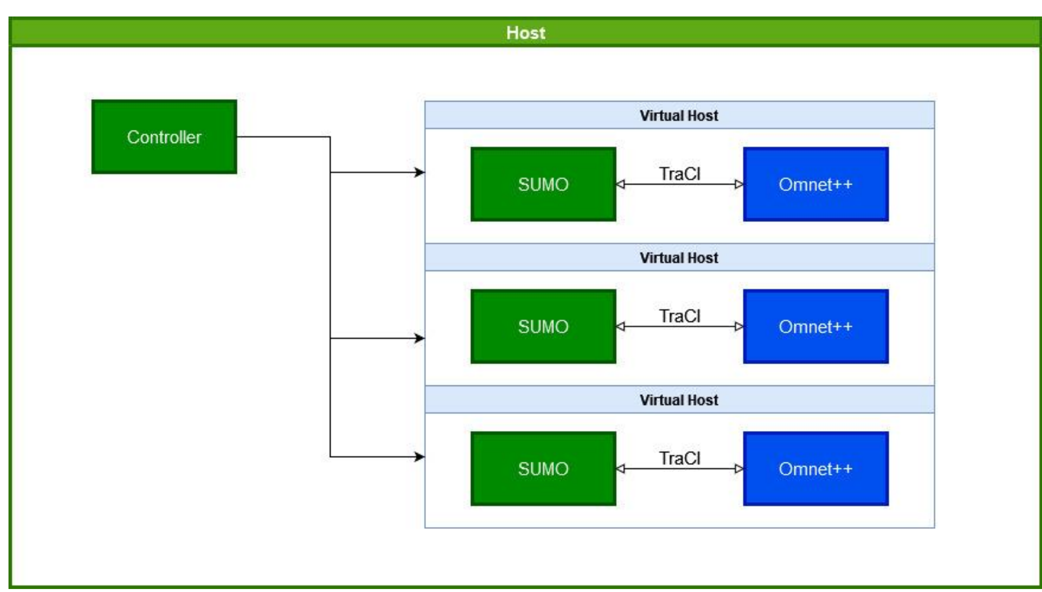
A futtatás így nem egy szimulációt, hanem több szcenárió futtatását eredményezi, egymás után sorban. Ez azt eredményezi, hogy az alapvetően több futtatás egymás után megy le időben, kitolva a teljes szimulációs konfiguráció futási idejét. Tehát felmerül a kérdés, hogy a futási idő csökkentése lehetséges-e párhuzamosítással. A párhuzamosítás lehetőségét adja, hogy a szimulátor rendszer nem képes teljesen kihasználni a rendszer erőforrásokat. Ezáltal, ha szcenáriókat a sorban indítás helyett, valamilyen fokú párhuzamosítással indítjuk a teljes futási időt jelentősen optimalizálhatjuk.

Ennek szükségességét az Omnet++ konfigurációk kihasználásakor vehetjük észre. Mikor egy esetben a térképek, generált forgalmak és más (például a véletlenszám generátor seed értéke) konfigurációk beállítása után, a valós szcenáriók száma ezen értékek szorzata. Ekkor a futási idő már jelentős változást mutathat soros és párhuzamos futtatás esetén.

Az Omnet++ futások párhuzamosításához, a saját ütemező rendszerét megkerülve kell futtatni az szcenáriókat. A programot több verzióban kell elindítani, ahol a konfigurációk különböznek.

Több esetben futtatva azonban nem szabad megfeledkezni a program összetettségéről. A komponenseknek, mint például a SUMO szimulátor programnak is több példányban kell léteznie, hogy a különböző szcenáriókhoz, különböző forgalom szimuláció tartozzon. A több térképfájl megadása már ezt feltételezi. A TraCI interfész, ami összeköti az Omnet++ szimulátorral a SUMO szimulátort az egy adott porton elérhető. Ez tehát magával hozza a TraCI módosítását és a SUMO-ét is, mivel azokat együtt kell konfigurálni a változtatásra.

A problémát viszont megoldhatjuk virtualizáció segítségével is. Ennek több fajtája is létezik, viszont mindkettő hordozza magával a megoldást a problémára, az elkülönített futtató környezetet. Ez azt jelenti, hogy a külön szimulációs szcenáriókat megvalósító szimulációs rendszerek módosítás nélkül képesek egy gazda rendszeren futni.



7. ábra Egyszerűsített virtualizált szimulátor futtatás

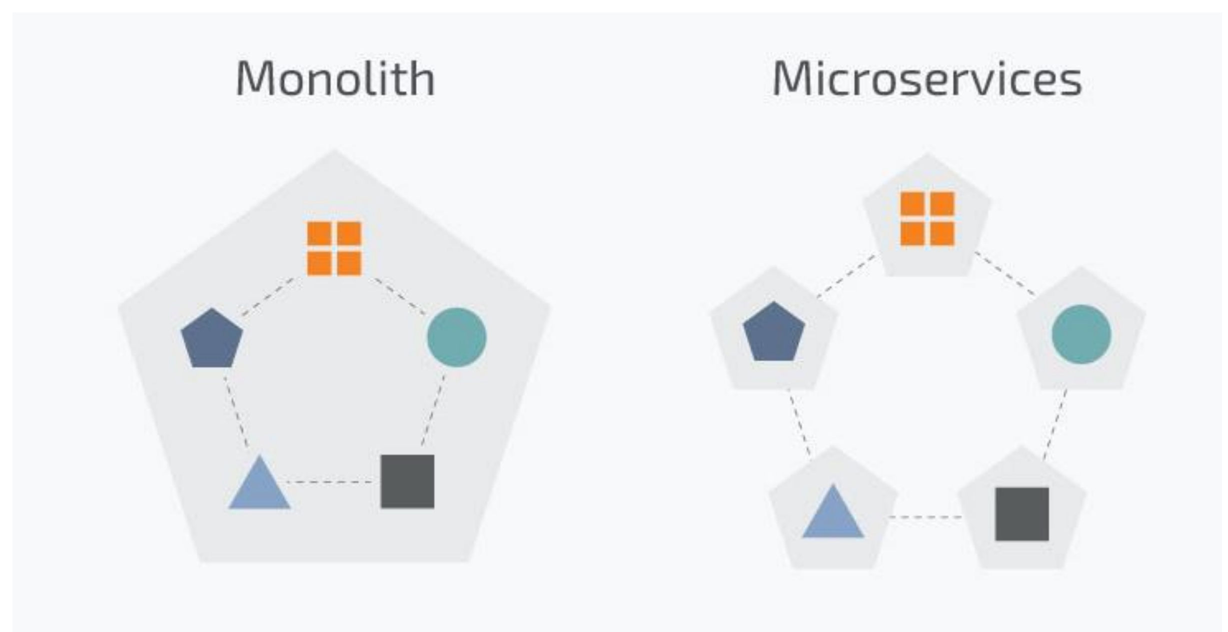
Innentől a feladat a konfigurációs szcenáriók előállítása, hogy azokat a virtualizációval elkülönített szimulátor programoknak szolgáltassuk. Azok a megfelelő szimuláció végrehajtása során már párhuzamosan, optimálisabb erőforráskihasználás mellett végzik el a feladatot.

Ennek tesztelése során a tapasztalatok azt mutatták, hogy nincs futásidő hátránya a konténerizáció alapú virtualizációs módszernek. A párhuzamosítás során a gazda gép processzor kihasználása javult, viszont a memória mennyiség felső korlátot szabott az egyszerre futtatott szimulációk számára. A nagyobb méretű szimulációk, hosszabb szimulációs idővel egyre több memóriát igényeltek, ezzel korlátozva a többi szimulátor konténert.

3 Kubernetes felhő alapú technológia

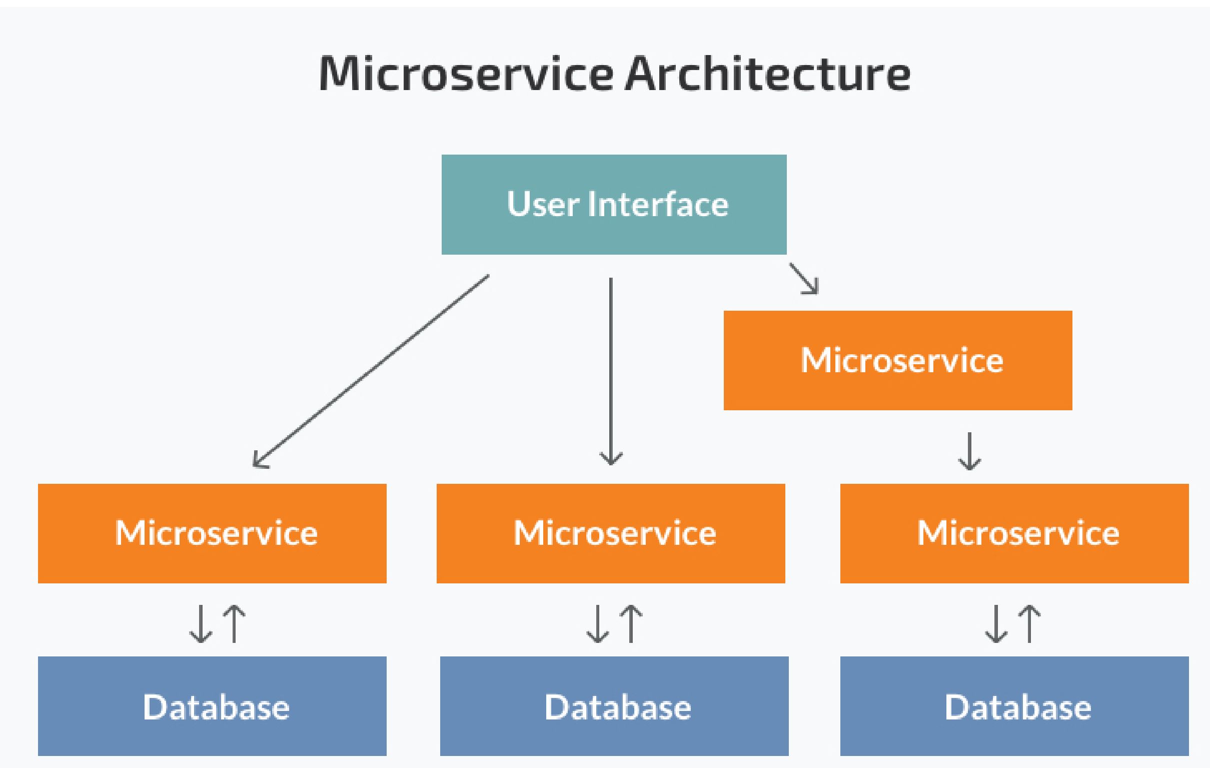
Az évek során programfejlesztési paradigma váltáson ment keresztül az ipar. Pár éve még a legtöbb rendszer alapját monolit szoftverek alkották, nagy kódbázisú összetett rendszerek amik néhány szálon futottak a szervereken.[14]

Ezek alapvetően a klasszikus programfejlesztési felfogás alapját jelentették. Jellemezte őket a lassú fejlesztési és tesztelési ciklus, illetve a lassú forgalomba hozási ciklus. A fejlesztési szakasz végén a tesztelések után a fejlesztőktől az üzemeltető részleghez került a teljes becsomagolt rendszer, majd ezt kézzel telepítették és üzemeltették a szervereken a régi kódbázist lecserélve. A rendszer migrációja nem csak a telepítések során volt lassú és felügyeletet igénylő, a szerver hibák javításánál is bonyolult kézi feladat volt.[14], [15]



8. ábra Mikroszolgáltatások és monolitok[15]

Ezeknek a programoknak a korlátait felismerve, és a növekvő kiszolgálási igény hatására a monolitikus programokat lebontották kisebb, önálló részekre. Ezek a programok az úgynevezett mikroszolgáltatások. A teljes funkcionalitást feldarabolva a részek között kooperálva nyújtották a teljes szolgáltatást. Ennek a hatására a különböző igényekre egyedi skálázási lehetőségek adódtak a mikroszolgáltatások egyéni kezelésével. Az magas rendelkezésre állású rendszerek esetén az egyes komponensek redundanciáját is külön lehetett így kezelni. [14]



9. ábra Mikroszolgáltatások felépítése [15]

Láthatjuk, hogy a paradigma váltással új lehetőségek nyíltak a fejlesztők számára, azonban ennek támogatására új infrastrukturális felfogás szükséges. A következő fejezetekben rövid betekintést adok a konténerizáció és a felhő alapú rendszerek működésébe, előnyeibe és kihívásaiba.

3.1 Konténerizáció

A konténerizációról röviden elmondható, hogy egy operációs rendszer virtualizációs megoldás. Képes egy számítógép viselkedését emulálni, úgy mintha annak saját erőforrásai lennének és saját operációs rendszere. Emiatt használjuk a szolgálatásaink futtatására. Mivel az alkalmazásaink komponensekre lettek bontva, emiatt nem lenne okos megoldás minden mikroszolgáltatást külön számítógépen futtatni. Ezeket inkább csomagoljuk egy konténerbe, ami magában foglalja a teljes futtató környezetet, a futtatandó kódot és magát a rendszert. Így egy kompakt és mobilis tárolót kapunk, ami képes a funkcióját ellátni.[16], [17]

A konténerizáció hasonló viselkedést mutat, mint a virtualizált számítógépek, azonban itt nincs külön VM (Virtual Machine) operációs rendszer. A konténerek mind közös operációs rendszer alatt futnak, és elosztott rendszer erőforrásokat kapnak. A virtuális gépek esetén jóval

több erőforrást emészt fel a VM operációs rendszere, az külön szimulált kernel és a gazda gép és a virtuális gép között elhelyezkedő felügyelő operációs rendszer.[16]

Mindezen tulajdonságai miatt és a kevés fölösen elhasznált erőforrás miatt használhatjuk a felhő alapú szolgáltatás fejlesztés alapjául. Az egyik legelterjedtebb használt konténerizációs megoldás a Docker, ami bár bőven ad lehetőséget a konténerek összetett rendszerbe állításához, mégis plusz funkciókat igényel a modern szolgáltatások nyújtásához.

A konténerek úgynevezett képekből jönnek létre, amik mint lenyomatként tartalmazzák a teljes futtató környezetet a telepített alkalmazással együtt. Amennyiben optimálisak, csak a legszükségesebb komponenseket tartalmazzák és emiatt gyorsan telepíthetőek.

3.2 Konténer Orkesztráció

A konténereket általában nem egyedül akarjuk használni, több konténert akarunk összekötni és az általuk nyújtott rendszer adja majd az alkalmazásunk funkcionalitását. Ez azt jelenti, hogy a konténereinket külön lefejlesztjük és futtatjuk a szervereinken, és ezt tesszük minden komponensre majd, amikor egy új verziójú mikroszolgáltatást szeretnénk használatba helyezni, akkor kézzel minden szerveren le kell cserélnünk a konténereinket.

Ez a megoldás alapvetően nem ad előnyt a monolitikus alkalmazások üzemeltetéséhez képest, de a rendszert kiegészítve már elérhetjük a fejlődést. Ez a kiegészítés a konténerek orkesztrációja lesz. Ezek olyan rendszerek, amelyek alapvetően átveszik a konténerek menedzselésének következő funkcióit[18]:

- A konténerek felügyelete és telepítése.
- A konténerek redundanciájának és elérhetőségének a felügyelete.
- Konténerek számának növelése, vagy konténerek eltávolítása a megfelelő terhelés elosztáshoz.
- Konténerek migrálása a gazda szerverek között amennyiben erőforrás hiány lép fel.
- A konténerek közt erőforrásfoglalás.
- Terhelés elosztás a konténerek között.
- A konténerek és gazda képek monitorozása életjelek szempontjából
- Az alkalmazások konfigurálása a futtató konténerekkel kapcsolatban.

Alapvetően amikor ezeket az orkesztrációs rendszerek alkalmazzuk, akkor valamilyen leíró módon, tipikusan Yaml fájl formátumban deklaratívan írjuk le az elvárt rendszer konfigurációit. Az orkesztrációs rendszer pedig ezt a konfigurált állapotot próbálja fenntartani.[18]

3.3 Kubernetes

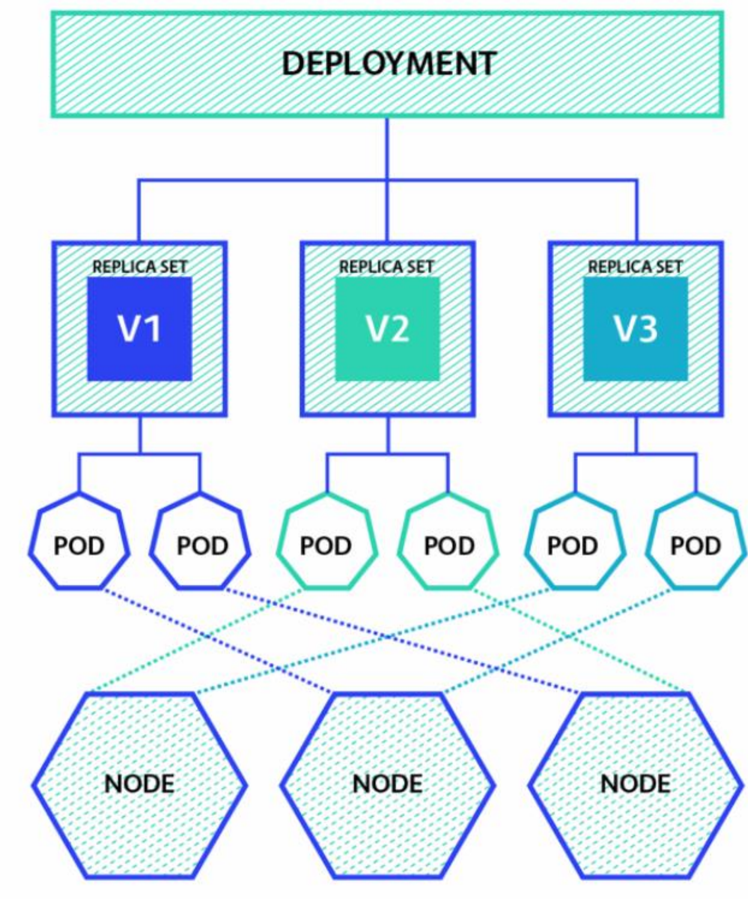
Az egyik legelterjedtebb konténer orkesztrációs és felhő alapú rendszer a Kubernetes. Eredetileg a Google fejlesztette, de miután hamar kinőtte magát mint vezető konténer orkesztrációs program, a Cloud Native Computing Foundation vette kegyeibe. A szervezetet több felhő alapú rendszerrel foglalkozó nagy szervezet támogatja mint például a Google, Azon Web Services, Microsoft, IBM, Intel, Cisco, és a RedHat.[18]

Docker, illetve az újabb verziókban Containerd alapú konténerizációs runtime rendszert használ. A konténerek életciklusát menedzselve hajtja végre az összetett feladatokat a különböző konténerek kooperációjának megteremtésével.

3.3.1 Kubernetes komponensek

A teljesség igénye nélkül a számunkra fontos Kubernetes elemeket szükséges megismernünk, a rendszer megértéséhez. Az összes komponens definiálása yaml fájlokkal történik.

A Kubernetes-ben a legkisebb önállóan menedzselhető egység, a Pod. A Kubernetes Pod egy vagy több konténert foglal magában. Saját egyedi azonosítója van és a végrehajtás alapegysége. A Pod-okat definiálhatjuk egy yaml-fájlban. Ebben megadhatjuk többek közt a futtatandó konténerek Image azonosítóját és nevét illetve a Pod által használt erőforrásokat, például Volume-okat, ConfigMap-okat és hasonlókat. A Pod-ok alapvetően felcserélhető egységek, legtöbbször állapotmentesen hajtanak végre (az alkalmazás szempontjából) egyszerű feladatokat. Ezáltal eldobhatóak és újra létrehozhatóak amennyiben szükséges.[19]



10. ábra Deployment és Pod-ok kapcsolata[20]

ReplicaSet az az egység, ami több Pod-ot tud magába foglalni. A ReplicaSet lényege, hogy a definiált Pod-okból adott időben garantálja a megszabott mennyiség meglétét. Felügyeli ezeket, szóval amennyiben nincs megfelelő mennyiségű adott Pod, megpróbál újakat létrehozni ameddig el nem éri a megfelelő replikációs számot. [19]

Deployment a tényleges feladatokat ellátó elem. A Pod-ok csak egyszerű példányai a részfeladatokat végrehajtó kis felelősséggel rendelkező szolgáltatásoknak. Az ezekre épített valós szolgáltatást a Deployment valósítja meg. A Deployment rendelkezik egy ReplicaSet-el ami a Pod-ok mennyiségét garantálja számára. Azonban a ReplicaSet-el ellentétben a Deployment feladata egy adott állapot előállítása deklaratívan. Tehát ReplicaSet-eket hozhat létre és szüntethet meg, és beállíthatja azok paramétereit. Általában egy konkrét feladatot, egy Deployment-ben fogalmazunk meg. Például egy webszerver megvalósítását egy Deployment-en keresztül létrehozott Pod-ok adják.[19]

A következő elem a Job, ami paramétereit és feladatát tekintve megegyezik a Deployment-el. Különbség abban van, hogy a Job-nak van célfeltétele. Adott számú ismételt

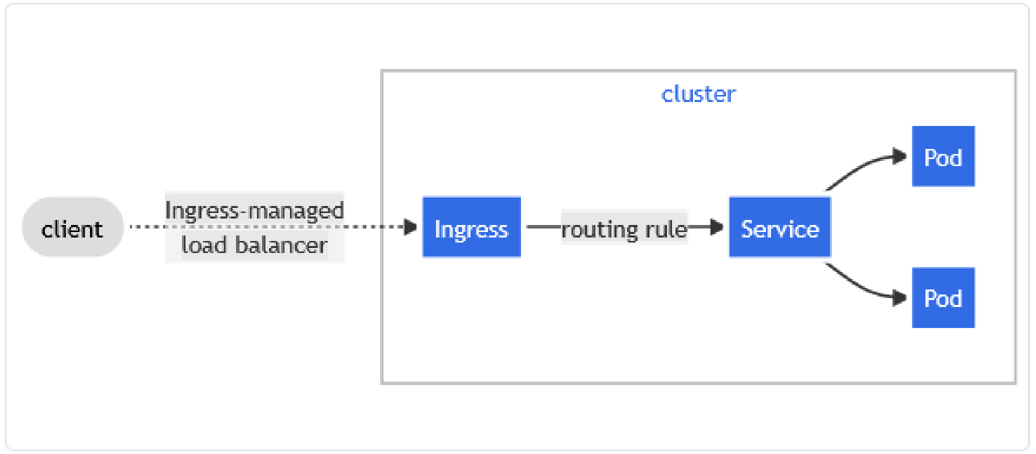
futást elérve a Job teljesítettnek tekintendő, a Deployment ezzel szemben nem definiált idejű, folyamatos szolgáltatást végez. A Job-ok replika száma is megadható, a párhuzamosítási számmal.[19]

Az előzőekben már említett Volume-ok a tárhelyeket kapcsoló objektumok. A Kubernetes rendszerben az általános helyzet, hogy több példány fut egyidőben. Ezek általában állapotmentesen dolgoznak, de az adatok, amiken dolgoznak közösek, így valósítják meg ugyanazt a szolgáltatást többszörösen. Az elosztott tárhely megoldás tehát kötelező a Kubernetesben.[19]

A Volume-ok amiket a Pod-ok felhasználnak elosztott tárhely entitásokra vonatkoznak. A valós háttértárat a Kubernetes számára PersistentVolume-ként kell lefoglalni. Ez szintén egy yaml-ben definiálhatunk Kubernetes elem, megadható a lefoglalandó tárhely mérete, illetve a Pod-ok hozzáférésének módja (Read-Write/Once-Many). Lehet hálózaton csatolt eszköz, S3 objektum vagy hasonló. A PesistentVolume-ra egy hivatkozással a PersistentVolumeClaim enittással lehet kapcsolódni. A PersistentVolume Kubernetes klaszter szintű entitás, a rá vonatkozó PersistentVolumeClaim azonban névtérrel ellátott entitás amit általában a Deployment-hez csatolunk. [19]

A ConfigMap a PersistentVolumeClaim mellett a másik Volume-ként hivatkozható entitás. A különbség, hogy a PersistentVolumeClaim egy PersistentVolume-ként felcsatolt könyvtárat tud Volume hivatkozáson keresztül a Pod-hoz kötni. A ConfigMap pedig fájlokat tud tárolni, kulcs érték párokban és ezt egy megadott útvonalon fájlként oda adni egy Pod-ban futó konténernek. Láthatóan a feladata az elosztott adatelérés helyett a specifikus információk Pod-okba injektálása. Ezeken keresztül szokás különböző programok konfigurációját felülírni a Kubernetes felhőbe integrálásnál.[19]

A PersistentVolume-ok menedzselése lehetséges statikus módon kézzel, vagy automatikusan egy StorageClass segítségével. Ez az elem, felelős a PersistentVolume-ok létrehozásáért és törléséért egy adott tárhelyen, az igényelt PersistentVolumeClaim-eknek megfelelően. Tehát amennyiben egy StorageClass-t üzemeltetünk akkor a Deployment-ekhez igényelt PersistentVolumeClaim-ek automatikusan kapnak egy PersitentVolume-ot és kapcsolódnak hozzá.[19]



11. ábra Ingress és Service kapcsolata a Pod-okkal[19]

A Serive-mint entitás a Pod-ok közös elérését hivatott megvalósítani. Mint említettük a Pod-ok közösen egy szolgáltatást adnak. Egymással felcserélhetőek és egyenértékűek legtöbbször, viszont mivel több van belőlük ezért a szolgáltatás egységes interfészét, a Service adja. Ez több Pod hálózati interfészének a portját teszi elérhetővé a klaszteren belül név szerinti címzéssel, vagy összevont ClusterIp-vel. Ezáltal egy szolgáltatás elérhető egy „helyen".[19]

Ingress komponenst azért használunk, hogy a külvilág számára is elérhetővé tegyünk szolgáltatásokat. Ez általában úgy történik, hogy az Ingress-en beállítunk egy domain nevet, amin keresztül publikálni akarjuk a szolgáltatást, majd az Ingress-t egy Service-re kapcsolhatjuk. Ezáltal az eddig belül elért szolgáltatást már a külvilág is használhatja.[19]

3.3.2 Kubernetes futtató környezet lehetőségei

A Kubernetes környezetben a már említett Deployment-eket használjuk arra, hogy szolgáltatásokat nyújtsunk más programok, felhasználók számára, egy összetett rendszer részeként. A rendszer számos problémára és feladatra automatizált megoldást nyújt, ami a mikroszolgáltatásoknál felmerül.

A környezet többek között garantálja az automatikus skálázását az egyes szolgáltatásokat futtató Pod-oknak. Az igények növekedésével párhuzamosan képes az erőforrásokat növelni a rendszerben, a Pod-ok számának növelésével.

A magas rendelkezésreállást igénylő rendszereket is kiszolgálja, mivel a megfelelő Kubernetes klaszter architektúrával a szolgáltatás kiesése nélkül lehetséges a program verziófrissítések és karbantartások elvégzése.

Esetleges hibák esetén a rendszer automatikusan helyrehozza a szolgáltatást, mivel a komponenseit bármikor újraindíthatja, lecserélheti. Az esetleges infrastruktúra szintű hibák kiküszöbölésére is képes, ilyenkor a kieső erőforrásokat pótolni tudja a Pod-ok maradék infrastruktúrára migrálásával.

A szolgáltatásról rendszerszintű adatgyűjtésre képes, így a teljes szolgáltatás megfigyelése lehetséges annak elosztott mivolta ellenére.

3.4 Szimulációk konténerizált futtatása

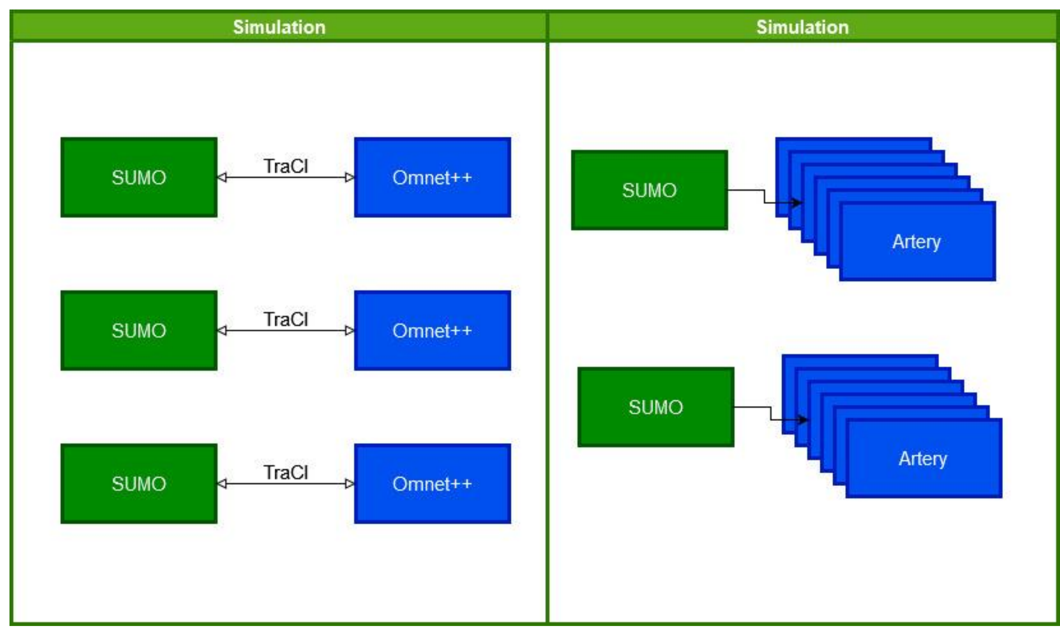
Az Artery szimulációk esetén már láthattuk a Docker alapú virtualizáció előnyeit. A szimulációk alapvetően monolitikus felépítésűek, viszont a szcenáriók mennyisége miatt mégis jól szegmentálhatóak. Emiatt a párhuzamosításuk is javallott, az időbeli nyereség miatt.

Ha egy szimulációról beszélünk, akkor azt egy nagy erőforrásokkal rendelkező rendszeren való futtatás tenné optimálissá. Azonban az Artery rendszer egy szimuláció futtatása során nem képes rendszer erőforrásait teljesen kihasználni, a programozásából adódóan.

Ezért az Artery szimulátort egy Docker konténerbe csomagolva lehet hatékonyabban futtatni. A Docker konténerben az Artery szimulátor és a SUMO forgalom szimulátor is jelen van.

Ezek szétválasztása két konténerbe teljesen lehetséges ugyanakkor nem optimális. A külön konténerben futó szimulátor két komponense közé tenni egy virtualizált hálózati réteget csak lassítja a szimuláció futását. Emellett a külön SUMO forgalom szimulátor konténer felhasználása sem változik, mivel az csak a hozzá tartozó Artery konténert tudná kiszolgálni mivel az alapesetben interaktív kódot futtat, ami változtatja a forgalom szimulátor járműveinek viselkedését.

Csak speciális esetben lehetséges a SUMO konténert hasznosítani, amennyiben az Artery program nem módosít a forgalom szimulátor viselkedésén. Ekkor elméletileg lehetséges lenne egy interfész megalkotásával az egy SUMO konténer összekötése több Artery szimulátorral. Azonban itt sem lenne optimális az eredmény, mivel az összes Artery konténer futását szinkronizálnánk, ezáltal a legrosszabb futási idejű szimulációhoz igazítanánk a kisebb számításigényűeket. Ez azt okozza, hogy a több kis futásidejű szimuláció addig foglalja az erőforrásait mint ameddig a legrosszabb futási idejű. Így alapvetően a maximálisan lefoglalt erőforrásigényűnk csökken a kisebb számú SUMO szimulátor konténer miatt, viszont a teljes szimulációs időre lefoglaljuk az össze Artery szimulációs programot kielégítő erőforrást.



12. ábra Szeparált konténerizálás

A Konténerben tehát alapesetben együtt tartjuk a SUMO szimulátor és Artery szimulátor példányainkat. Ezek együtt egy szcenáriót futtatnak egyidejűleg. A megfelelő tárhely menedzseléssel pedig különböző konfigurációkkal dolgoznak és különböző eredmény fájlokba, könyvtárakba írnak.

3.5 Kubernetes alapú szimulációk előnyei

A konténerizált szimulációk futtatása során felmerülő problémák vezettek rá, hogy esetleg Kubernetes rendszerben ez a koncepció hatékonyabban tud működni. A Kubernetes konténerizált alapokon nyújt szolgáltatásokat és a legtöbb problémára, amit ez a technológia felvet már van megoldás.

Kubernetes-ben futtatott szimulációk log-jait például összesíthetjük közös felületekre, amivel a fejlesztést, hibakeresést és analízist gyorsíthatjuk. A nagymennyiségű entitás log kimeneteit ráadásul könnyedén irányíthatjuk, feldolgozhatjuk mivel az entitások közötti kommunikáció már jól kidolgozott alapokon nyugszik. Egy SideCar konténer például anélkül képes a log-kimeneteket összeszedni a Pod-okból, hogy a konkrét programkódot módosítanunk kellene. Más megoldásokban, a programkód változtatásával érjük el ugyanezt, de ekkor is nagy segítséget nyújtanak a Kubernetes klaszterbe telepíthető különböző third-party alkalmazások, amik ezeket aggregálva tudják feldolgozni.

A Kubernetes a szimulációink konfigurálását is segítik, mivel minden példánynak külön tudunk ConfigMap-ok segítségével konfigurációs fájlt szolgáltatni. Ezáltal nem kell teljes külön könyvtárszerkezeteket felcsatolni és létrehozni a különböző szcenáriók futtatásához. Az első korai verzióban például egy temporális konfigurációs fájlt változtattam minden szcenárió indítás előtt, hogy megfelelő konfigurációkkal rendelkezzen a program. Később látjuk majd, hogy mennyivel letisztultabb megoldást ad a Kubernetes rendszer erre.

Az elosztott tárhely problémára a Docker rendszerhez hasonló megoldást ad komplexitásban, de itt nem szabad elfelejteni, hogy a rendszerünk már nem csak egy gazda gépből áll, hanem több Kubernetes Node együttese, és azok bármelyikén végrehajtható a szimuláció. Ezáltal az erőforrásainkat a „végtelenségig" lehet növelni, amennyiben szükséges.

Az alapvető probléma, ami felvetődik, hogy a Kubernetes Pod-ok rövid életűnek és eldobhatónak lettek tervezve. Ezáltal a mi szimulációink végrehajtásának a tulajdonságai, miszerint egy nagy erőforrásigényű speciális feladatot hajtunk végre és nem tudjuk felcserélni az egyes szcenáriókat a másikkal, ellentétben állnak a Kubernetes Pod-ok tervezési modelljével.

Ez az ellentét azért áll fent, mert a szimulációs programunk alapvetően monolit természetű. A program alkotóelemeit egy másik megközelítésben lehetséges szegmentálni, hogy azok önállóan tudjanak feladatot végrehajtani és úgy alkossák meg a szimulátor programot. Erre egy példa az lenne, ha az Omnet++ különböző modellkönyvtáraiban megvalósított osztályokat úgy darabolnánk fel, hogy az egyes hálózati rétegetek megvalósító osztályok külön programba kerülnének, és minden állapotukat a háttértáron helyeznék el. Ez azt jelenti, hogy a memóriában végrehajtott műveletek helyett, adott szimulált csomagokat úgy dolgoznának fel, hogy arról csak a háttértáron van információjuk és a végrehajtott feladat kimenetelét is oda mentik el. Ezáltal az adott részfeladatot ellátó egység, állapotmentes Pod-ként futna. Viszont ez a megközelítés több szempontból is hátrányos. A programozási feladatok többszörösére nőnek, mivel egy szimulátor program helyett, minden alkotóelem számának megfelelő külön programot kapnánk. Ezeket karbantartani és fejleszteni és több munka, és a kompatibilitásukat is nehéz fenntartani. Az új szimulátor összetett entitás lenne, aminek az elosztott tárhely megoldása és az elemek összekötése, ugyan már megoldott feladat a Kubernetes-ben, viszont a köztük lévő kommunikációs késleltetés jelentős lenne. Lényegében minden fájlművelet, és az elemek közötti kommunikáció egy virtualizált hálózaton (ami lehetséges, hogy valós hálózat felett létezik, nagyobb késleltetéssel bíró rendszerek között, a Kubernetes klaszter Node-jai ugyanis ajánlások szerint legtöbbször más lokációban

helyezkednek el a georedundancia miatt.) történne. Ez jóval lassabb, mint az eredeti szimulátor program futásam, az Omnet++ szimulátor keretrendszer C++ nyelven íródott, a számítási sebessége miatt pedig a két megoldás közti különbség jelentős lenne.

A keretrendszer és a szimulátor feldarabolása nélkül tehát monolit felépítésű programot futtatunk, a mikroszolgáltatásoknak tervezett rendszerben. A megnövekedett erőforrások miatt, és a jól implementált elosztott tárhely és futtatási rendszerei miatt azonba a Kubernetes még így is jelentős előnyökkel szolgál. Viszont a szimulációk futtatásának ütemezése innentől valós probléma lesz. A szcenárió elvesztése esetén ugyanis jelentős erőforrás pazarlás léphet fel. Jobb nem elindítani olyan szimulációkat, amiket valószínűleg nem fogunk tudni végrehajtani, mivel a félúton erőforrás híján félbeszakadó szcenáriókat elveszítjük, és újra végre kell hajtani őket.

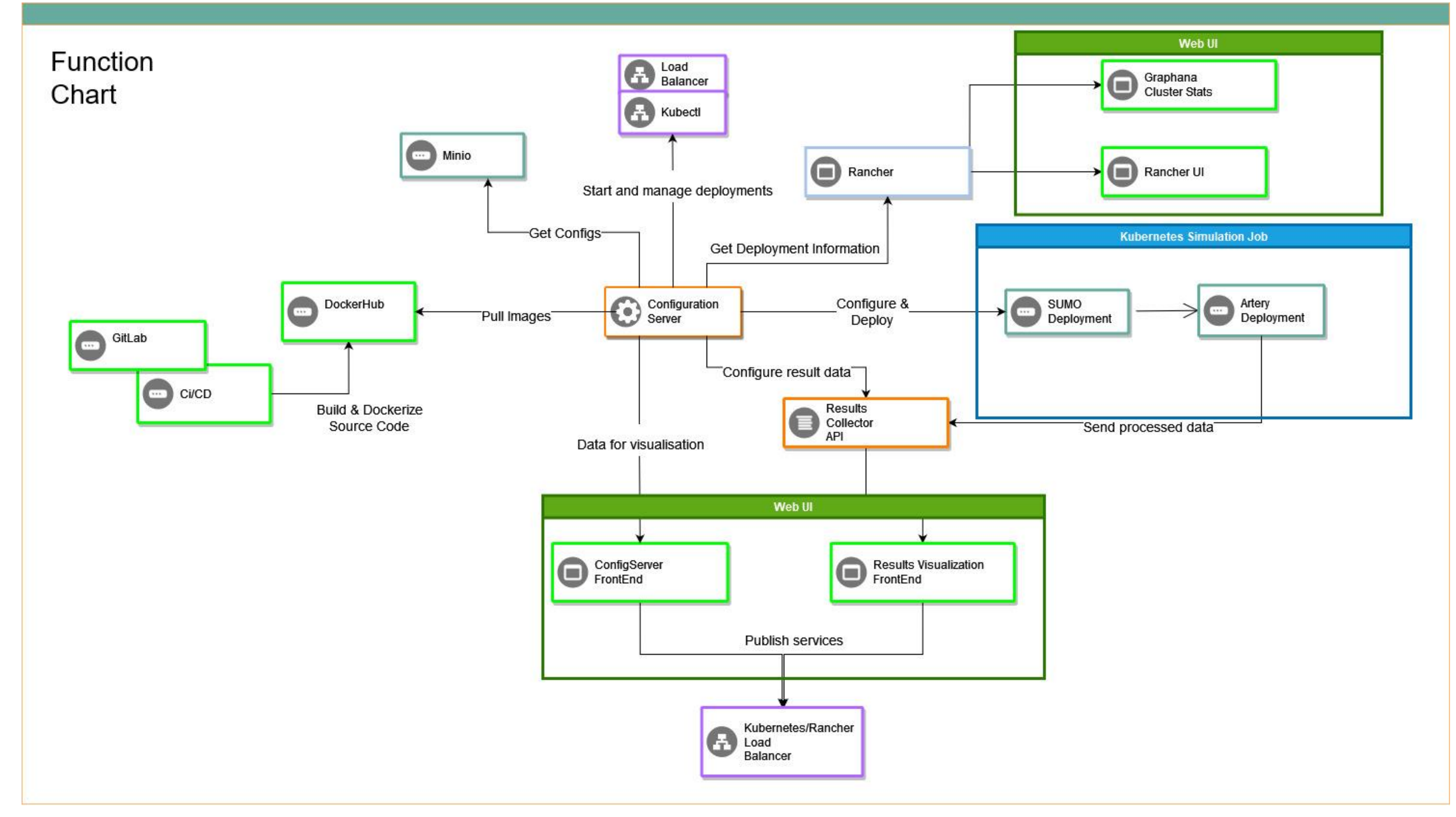
Ez felvet még egy problémát, nehéz megjósolni a szimulációk erőforrásigényét. Egy szimulációs szcenárió során a járművek alapesetben folyamatosan lépnek be és ki a szcenárióból. Ez azt jelenti, hogy a szimulált járművek, és így a szimulált hálózati node-ok száma jelentősen eltérhet a szimuláció egyes pontjain. Tehát a maximális erőforrás igény többszöröse lehet az átlagosnak, szélsőséges esetben. Ez a mi szempontunkból végzetes lehet, mivel a konténerizált futtatás felső korlátját a rendszer memória adja. A processzor idő, mint erőforrás, virtuálisan van megszabva. Tehát a processzoridő túlallokálása csak lassítja a végrehajtást, nem gátolja azt meg.

4 Összetett automatizált folyamatmenedzsment rendszer tervezése

A rendszer alapvető feladatai közé tartozik minden, ami a kód újverziójának a kikerülése és a szimulációk eredményének a feldolgozása és vizualizációja között történik. Röviden a módosított kódból új szimulációt hoz létre, amit képes az előre elkészített szcenárió térképekkel és forgalmakkal többszörösen és párhuzamosan futtatni a felhő architektúrában és az eredményeket feldolgozva azokat elérhetővé teszi a felhasználó számára.

4.1 Rendszer architektúra

A rendszer több komponensből áll össze, a legtöbb magában a Kubernetes klaszter környezetben fut, így ezek is mind skálázhatóak és hordozhatók. A komponensek között találunk nyílt forráskódú programokat és saját applikációkat is. Ezek összekötve tudják a teljes rendszer funkcionalitását kiszolgálni. Mivel minden komponens a klaszteren belül található ezért a szolgáltatások összekötése egyszerűen megvalósítható Kubernetes natív hálózati megoldásokkal.



13. ábra Rendszer komponensek

4.2 Komponensek

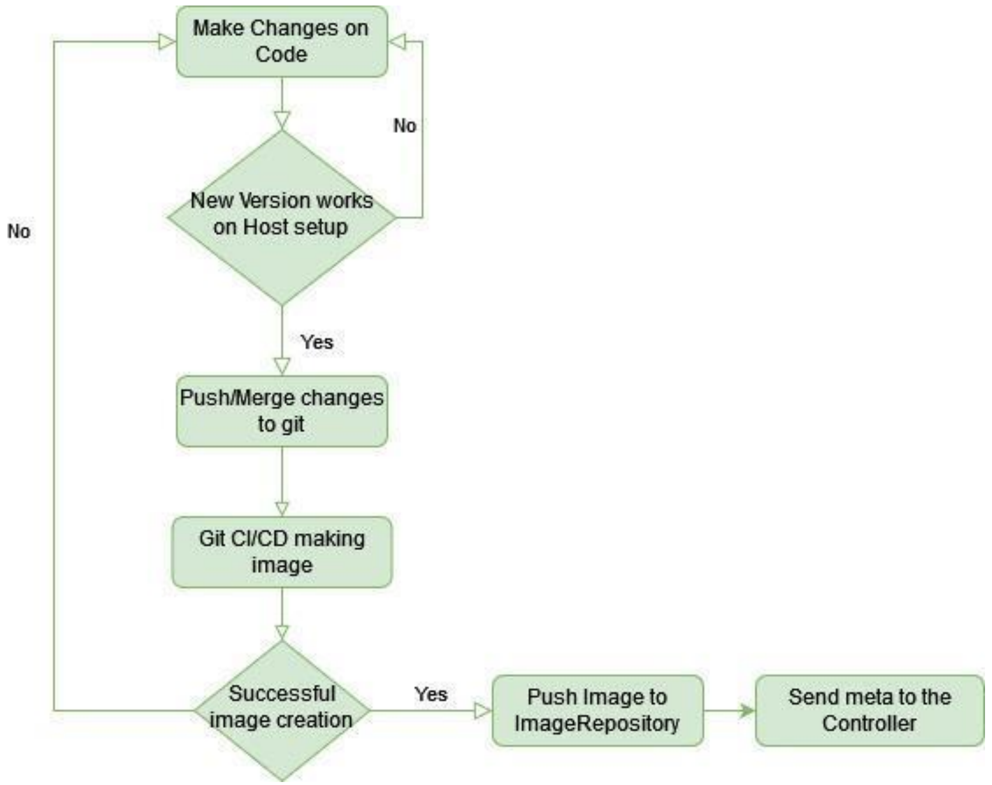
A rendszer komponenseit konténerizált alkalmazások adják, amiket a következő pontokban fejtek ki.

4.2.1 GitLab

A GitLab egy DevOps szolgáltatás, ami verzió kezelő és kollaborációs megoldást is nyújt. A feladata a kódbázis menedzselése és verziókezelése, valamint a CI/CD pipeline alapját szolgáló Yaml fájl alapú megoldás szolgáltatása amit Jenkins alapú CI/CD megoldásra fordít majd, azonba ennek használata jóval egyszerűbb.

4.2.2 CI/CD Pipeline

A Continuous Integration (CI) folyamatos integráció, Continuous Delivery (CD): folyamatos szállítás, egy olyan kódolási filozófia és gyakorlatok összessége, amelyek arra késztetik a fejlesztő csapatokat, hogy kis változtatásokat hajtsanak végre, és gyakran vigyék fel a kódot a verziókezelőkbe. Mivel a legtöbb modern alkalmazás fejlesztése igényli a különböző platformok és eszközök használatát, a csapatnak szüksége van egy mechanizmusra a változtatások automatikus integrálásához és érvényesítéséhez.



14. ábra CI/CD folyamat

4.2.3 Konfigurációs szerver és vezérlő backend szolgáltatás

A konfigurációs szerver alkalmazás az egyik fő saját komponens, ami az egész rendszer menedzseléséért felel. A komponens egy Java Spring Boot alkalmazás, ami REST API-on keresztül szolgáltat információt a Frontend alkalmazás és a többi komponens számára.

A fő logikát ő valósítja meg, ami a szimulációk tárolását, menedzselését és a eredmények gyűjtését is magában foglalja. A szimulációkról tárolja az összes adatot, amit a Kubernetes rendszer szolgáltat. Ezeket a Frontend web alkalmazáson képes a szimulációk valós idejű jellemzéséhez megjeleníteni. A szimulációk konfigurálásához kapcsolódó adatokat is tárolja és képes azokat újra hasznosítani a szimulációk többszöri futtatásánál. Maga a szolgáltatás konténerizáltan fut, a klaszteren belül.

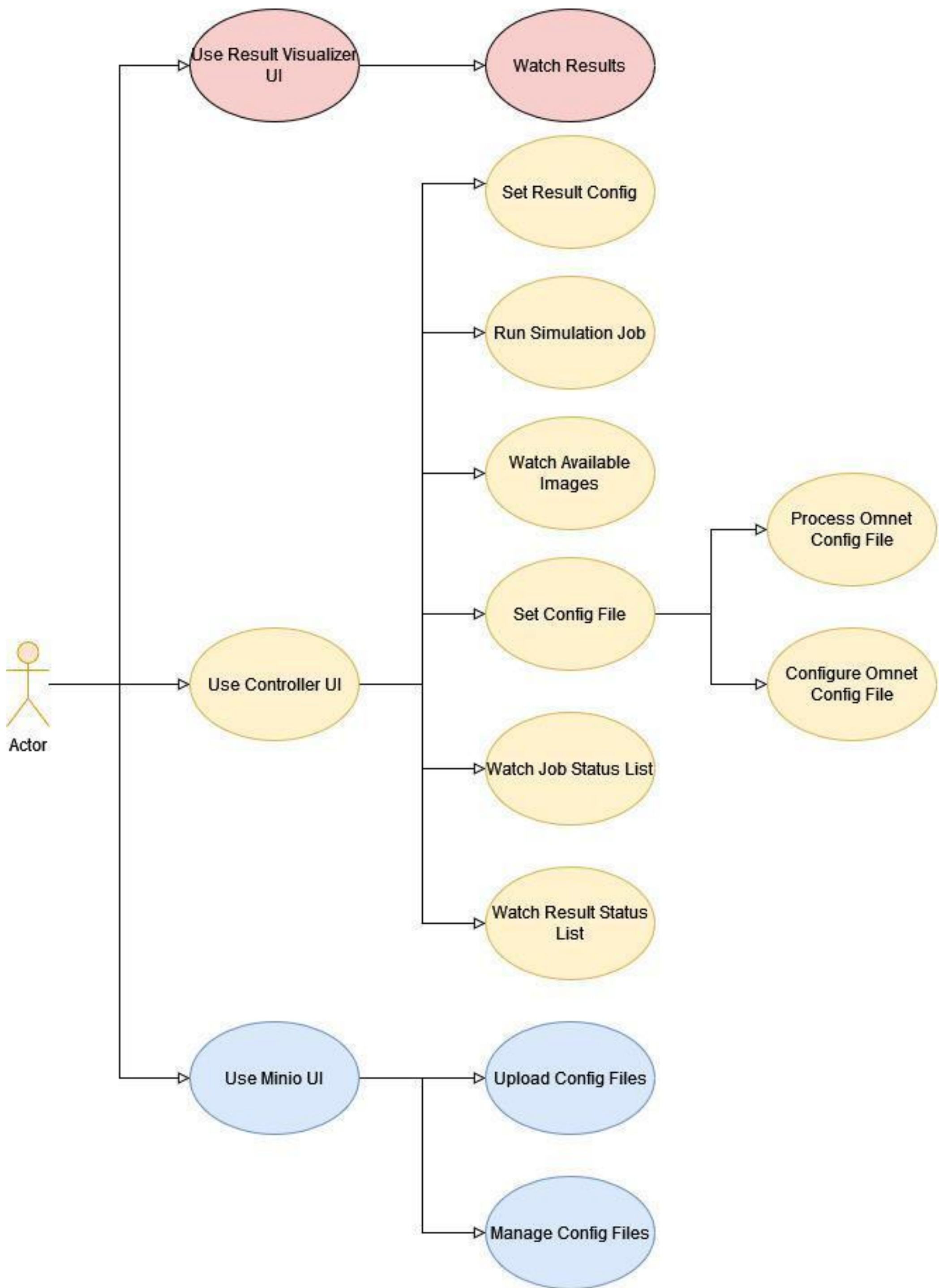
A szimulációk futtatása során a fő feladata a konfigurációs opciókból Kubernetes Job-okat generálni és a konfigurációikat létrehozni. A szimulációk megfeleltetésében az Artery szimulátor többszörösen futtatott szimulációs szcenáriói itt egy-egy Job-ra fordulnak le. Ezeket innentől párhuzamosan indíthatjuk ezáltal növelve a rendszer kihasználtságát és az teljes futási időt.

4.2.4 Konfigurációs szerver Frontend szolgáltatás

A Frontend alkalmazás a szimulációk menedzseléséhez nyújt web felületet. Ez egy React JavaScript alkalmazás, ami szintén konténerizáltan fut a klaszterben.

A felület lehetőséget ad a futtatási paraméterek megadására. Megadható a választott kód verzió Docker Image, a szimuláció szcenárió paraméterek és a hozzájuk tartozó fájlok. Az Omnetpp fájl is ezen keresztül kerül feltöltésre, ami a fő konfigurációs entitás.

Megjeleníti az elérhető információkat a jelenleg futó szimulációkról interaktív formában. A szimuláció kimenetek állapotát is láthatóvá teszi.



15. ábra Webalkalmazás felület lehetőségei.

4.2.5 Monitoring

A Kubernetes komponensek különböző metrikáit és statisztikáit megfigyelhetjük egy Third Party komponens segítségével, ami az összes telepített Kubernetes Deployment-ről tud adatokat gyűjteni. Ezt a Prometheus program szolgáltatása teszi lehetővé, ami a választott Rancher keretrendszerben egyszerűen telepíthető a klaszterben. Ezt egy Grafana szolgáltatással

integrálva tesszük interaktívvá és informatívvá ami látványos grafikus felületet biztosít az adatok vizualizációjához.[21]

16. ábra Grafana monitoring ábra[21]

4.2.6 Szimulációs szolgáltatás

A szimulációs Deployment egy pod-ból álló rendszer, ami egy előre elkészített konténer Image-ből készül. Ez tartalmazza az éppen futtatott kód verziót és így a szimulációt is. A Deployment maga Job-okból áll, amiket előzetesen generált a Konfigurációs backend szolgáltatás.

A konténerben az Artery (vagy Simu5G) Simulator natív megoldásának a másolata van telepítve a SUMO megfelelő verziójával és a megfelelő csomagokkal. A konténerekből egyszerre több is futhat és végső soron ezeknek a lefutása adja majd a teljes szimulációs szcenárió egészét.

A szimulációs szolgáltatásokat úgy kell generálnunk és végrehajtanunk a Kubernetesben, hogy azok külön konfigurációkkal hajtsák végre a szimulációs szcenáriót. Ehhez szükséges, hogy minden szimulációs Pod elérje a konfigurációjában hivatkozott konfigurációs fájlokat, tehát az elosztott tárhely megoldást kell alkalmaznunk. A végeredményeiket egyenként meg kell tudnunk különböztetni, viszont az egy szimulációhoz

tartozó eredményeket össze kell tudnunk kötni. Ezeket az eredményeket szintén elosztott tárhely megoldással kell perzisztens módon tárolni egy számunkra is elérhető helyen. Illetve olyan helyen, ahol az eredményeket feldolgozó szolgáltatás is eléri azokat.

4.2.7 Sidecar Konténer

A sidecar konténer séma egy bevált és ismert Kubernetes megoldás arra, hogy a futó konténerekből adatokat nyerjünk ki egy csatlakoztatott konténerrel. Ezen konténer általában hozzáfér egy közös tárhelyhez, Volume-hoz ami segítségével a futó konténer adatait tudja kezelni. Ezeket az adatokat ezután bárhogy felhasználhatjuk anélkül, hogy az eredeti konténerbe kellene bármilyen megfigyelő programot vagy hasonló változtatást eszközölni. A megoldást sokszor Service Mesh megoldásoknál használják.[22]

Esetünkben nem biztos, hogy a használata előnyös lesz, a valódi erőssége a log-ok aggregálásának segítésében lehetséges számunkra, amennyiben megvalósítunk ilyet.

4.2.8 Eredmény gyűjtő szolgáltatás

Az eredmény gyűjtő az összegyűjtött eredmények aggregálásával és feldolgozásával foglalkozik. Ez egy Python Django Rest API-t megvalósító program, ami második backend szolgáltatásként funkcionál. A Konfigurációs backend szolgáltatásunk megvalósítása során a jól kidolgozott Kubernetes Java könyvtár indikálta a Java nyelv használatát. Az elgondolásunk itt is hasonló, az interops megoldásokat elkerülendő, natív környezetükben szeretnénk futtatni a programkódokat. Az Omnet++ eredményfájlok feldolgozására és kiértékelésére pedig a Python Scientific könyvtárak, mint Pandas, Numpy és Matplotlib adnak lehetőséget. Emiatt ezek integrálása egyszerűbb és letisztultabb feladat egy Python API számára.

Az eredmények feldolgozásához, azokat egy formázott csv kiterjesztésű fájlba kell írnunk. A nyers vektor adatok formázása „csv” fájlokká az Omnet++ saját Scavetool scriptje segítségével történik. Ezt integrálva képesek leszünk továbblépni az eredmények értékeléséhez és vizualizálásához.

A csv fájlokat feldolgozzuk a megfelelő adatfeldolgozó alkalmazásban. Az adatokat ezután felhasználjuk valamilyen vizuális vagy szöveg alapú visszacsatolásra, amivel a fejlesztő, vagy tesztelő képes levonni a következtetéseket a szimulációról.

4.2.9 Eredmény megjelenítő frontend szolgáltatás

Az eredmények vizuális megjelenítését végzi. Ez a Konfigurációs Frontend-hez hasonló szolgáltatás, ami React Javascript alapon. Ez ad lehetőséget a vizsgálni kívánt szimulációk eredményeinek kiválasztására és megjeleníti azt.

4.2.10 Támogató komponensek

A rendszerünk működéséhez van néhány elengedhetetlen és néhány kényelmi funkciót ellátó szolgáltatás, amiket a teljes rendszerbe integrálunk. Ezek Thrid Party alkalmazások, amiket a Kubernetes környezetben már ismernek és alkalmaznak. Így kiforrott megoldásokat kapunk, viszont a konfiguráció, klaszterbe telepítés, és a komponensek összekötése a mi feladatunk.

Az első ilyen dolog egy MySQL adatbázis, ami a két backend alkalmazásunk tárhelyeként szolgál. A programokat gyakran fejlesztési környezetben más adatbázis megoldásokkal használjuk, de „production ready” környezethez kell egy valós adatbázis. Ez a program elérhető kell legyen a klaszterben, és a tárhelyének perzisztensnek kell lennie, a hosszútávú működéshez.

A második már kényelmi funkciót valósít meg, ez egy phpMyAdmin Deployment, ami a MySQL adatbázis kezelését könnyíti meg. Perzisztens tárhelyet nem igényel, mivel csak alap konfigurációkat kell megadni, hogy a MySQL adatbázist tudja menedzselni, ami valós adatokat tárol. A fejlesztést és hiba javítást minden esetben megkönnyíti és jól átlátható felületet ad.

MinIO elosztott tárhely megoldás. A MinIO program egy felhő alapú tárhelyalkalmazás ami a Kubernetes rendszerhez tud elosztott tárhelyet szolgáltatni. A mi felhasználásunkban csak a fájlkezelő interfészét fogjuk használni, a szolgáltatásaink által használt elosztott tárhelyek menedzselésére. Ez a háttértár állapotának vizsgálatát segíti, anélkül, hogy a Node-ok tárhelyéhez kellen hozzáférnünk.[23]

4.3 További rendszer tervek

4.3.1 Ütemező Funkció

A Kubernetes rendszer jellemzői és a szimulációs rendszer jellemzői egy komoly feladat elé helyezik a rendszert. A felhő alapú szolgáltatások sok alapvetően azonos feladatot elvégző

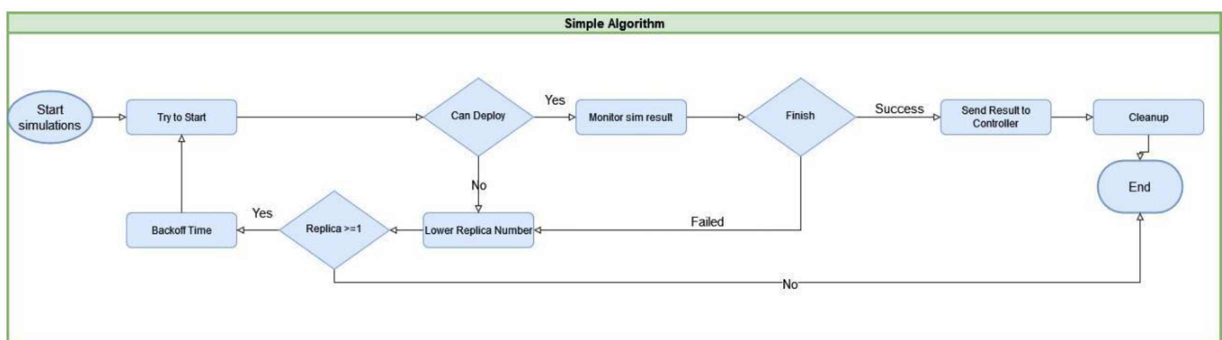
komponens futtatására lett kitalálva. Ezek erőforrásigénye ugyan változó lehet, de az erőforráshiányt a komponensek skálázásával orvosolják.

A szimuláció maga viszont erősen változó erőforrásigénnyel rendelkezik a különböző konfigurációk alapján. Emiatt nem megoldható a komponensek skálázása mivel a program alapvetően monolitikus. Ez azt jelenti, hogy az elindított szimulációk párhuzamosíthatósága és a sorrendje is számít a teljes szimulációs szcenárió futtatásánál. A rendszer alapvető korlátját a memória igénye jelenti, mivel a szimuláció hibásan nem fog lefutni amennyiben a memória igénye nagyobb, mint az elérhető.

Ennek a viselkedésnek az orvoslására és optimalizálására egy saját Scheduler, ütemező megoldást adhatunk, ami számol a sajátos alkalmazás tulajdonságainkkal.

A különböző feladatokhoz gyakori, hogy sajátos ütemező használata, például gépi tanuláshoz vagy más specifikus feladathoz általában szükség van saját erőforrás elosztásra. A különböző feladatok különböző megoldásokat szülnek és ez által még optimálisabb szolgáltatásokat hozhatunk létre.[24]

Erre egy példát ad a következő algoritmus:



17. ábra Scheduler Algoritmus példa ábra.

Az algoritmus az egyszerű vak próbálkozásra alapul. Addig növeli a párhuzamosan futtatott szimulációk számát ameddig azok el nem kezdenek hibával visszatérni. Ekkor csökkenti az párhuzamosítani kívánt szimulációk számát addig, ameddig nem tud sikeren szimulációs eredményt elérni. Ekkor újra indul a folyamat növelése.

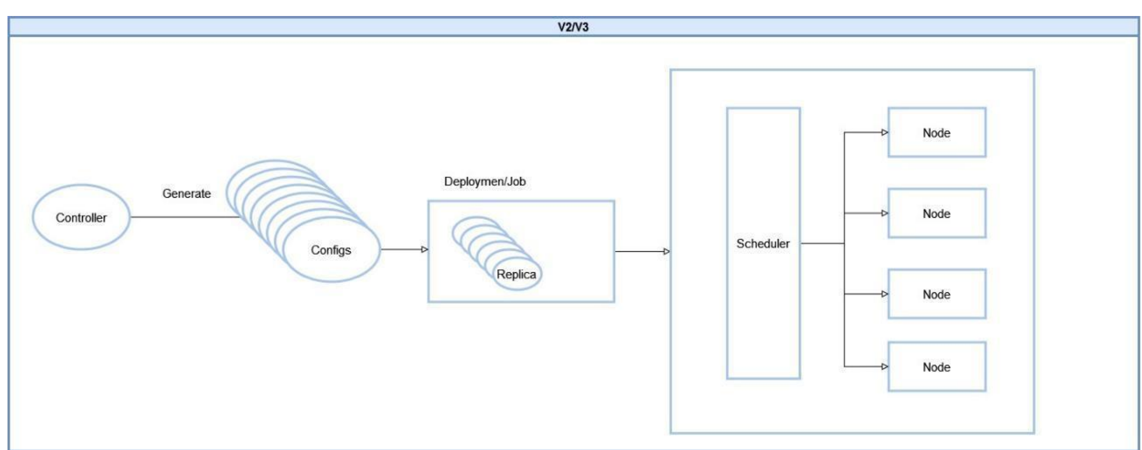
Ez egy egyszerű példa a szimulációk párhuzamosított futtatására, más algoritmusokat és alkalmazásokat is átgondolhatunk.

4.3.2 Szimulációk megfeleltetése Kubernetes Deployment-ekre

Az Artery szimulátor az Omnetpp.ini fájlban megadott paraméterek alapján képes összetett kampányokat csinálni a futtatandó szimulációs szcenáriókból. Ezt azonban sorban képes elvégezni és mint pont ezt akarjuk a felhőben párhuzamosan futtatni optimalizációs szempontok miatt. Ehhez a szimulációkból a Kubernetes-ben futtatható entitásokat kell létrehozni.

Erre több megoldás is létezik, de mindegyik alapja az, hogy az eredeti konfigurációs fájlban definiált szcenáriókat kibontjuk és a kombinációkat egy-egy szcenárió konfigurációs fájlban írjuk le, tehát minden esetet generálunk, amit tudunk az eredeti konfigurációs fájlból.

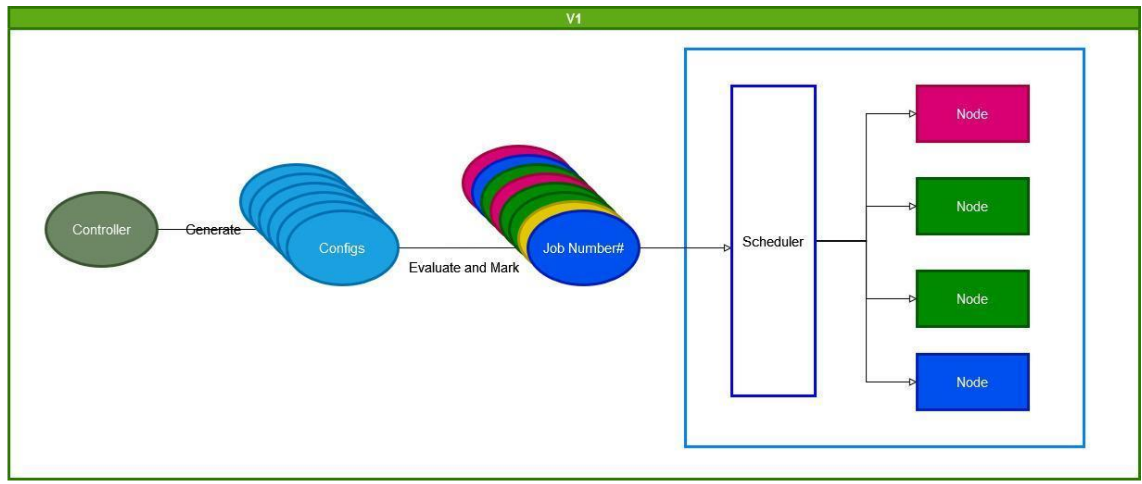
Lehetséges egy végrehajtó Deployment-ben definiálni magát a futtató kód verziót, aminek a replikációs számát menedzselve végezzük a szimulációk futtatását. Ennek a hátulütője, hogy magát a konténerben futó szimulátort kell valahogy megtámogatni egy nem Kubernetes natív megoldással a konfigurációk szétosztásához. Például a minden Pod számára látható tárhelyen lévő konfigurációkból kiválasztja a neki szükségeset, de ehhez is információra van szüksége valamilyen külső féltől.



18. ábra Deployment szimulációs konfiguráció

A másik megoldás, hogy minden konfiguráció generálása után, ezekhez egyedi Kubernetes natív Job-okat definiálunk. Ezeknek tehát egy konfigurációs szcenárió futtatása lesz a feladata és a saját Kubernetes Configmap leírásban lesz eldöntve melyiket kell használni. Másik előnye a megoldásnak, hogy tudjuk őket külön kezelni és Scheduler segítségével meghatározott Node-okon végrehajtani.

Itt akár arra is lehetőségünk van, hogy a Node-ok teljesítményét figyelembe véve külön Taint-Toleration megoldásokkal vezényeljük a szimulációkat a megfelelő helyre.



19. ábra Jobs szimulációs konfiguráció.

Az ilyen módon vezérelt végrehajtás a homogén Node-okkal rendelkező rendszereken hasznos lehet, az erőforrások kihasználásának optimalizálásához. Az egyes szcenáriók erőforrásigényének beállítása viszont nehéz feladat.

Az egyes szcenáriókról végrehajtás előtt csak korlátozott információink vannak. A konfigurációs fájlok tartalmát feldolgozni és abból meghatározni a szimulációink erőforrásigényét, az a szimulációk végrehajtásával egyenértékű nehézségű feladat. Így valamilyen találgatással kell meghatározni ezt az értéket. A konfigurációs fájlok méretének vizsgálata adhat hasznos támpontot. Az előzetes vizsgálatok során kiderült, valójában a járművek száma, és igazából a szimulálandó hálózati node-ok száma az, amit erősen befolyásolja a szimuláció futási idejét, ez indikálhatja a szimuláció erőforrásigényét is. Tehát a szcenárióban használt konfigurációs fájlok méretéből adhatunk arányszámokat, az erőforrásigényre.

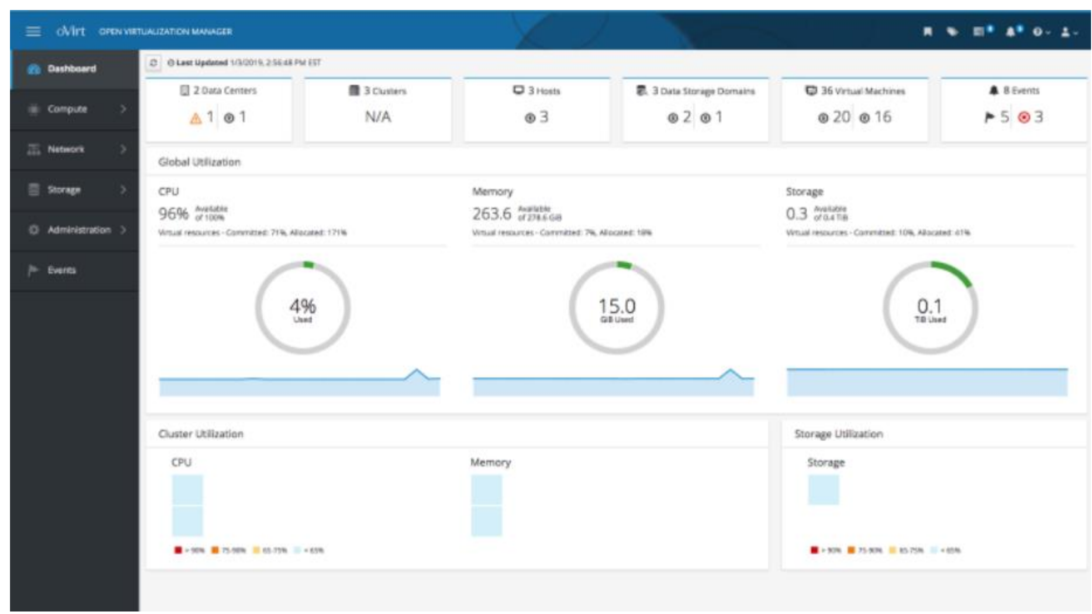
A másik megoldás, a tanulás alapú tapasztalati jóslás. Amennyiben meghatároztuk, melyik szimuláció lesz a legnagyobb, illetve legkisebb erőforrásigényű, az egyik lefuttatása után az eredmények értékelésével kaphatunk információt a legrosszabb vagy legjobb erőforrásigényű esetünkről. Az arányszámainkat használva ez már fejlettebb előrejelzést adhat.

5 Kubernetes alapú felhő infrastruktúra létrehozása

5.1 Virtualizáció

A Kubernetes környezet kialakítását és tesztelését valós ipari környezetben, a Datatronic Kft. rendszerein végezhettem. A cég hosting megoldásokkal már régóta foglalkozik, az utóbbi időkben pedig Kubernetes technológián alapuló felhő hosting szolgáltatásokat is nyújt.

A Datatronic Kft. cég infrastruktúrájában a virtuális gépeket KVM (Kernel based Virtual Machine) technológiával felügyelik. A virtuális gépek menedzseléséért pedig az Oracle Linux Virtualization Manager felelős. Ebben az infrastruktúrában nagy számban lehet létrehozni és menedzselni a különböző virtuális gépeket és virtuális hálózatokat, a valós szerver gépek klaszterein. A hiperkonvergens architektúrával rendelkező valós szervereken a használt technológiák révén a Kubernetes számára megfelelő infrastruktúrát lehet létrehozni.



20. ábra Ovirt menedzser grafikus felület[25]

5.2 Virtuális Host a Kubernetes számára

A Kubernetes klaszter létrehozásához az első lépés, a megfelelő virtualizált infrastruktúra létrehozása. A „csupasz” szervereken futtatott Kubernetes helyett, a szerveken elérhető virtualizált host réteg, ami a Kubernetes klaszter node-jaiként szolgál. Ennek számos

előnye van. A virtualizált réteg lehetővé teszi, hogy dinamikusan hozzuk létre és menedzseljük a klaszter node-okat és a hálózatokat közöttük.

A virtuális host-okat több módon létrehozhatjuk. Az infrastruktúra menedzseléséért felelős Oracle Linux Virtualization Manager webes felületén keresztül grafikus interfész segítségével alkothatjuk meg a host virtuális gépeinket. Emellett viszont ennek a programnak több API alapú vezérlése elérhető. A számomra legkézenfekvőbb megoldást a Python SDK [26] alkalmazása adta, mivel a klaszter létrehozása során többször újra kellett alkotni a rendszer node-jait. Ennek az ismételt folyamatnak a megkönnyítése végett egy virtuális gépeket menedzselő scriptet hoztam létre, ami parancssorból könnyen konfigurálható, és képes több egyező konfigurációjú virtuális gépet létrehozni. Ezeket képes egy hálózati interfésszel ellátni és elindítani. Ezáltal az egyszer megalkotott „template” ként szolgáló virtuális host-ot sokszorosíthatom.

```
# Create a new virtual machine, cloning it from the snapshot:
cloned_vm = vms_service.add(
    vm=types.Vm(
        name=myVmName,
        snapshots=[
            types.Snapshot(
                id=snap.id
            )
        ],
        cluster=types.Cluster(
            name=myCluster
        ),
        memory=myMemory,
        cpu=newCpu
    )
)
```

21. ábra Pythond Ovirt SDK példa

A Python scriptek használata kielégítő volt a számomra, viszont a karbantartásuk nehézkes volt és bármi nemű fejlesztéshez is átfogó tudásra volt szükség. Emiatt használható Ansible technológia, ami deklaratívan képes az ilyen rendszerek menedzselésére, jóval fejlettebb és kiforrottabb megoldásokkal érhetünk el széleskörű funkcionalitást.

A virtuális gépek, amiket létrehoztam Oracle Linux 7 operációs rendszerrel futottak. Ezt az operációs rendszert azért választottam, mert a cégnél ez egy ismert és használt operációs rendszer volt. A virtuális gépek egy virtuális alhálózatra kerültek, azért, hogy elérjék egymás

privát hálózati úton és az esetleges Kubernetes által generált forgalom házon belül maradjon. Ennek performancia és adatvédelmi okai is voltak.

A virtuális gépeken alapvetően nem sok változtatást inkább konfigurációkat kellett eszközölni a Kubernetes-be integráláshoz. A gépek hálózati interfészeit kézzel egy shell script segítségével végeztem, ami a virtuális interfészek beállítását végezte. A Kubernetes kapcsolatot gátolta még a SELinux modul, aminek használatától emiatt eltekintettem, tehát deaktiváltam. Az utolsó és legfontosabb része a virtuális host-nak a Docker Runtime telepítése volt. Ez adja az alapját a Kubernetes rendszernek. Azóta, az újabb verziókban már Containerd alapú runtime-ra épít a Kubernetes.

5.3 Kubernetes rendszer

A Kubernetes node-jai megalkotása után, egy jó módszer kellet a klaszter telepítésére. Nem térek ki részletesen a Kubernetes rendszer alkotó elemeire, de összetett és elosztott rendszer lévén a telepítés feladata nem mindig egyszerű. A legtöbb esetben ezeket a rendszereket online használjuk és béreljük a nagyobb szolgáltatóktól, mint a Google GKS, Amazon AWS és a Microsoft Azure. Az ilyen esetekben úgynevezett menedzselt Kubernetes rendszert kapunk. Ezzel szinte minden karbantartási feladat áthárul a szolgáltatókra és mi fókuszálhatunk a fejlesztésre és a szolgáltatásra.

A mi esetünkben azonban egy saját megoldásra volt szükség, hogy a mi node-jainkat össze kössük. A cégnél már ismert Rancher Kubernetes menedzsment rendszert használtam a telepítéshez

5.3.1 Telepítési módok

A Rancher alapvetően egy menedzsment rendszerként indult, viszont a program sikere után saját Kubernetes motort hoztak létre. Két megoldást adnak a kezünkbe, az egyik a K3s egy kisméretű „lightweight” Kubernetes motor [27] ami a kis erőforrásigénye miatt előnyös, a másik az RKE Kubernetes motor [27] ami pedig az egyszerű karbantartása és telepítése miatt hasznos.

5.3.1.1 K3s

A két rendszer funkció szempontjából azonos Kubernetes klasztert hoz létre. A K3s bár erőforrásigénye kisebb, a telepítés és karbantartás szempontjából nehezebben használható. A Kubernetes adattárolásához használt elosztott adatbázist a mi feladatunk létrehozni és

szolgáltatni a K3s számára. Ez lehet MySQL vagy más adatbázis, de a legkézenfekvőbb az etcd kulcs-érték alapú adattároló használata [28]. Miután létrehoztuk és elindítottuk a rendszert a működése során nincs eltérés a többi Kubernetes rendszerhez, de új vezérlő node-ok csatlakozása esetén az elosztott tárhely node-jait is bővítenünk kell.

5.3.1.2 RKE

A másik megoldás, amit vizsgáltam, és végül használtam az az RKE Kubernetes motor volt. Ennek a megoldásnak a fő ismertetője, hogy konténerizáltan futtatja az összes Kubernetes komponenst a host gépeken, még az etcd adattárolót is. Emiatt a telepítése igen egyszerű, hiszen a megadott node-okra mindent konténerizáltan tud telepíteni.

A telepítés maga elég egyszerű, egy yaml fájlban kell megadnunk deklaratív módon a Kubernetes rendszer elvárt állapotát. A fő elemei a node-ok és azok szerepének felsorolása. Ezekre SSH elérést kell biztosítani, tehát első lépésként létre kell hozni egy felhasználót a host gépeken, akinek van Docker vagy más konténer runtime rendszerhez való hozzáférése, majd ennek a felhasználónak kell az SSH-eléréshez szükséges kulcsait eljuttatni az RKE menedzsment gépre, ahol magát a szoftvert futtatni fogjuk. A szoftver maga, ezután belép a node-okra és a szerepüknek megfelelő komponenseket telepíti és kialakítja a kapcsolatot közöttük a megfelelő agent-ek telepítésével.

```yaml
nodes:
    - address: 10.2.108.102
      user: Rancher
      role:
        - controlplane
        - etcd
        - worker
      hostname_override: master1
      taints:
        - key: CriticalAddonsOnly
          value: True
          effect: NoExecute
    - address: 10.2.108.100
      user: Rancher
      role:
        - worker
      hostname_override: worker1
    - address: 10.2.108.101
      user: Rancher
      role:
        - worker
      hostname_override: worker2

cluster_name: local

addon_job_timeout: 120
```

22. ábra Három node konfigurálása RKE yaml fájllal

Ahogy látható az ábrán, a node-ok leírása egész egyszerűen elvégezhető pár sorban. Ez azért lehetséges, mert az RKE alapbeállításokat szolgáltat a legtöbb konfigurációjához. Ha minden konfigurációt szerepeltetünk akkor egy teljesen production ready Kubernetes konfigurációt hozhatunk létre, a mi esetünkben az alap beállítások viszont megfelelnek.

Ezután az RKE binárist telepítve egyszerűen létrehozható a Kubernetes klaszter. Egyetlen „rke up" parancs kiadása a klaszter yaml fájl könyvtárában elindítja a programot ami az elvárt állapotba hozza a klasztert. Ezután bármilyen módosítást szeretnénk, a yaml fájl módosításával és a parancs újra futtatásával képesek vagyunk végrehajtani azokat. Amennyiben a program sikeresen végrehajtotta feladatát, kapunk egy állapotleíró „rkestate" fájlt, és egy Kubernetes konfigurációs yaml fájlt, amivel elérhetjük az újonnan létrejött klasztert. Ezt általában kubectl program használatához használhatjuk.

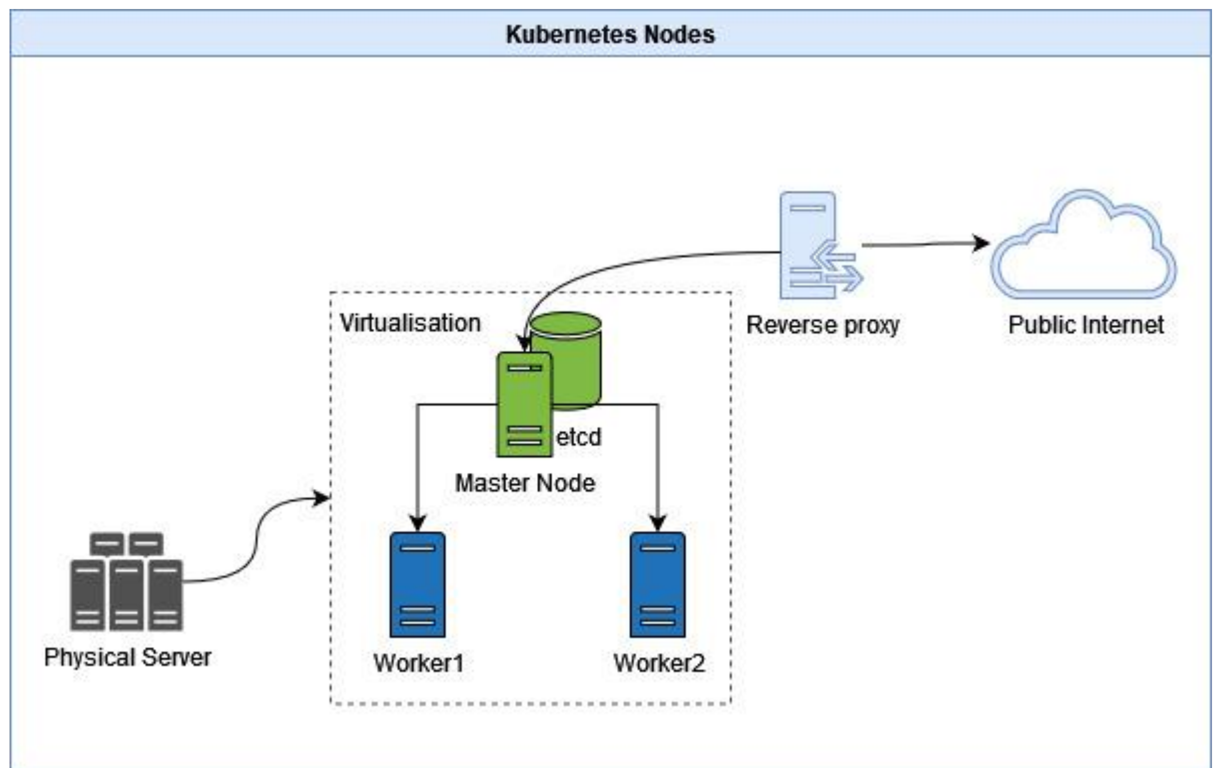
A kubectl egy Kubernetes menedzsment eszköz, amivel a klaszterben hajthatjuk végre a szolgáltatásainkat. Ez a fő parancssori eszköz a klaszter használatához.

5.3.2 Kubernetes node-ok típusai

Az előző pontban látott klaszter létrehozásakor különböző szerepkörű node-okat hoztunk létre. A klaszter futásához elengedhetetlen vezérlő és feladat végrehajtó node-okat, aminek a mi esetünkben két fajtája van.

Az úgynevezett master node, a kontroll entitás a Kubernetes-ben. A Kubernetes szerepkörei a „controllplane” és „etcd”. Ez azt jelenti, hogy a kontroll feladatokat és a klaszter állapotának elosztott tárolásáért is ő a felelős. Anélkül, hogy a pontos komponensek listáját megvizsgálnák tudhatjuk, hogy lényegében ezek a node-ok a Kubernetes klaszter működéséért felelnek. Az RKE motor, ezekre a node-okra telepíti az etcd komponenseket konténerizálva, ezáltal a magas rendelkezésre állásért felelős komponensek is itt vannak.

A másik node fajta, amit használunk a „worker” node. Ennek a feladata a Kubernetes „worker” szerep elvégzése, a szolgáltatások futtatása. Ez azt jelenti, hogy a Kubernetes klaszter működéséért alapvetően nem felel, ha egy ilyen node kiesik, bármikor pótolhatja egy másik „worker” szerepű node anélkül, hogy a Kuberenetes klaszter működését zavarná.



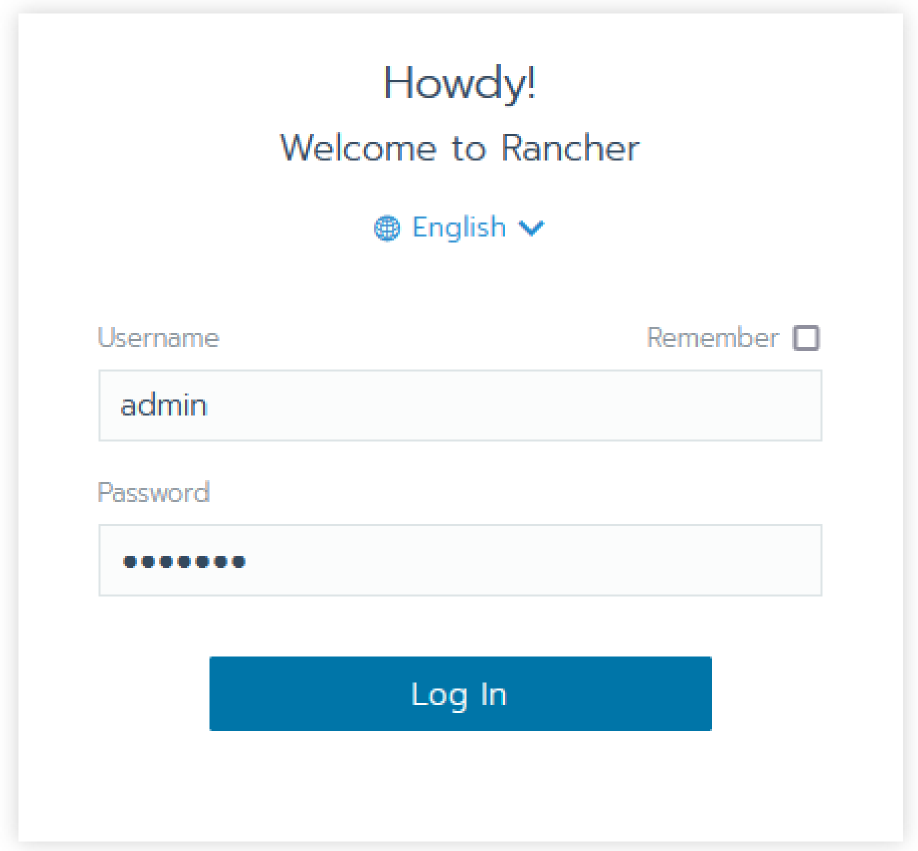
23. ábra Kubernetes Node architectúra

A Kubernetes szerepek bármilyen formában kombinálhatóak a node-okon, viszont logikus elgondolások szerint a vezérlő elemeket egy helyen tartani, és tehermentesíteni a szolgáltatások erőforrásigénye alól egy bevett szokás. A mi esetünkben a „master” node-okon

is van „worker” szerep, ennek oka egy infrastruktúra specifikus megoldás volt. A megoldás tömören abból állt, hogy a klaszter külső IP-címe, amit esetleg használni tudtam a „master” node-on volt felkonfigurálva. Ide volt a forgalom bekapcsolva a külső hálózatok felől. Ezután a forgalom irányítást a Kubernetes Ingress oldotta meg a többi node-között, azonban az Ingress csak a „master” node-on keresztül tudta felvenni a külső IP-címet. Ezért a mi esetünkbe a „master” node is képes a szolgáltatásainkat végrehajtani, de a megfelelő Node Taint alkalmazással, csak a működéshez szükségeseket engedélyezzük. Ez a Critical Addons Only Taint ami megköveteli, hogy csak a kritikusnak megjelölt szolgáltatások induljanak el ezeken a node-okon.

5.4 Rancher

A Rancher mint már említettük menedzsment felületként indult, és a jó grafikus felülete mellett a sok előre konfigurált és telepíthető komponens adj az erejét. A Kubernetes-ben végrehajtható akciók közül mindent támogat, és még sok saját megoldást is ad, például Kubernetes Service-ek újfajta Rancher specifikus Deployment-hez kapcsolására.



24. ábra Rancher bejelentkezés

A Rancher saját Role Based Authentication rendszerrel bír. Ez segít abban, hogy több fejlesztő csapatot egymástól elkülönítve tarthatunk a rendszerben, akiknek a hozzáférését és jogait is menedzselhetjük. Tehát vállalati környezetben alkalmazható igazi előnyökkel járó alkalmazást ad.

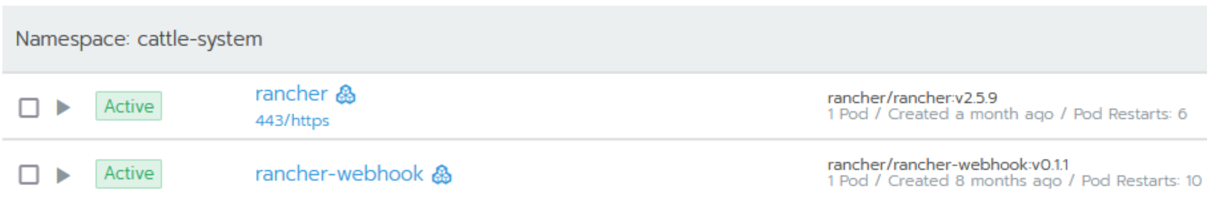
Fő felülete a klaszterünkről és a szolgáltatásainkról nyújt adatokat.

25. ábra Klaszter áttekintés

A saját szolgáltatásainkat a Kubernetes által használt yaml fájlok részletes ismeret nélkül is létrehozhatjuk és végrehajthatjuk. Ez sok segítséget ad a Kubernetes nem ismerő fejlesztőknek akik alkalmazkodni akarnak ehhez a környezethez.

5.4.1 Rancher telepítése

A Rancher maga is a klaszterben fut. A telepítését a Helm csomag menedzserrel végezhetjük el, ami a Kubernetes szolgáltatásokat csomagként képes verziókezelten menedzselni és telepíteni a klaszterünkbe. A Helm-et a menedzsment virtuális gépen futtatva, a kubectl elérés beállítása után tudjuk használni.



26. ábra Klaszterben futó Rancher

A host gépeken is lehet Rancher-t futtatni, illetve Docker konténerben is, viszont a magas rendelkezésre álláshoz ajánlott a Ranchert magát is a Kubernetes klaszterbe telepíteni. Itt vonatkozik rá minden más szolgáltatásra felsorolt előny, skálázódás, helyreállás stb.

A telepítés előtt a Helm segítségével egy tanúsítvány menedzser telepítése is szükséges, viszont ezt is pár paranccsal megtehetjük.

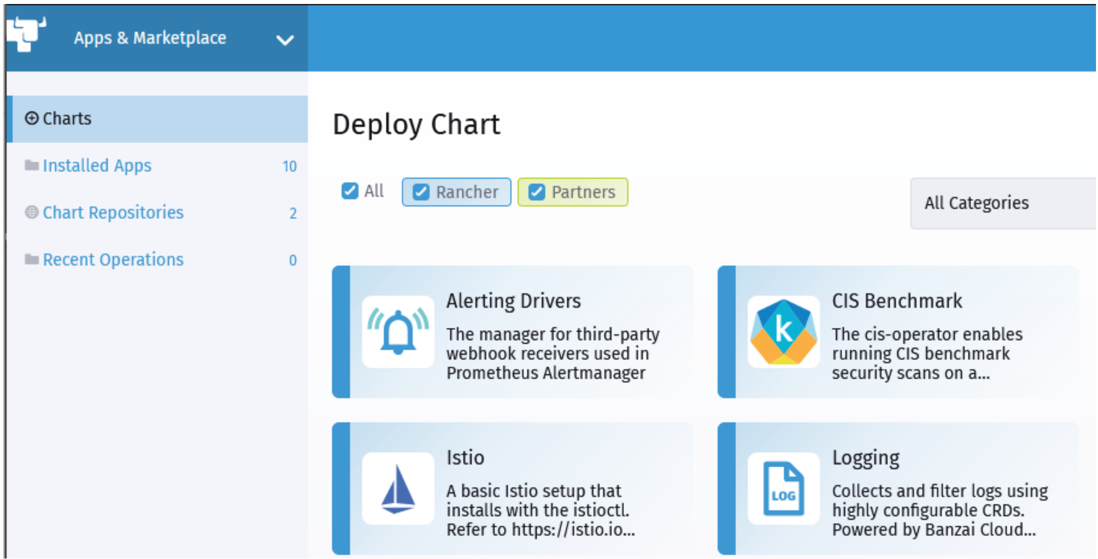
```
helm install \
  cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --version v1.5.3

helm install rancher rancher-stable/rancher \
  --namespace cattle-system \
  --set hostname=rancher.rke.com
```

A két parancs végrehajtásához a megfelelő Helm repository-k hozzáadás és frissítése szükséges, illetve a Kubernetes névterek létrehozása. Viszont ezután a használható a felület a szolgáltató node-ok, illetve a „master” node-okon keresztül.

5.4.2 Rancher kiegészítők

A Rancher több alkalmazást is képes felületről telepíteni, ami segítheti a munkánkat. Ezeket saját Repository alkalmazásával teszi meg, és a Rancher beépített Helm segítségével oldja meg.



27. ábra Rancher beépített applikációk

5.4.3 Elosztott tárhely megoldás alkalmazása

A Kubernetes elosztott tárhely megoldást igényel. Ezt sok módon meg lehet valósítani, a Rancher saját Longhorn nevű megoldása is ezt teszi lehetővé. A Longhorn vizsgálata közben viszont az erőforrásigénye hatalmas volt, a kis méretű Kubernetes klaszteren. Ezért a Datatronic Kft. GlusterFS alapú megoldását használva, a virtuális host-ok számára adunk elosztott fájlrendszert. Ezáltal a virtualizáció szintjét a host-ok alá mozgatva a Kubernetes HostPath Storage Class megoldását használhatjuk. Ennek hátulütője, hogy alapvetően megköti a szolgáltatás futtatási helyét, egy host gépre, tehát egy Kubernetes node-ra. Ezt teszi amiatt, mert az ő információja szerint az adat csak a host gépen van jelen. A LocalPath Provisioner módosításával azonban lehetséges ezt a megkötést kiiktatni és így használni mint egy valós Storage Class, segítve a munkánkat.

6 Összetett Kubernetes alapú rendszer megvalósítása

6.1 Menedzsment backend szolgáltatás

6.1.1 Adatmodell és API

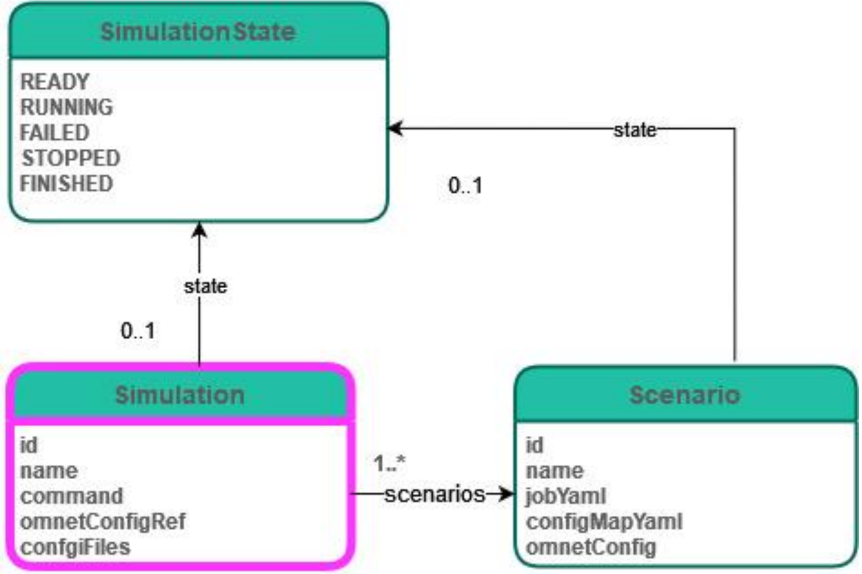
A tervekben elképzelt backend szolgáltatásunk fő feladata, információkat szerezni a Kubernetes klaszter állapotáról és azon belül is a szolgáltatásainkról, ezek mellett pedig a konfigurálható szimulációkat leképezni, tárolni és futtatni a Kubernetes klaszteren.

A megvalósítás egy Java Spring API megalkotásával történt. Az első dolog, ami feladatként felmerült, az a klaszter és szolgáltatások állapotának számontartása. Ennek adatmodelljét annyira egyszerűen terveztem meg amennyire lehetett.

28. ábra Klaszter állapot adatmodell

Az adatmodellünk valójában Data Transfer Object, mivel a klaszter menedzseléséhez használt Kubernetes Java könyvtár [29] egy jóval bővebb reprezentációval szolgál, a Kubernetes objektumot tárolására. Azonban ennek az információ tartalma a megjelenítés szempontjából nem releváns és emiatt csökkenteni lehet az adatok mennyiségét és a szerkezet bonyolultságát egy DTO használatával.

A másik része a feladatnak a szimulációk és azok szcenárióinak menedzselése. Ehhez egy leíró adatmodell kellett, amivel a megfelelő konfigurációkat és Kubernetes-hez tartozó yaml-fájlok konfigurációit is tárolni tudtam.



29. ábra Szimulációk adatmodellje

Az adatmodellekben a vastagon jelölt és magenta színezett rész, az jelöli a API végponttal elérhető részeket. Ezeket valósítottam meg a kódban teljes CRUD (Create Read Update Delete) funkciókkal. Ez azt jelenti, hogy az API által szolgáltatott Json kiterjesztésű válaszokat is ezek szerkezete reprezentálta. A fejlesztés során Postman segítségével teszteltem ezek működését a korai fázisban.

Az implementáció során három fő részre tagoltam a kódot.

Az első a domain, repository és service package-ekben található kódokból áll. Ezek feladata implementálni az adatmodellt, menedzselni annak adatbázisba való perzisztálását és elérését. A Java Spring JPA és Hibernate programkönyvárakat használva implementáltam. Ezek fölé az adatbázis lekéréseket generáló Repository és Service osztályokat tettem, amivel menedzselhetjük az adatbázisba mentett adatokat.

A második rész a controller osztály. Ennek feladata a Spring RestController megvalósítása, és a Rest végpontok implementálása. Az alap CRUD metódusokat hoztam létre, és a Service osztályok metódusait hívva kötöttem össze a Rest végpontokat az adatbázissal.

A harmadik rész a logikát megvalósító kódokból áll. Ezek a kubernetes package-ben és a clusterstate.engige és generator package-ekben találhatóak. Ezek hajtják végre a szolgáltatás funkcionalitást az API funkciók mellett. Ezeknek a feladata a Kubernetes klaszter menedzselése a mi igényeink szerint.

Ezek mellet még néhány segítő kódrészlet található a helper package-ben, ami a program futását és konfigurálását teszi lehetővé yaml-fájlból.

6.1.2 Funkciók implementálása

6.1.2.1 Klaszter állapot vizualizálás

Amint említettem a két fő feladat a klaszter állapotának lekérdezése és a szimulációk generálása és végrehajtása. Mindkettőt a Java Client használatával lehet megtenni. Ennek a létrehozott klaszterem kubeconfig fájlját kellett megadni, amivel elérheti a klasztert. Ezután a Kubernetes akciók nagyrészét végre lehet hajtani ezzel a klienssel. Számomra hasznos volt a Kubernetesped futó node-ok valamint Pod-ok lekérdezése, mivel ezeket szerettem volna megjeleníteni.

A Kuberenetes klaszter funkciókat egy KubernetesManagementService osztályon keresztül tettem elérhetővé. Ez lekérdezi a klaszterben a node-ok állapotát és a Pod-ok állapotát. Ez még a saját modellrendszerében található objektumokkal kerül vissza többi osztályhoz.

A klaszter állapotát a RefreshData metódus frissíti a KubernetesManagementService metódusain keresztül, ezeket pedig perzisztálja a modellünkbe. Ennek az időszakos meghívásáért felel a RefreshScheduler osztály, ami állítható időközönként meghívja újra a frissítés metódusát. Ezáltal a Kubernetes állapota frissítve kerül elmentésre a saját adatbázisunkba.

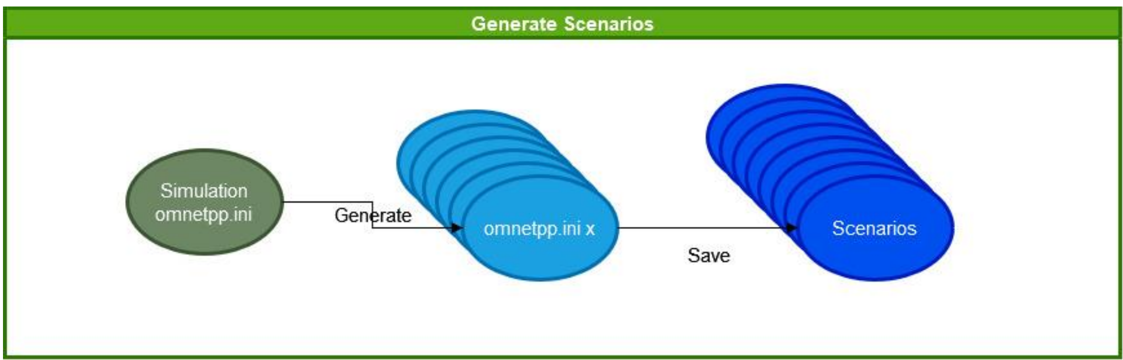
6.1.2.2 Szimulációk kezelése

A szimulációkat a klaszter állapothoz hasonló módon tároltam. A feladat, hogy a felhasználó az API-on keresztül létrehozzon egy szimulációt reprezentáló entitást. Ezt úgy teheti meg, hogy a szimulációhoz használt konfigurációkat is csatolni tudja a szimulációjához. Kiválaszthatja a szimulációs konténert, amit használni szeretne. Majd ezeket mi a Kubernetes-ben futtatható formában generáljuk és tároljuk. Végül pedig végrehajtjuk számára a Kubernetes klaszterben.

Első feladat az alapvető CRUD végpontok kialakításán túl, tehát a fájlok feltöltését megvalósítani. Ezeket külön végpontokkal tettem lehetővé a SimulationController osztályában a szimulációknak. Ezáltal két olyan végpont jött létre ami fájlokat képes fogadni, és azokat a lokális fájlrendszerre elmenti, ezekre pedig referenciákat tárolunk az adatmodellben, az adatbázisban. Ezek megvalósítása a Spring és Java alapvető metódusait igényelte.

Amennyiben kész a szimulációs entitásunk, tehát minden adatot és fájlt létrehoztunk és perzisztáltunk a backend alkalmazás segítségével, akkor kezdhetjük a Kubernetes entitásokra fordítását a szimulációnak. Ennek megoldása a generator package-ben található. Két lépcsőben történi a generálás.

Az első lépcsőben a feltöltött omnetpp.ini fájlt, ami leírja a szimuláció fő paramétereit dolgozzuk fel. Alapesetben az Artery szimulátorban az INET egyik osztálya teszi meg ezt a feldolgozást és hajtja végre sorban a szimulációkat, itt ezt kézzel tesszük. Az omnetpp.ini-ben szereplő változókat (seed,traffic,penetration) dolgozzuk fel úgy, hogy azok kombinációit generáljuk és külön külön omnetpp.ini reprezentációként tároljuk. A tartalmukat nem írjuk tárhelyre, az adatmodellben perzisztáljuk egy szcenárió entitásban. A metódus végén a Simulation objektum már tartalmaz Scenario objektum listát, amik saját számozott névvel jönnek létre és a szimuláció szcenárióit egyenként tartalmazzák saját konfigurációként.



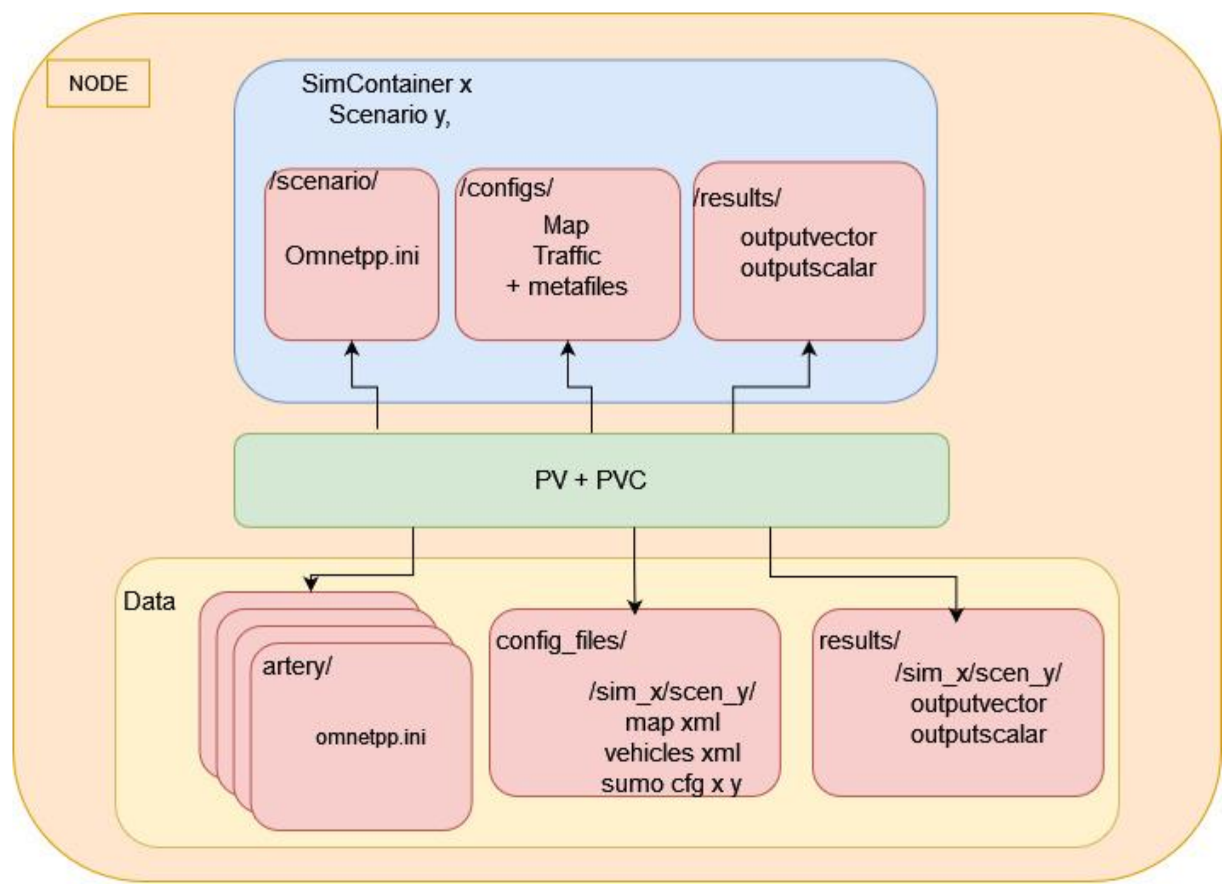
30. ábra Szcenárió generálás

A második lépésben ezekhez a szcenáriókhoz generálunk Kubernetes Job-okat és azokhoz ConfigMap-okat. Ezeket már a szcenáriók omnetpp.ini verziója alapján tesszük.

A ConfigMap és a Job yaml fájloknak is van egy-egy template verziója. Ezeket egészítem ki miután betöltöttem a Kubernetes Client modelljébe. Ez azt eredményezi, hogy kódmódosítás nélkül is tudunk majd Kubernetes infrastruktúra függő elemeket módosítani. A mi esetünkben például a névtereket könnyen lehet változtatni ezek módosításával anélkül, hogy ezt implementálni kéne.

A ConfigMap fogja tárolni a megfelelő omnetpp.ini fájlt a szimulációhoz. Ez egy kulcs érték alapú konfigurálási módszer ahol a fájlnév kulccsal és a fájltartalom érték párral tudunk injektálni a Pod konténerének egy fájljába információt.

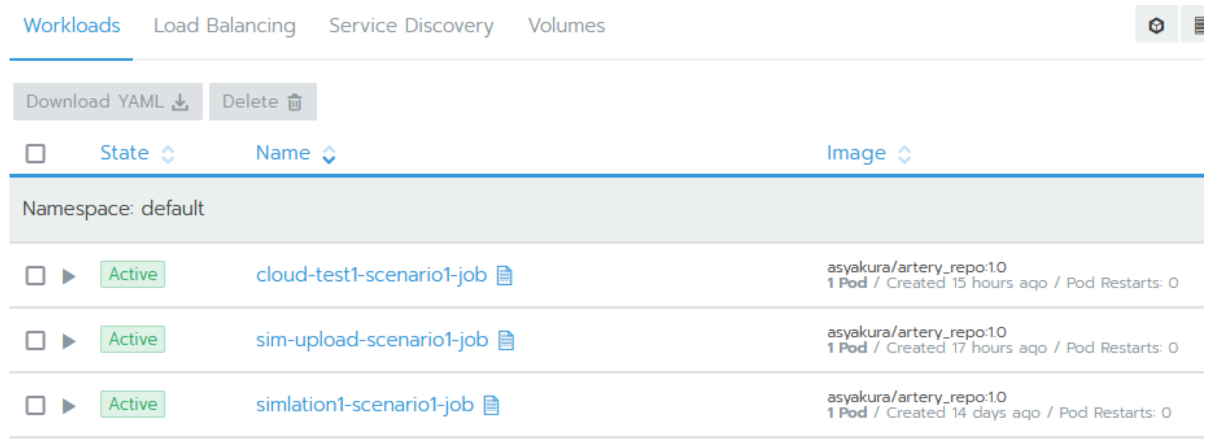
A Job yaml az a fájl, ami leírja a szcenárió futtatását. Ebben hivatkozzuk a megfelelő szimulátor konténer image-t és a többi beállítást. A Job-nak mindig változik a neve, és így a Pod-ok neve amire a MatchLabel egyeztetést megteszi amikor azonosítja a Pod-jait. Emellett fontosak a PerzistentVolumeClaim-ek és VolumeMount-ok specifikációi. A Job-nak ez adja a megfelelő elosztott tárhelyet, amire szüksége van hiszen a konfigurációs fájlokat el kell érje és az eredményeit a megfelelő tárhelyre menteni kell tudnia. Ezen kívül a ConfigMap-ban tárolt adatokat is csatolni kell.



31. ábra Elosztott tárhely a szimulációknak

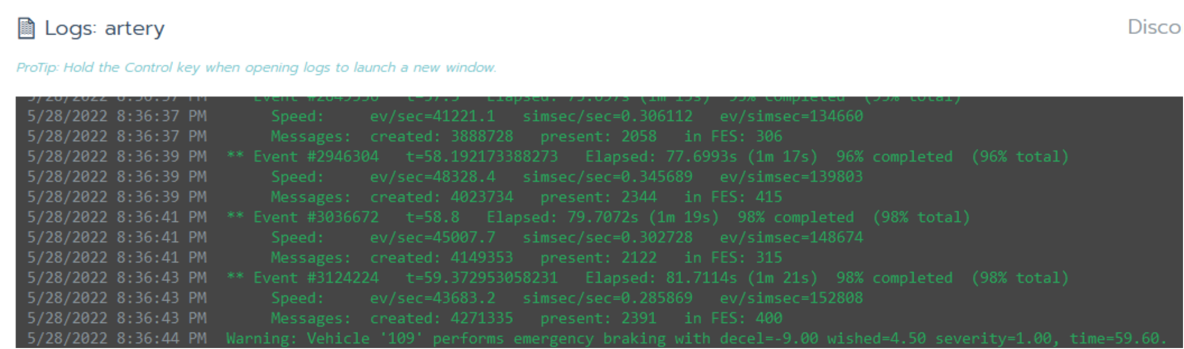
A backend a konténerizálása során, ugyanúgy megkapja a konfigurációs fájlok helyét és oda helyezi az adatokat, amiket a webfelületről a backend API segítségével feltöltünk. Így a szimulációs szcenárió konténerei elérik a konfigurációs fájlokat. A ConfigMap-ok segítségével így mindössze az omnetpp.ini verzióinak kell külön Kubernetes entitást létrehozni a konfigurációs fájlok egy példányban lesznek szerepeltetve. Ezt az omnetpp.ini manipulálásával érjük el, mivel abban a szcenárió és szimuláció specifikus útvonalakra cseréljük az eredetieket. A scheduler számára pedig megadjuk a minimális elvárt processzoridő és memória mennyiségeket az ütemezéshez. Ezt a konfigurációs fájlok vizsgálatával tesszük.

A yaml-fájlok legenerálása után, ezeket végrehajtjuk a Kubernetes Client segítségével. Ekkor megjelennek a szcenáriókhoz tartozó Kubernets Job-ok, amiket nyomon követhetünk a Rancher felületen.



32. ábra Szcenárió Job futás

Ezeknek a Log fájljait is vizsgálhatjuk futás közben és akkor az eredeti szimulátorunk futását ismerhetjük fel.



33. ábra Szcenárió logjai

6.2 Eredmény feldolgozó backend szolgáltatás

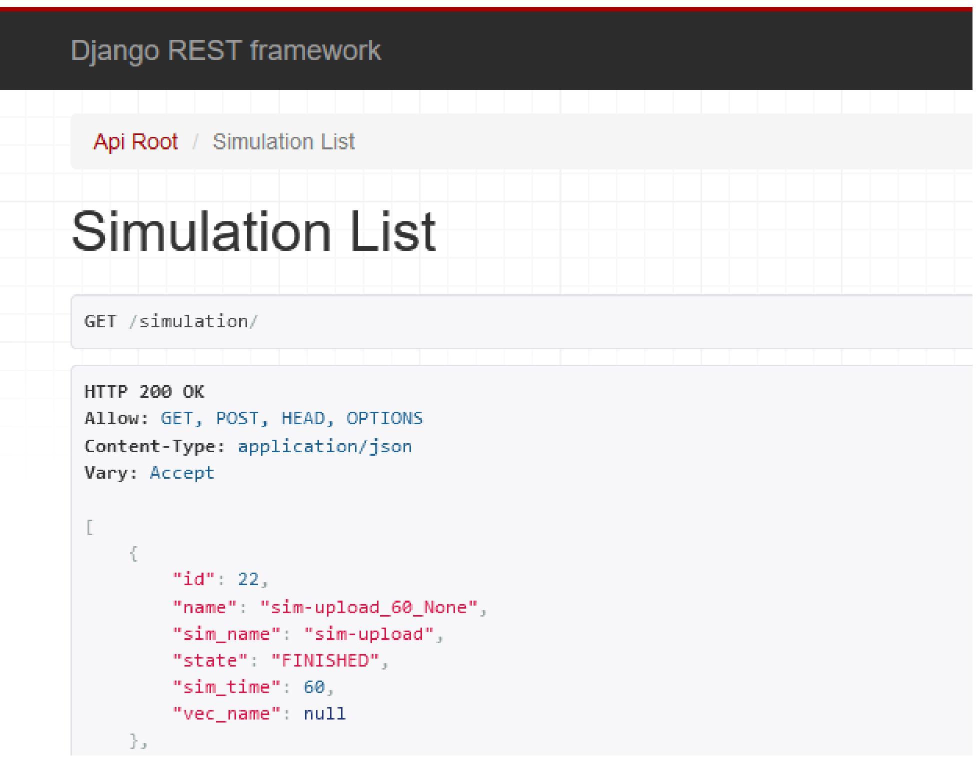
A szimulációk elkészítése és végrehajtása után, az eredmények feldolgozása és vizualizációja a feladat. Ezeket egy másik, Python Django backend szolgáltatás implementálásával végeztem el.

A Python nyelv váltás azért indokolt, mert a Django által nyújtott eszközöknek hála az API technológiai megalkotása nagyon egyszerű feladat, a Java Spring után. A konkrét implementációs feladat így az eredmények feldolgozására irányul, ezeket pedig Python

Scientific könyvtárak segítségével tudjuk hatékonyan és könnyen megtenni. Ezeket a kódokat jóval nehezebb lett volna Inter Op-technológiákkal megvalósítani mint a Python API-t. Plusz a teljes rendszert erősítheti a technológiai sokszínűség.

6.2.1 Modell és API

A Modell itt egyszerűre lett tervezve, hogy csak a megfelelő funkciókat legyen szükséges implementálni. Lényegében a szimulációs objektumot valósítottam meg itt is, viszont itt az eredmények értékeléséhez szükséges mezőkkel.



A Django által nyújtott Rest framework segítségével látható, hogy az adatmodell egyetlen entitásból áll, a megfelelő mezőkkel, ami a szimuláció neve, az eredmény objektum származtatott neve és az eredmény értékeléséhez a szimulációs idő és a vizsgált vektor neve.

A teljes CRUD funkcionalitást egy Python Rest framework ViewSet és Model megvalósításával értem el. Ez egy default Router létrehozásával módosítások nélkül szolgáltat teljes CRUD alap metódus palettát. Ezt ugyan később a Create metódus felülírásával bővítettem mivel ott végezzük a funkció végrehajtásának inicializálását is. Az osztály és fájlstruktúra ismertetését mellőzöm, mivel az teljes mértékben a Django keretrendszer generálta.

6.2.2 Szimuláció eredmény feldolgozás

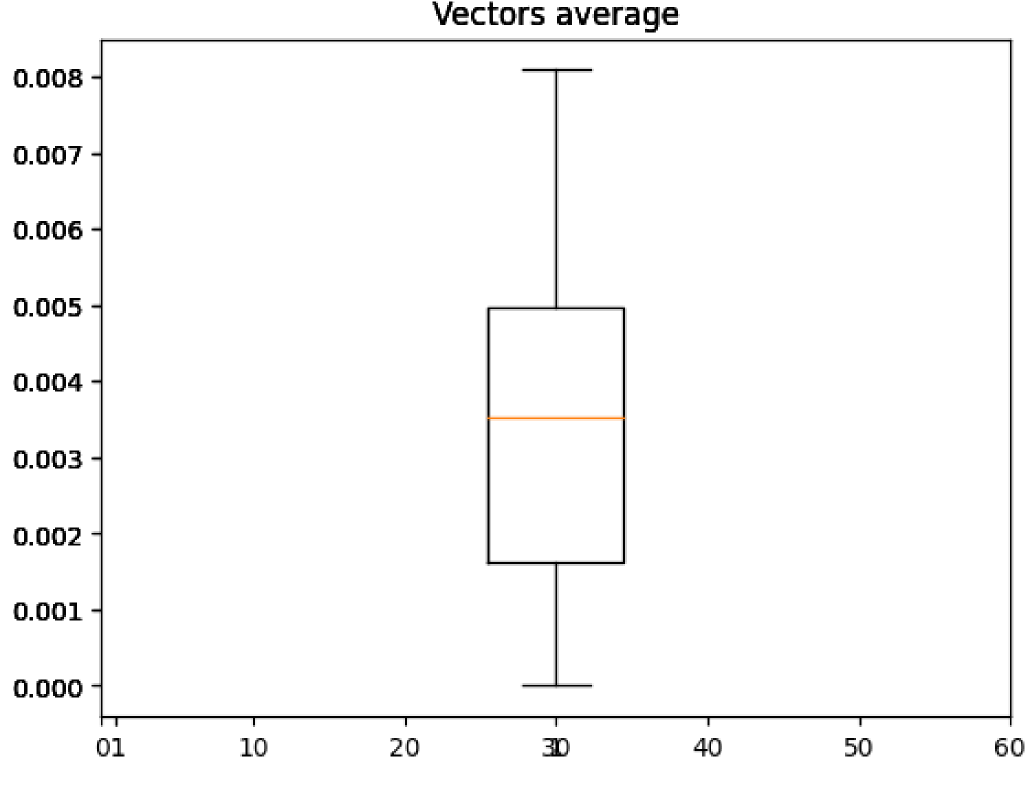
Az eredmények feldolgozásához a számomra már ismert Python scripteket kellett a Jupyter Notebook környezetből implementálom a backend szolgáltatásom metódusaiba. Ezeket tehát a használt Python könyvtárak telepítése után, konfigurálható formába hoztam. Ezt a result_maker.py fájlban tettem meg.

Az első feladat a Scavetool Omnetpp program futtatása, ami feldolgozható csv fájlokat csinál a megfelelő vec kiterjesztésű fájlokból. Ezután feldolgoztuk a csv-fájlt.

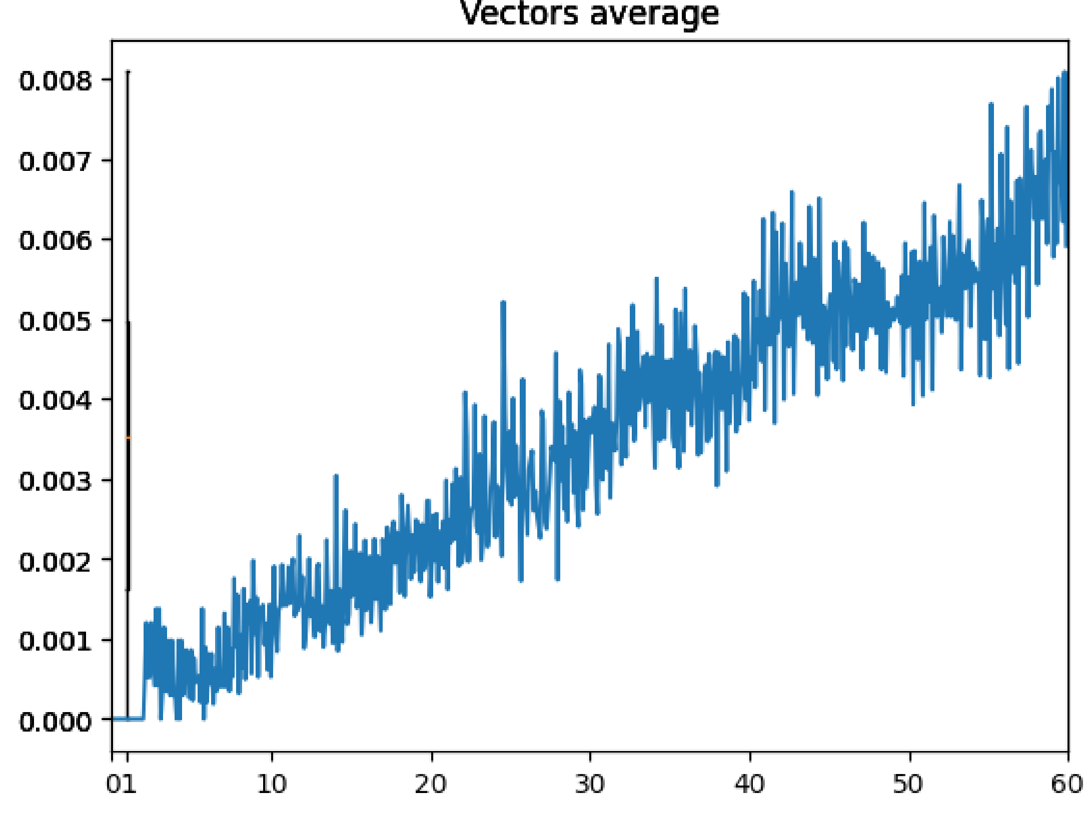
A legfontosabb metódus, ami az adott vektort a teljes szimulációra tekintve átlagolja, tehát minden szcenárió minden vektorát időbélyeghez kötött módon átlagolja. Ezt egy script segítségével tettem meg, ami listákat használva járta be az egész szimulációt. Ennek használata nagyobb szimulációk során, több szcenárióra viszont rengeteg időt vett igénybe, ezért át kellett alakítom Python dictionary-ket használó verzióra, amivel a performanciát optimalizálhattam.

Ez a függvény egy szcenárión futott, majd az össze szcenárión futtatva, létrehozta a teljes szimuláció átlagos vektorait. Ezt használtam az eredmények vizualizálásához, hogy egy teljes képet adjak a szimuláció kimeneteléről.

Ezekből egy box plot, egy vonal ábra és egy eredményeket leíró json, készült el.



34. ábra Szimuláció aggregált box plot



35. ábra Szimuláció aggregált vonalas ábra

A json fájl tartalma pedig az átlagos, közép és leggyakoribb érték-et tartalmazza.

36. ábra Eredmény összegző adatok

Az eredmények vizualizálását a végletekig lehet fejleszteni, de a projekt összetettsége miatt ez a verzió megfelelő adatokkal szolgál több szimuláció alapvető összehasonlításához.

6.3 Konfigurációs és eredmény megjelenítő felület frontend alkalmazás

A webes kliens alkalmazás egy Single Page Application – SPA. Megvalósításához egy manapság igencsak felkapott JavaScript keretrendszert, a React-ot választottam eszközül. Ennek a JavaScript keretrendszernek egyik alapvető előnye az, hogy virtuális DOM fát – azaz virtuális Document Object Model fát – tart számon, aminek a segítségével képes a böngészőbeli újra rendereléseket (képernyőfrissítések és ezzel járó összes funkció újra meghívását) optimalizálni, ezzel teljesítmény szempontjából nő a hatékonysága.

A React 17-es fő verzióját használtam, ami már tartalmazza a függvény komponenseket (Functional Components), illetve további könyvtárakat is igénybe vettem mind a kód tömörsége érdekében, mind a vizuális megjelenítés javítása céljából – illetve online kurzusokat, forrásokat követve amilyen technológiai stacket megismertem és hasznosnak találtam.

6.3.1 Alkalmazott könyvtárak

6.3.1.1 Styled-components

A React styled components könyvtár lehetővé teszi a CSS JavaScript kódban való megfogalmazását, olyan már stilizált komponensek – komponens alatt értsük a klasszikus HTML tag-eket (div, p, button) – létrehozását, melyeket akár más fájlból importálva, vagy saját JSX fájlunkban deklarálva, már HTML tag-ként használhatunk, a React képes ezeknek a feldolgozására és megjelenítésére. Ezzel akár újra felhasználható komponensek, komponens hierarchia is megvalósítható –projektem méretéből adódóan azonban csak kevés előnyét volt lehetőségem kihasználni.

6.3.1.2 React-Redux

A Redux, azaz pontosabban React-Redux könyvtár használatával érhető el a fejlesztett SPA állapotának egy, központi helyen való tárolása illetve egyes elemek állapotának oly módon való tárolása, hogy az elérhető legyen az alkalmazásban felvett bármely komponensből. Alapvető elemei a reduxnak a Store, mely a központi adat tároló egység, melyben definiálhatóak Slice-ok, melyek már csak adott szegmensét tárolják az adatoknak – az alkalmazásomban erre példa a klaszter állapotát tároló, annak frissítésére képes, API hívásokat is kezelő ClusterStateSlice. A Slice-okban definiálhatóak különböző akciók (Reducer-ek),

melyek alkotják a Slice-unk interfészét az alkalmazás felé, alapvetően ezeken keresztül lehet módosítani az adott Slice állapotát. Az adatok lekéréséhez nem akciókat, hanem ún. Selector-okat lehet definiálni, melyek biztosítják a Slice-ban (illetve Store-ban) tárolt adat és a megjelenített adat közötti adatkötést. API hívások kezeléséhez szükséges volt azonban egy a React-Redux könyvtárat kiegészítő könyvtárra, a ReduxJS/Toolkit-re. Ez a könyvtár biztosítja a Slice-ban definiált Reducerek alapvetően szinkron viselkedéséhez azt a képességet, hogy aszinkron eseményekre is reagálhassunk, Async Thunk-ok formájában. Technológiai hátterét nagyjából ezen a szinten megértve, gyakorlati példák követésével képes voltam az SPA állapotának ilyen módon való menedzselésére.

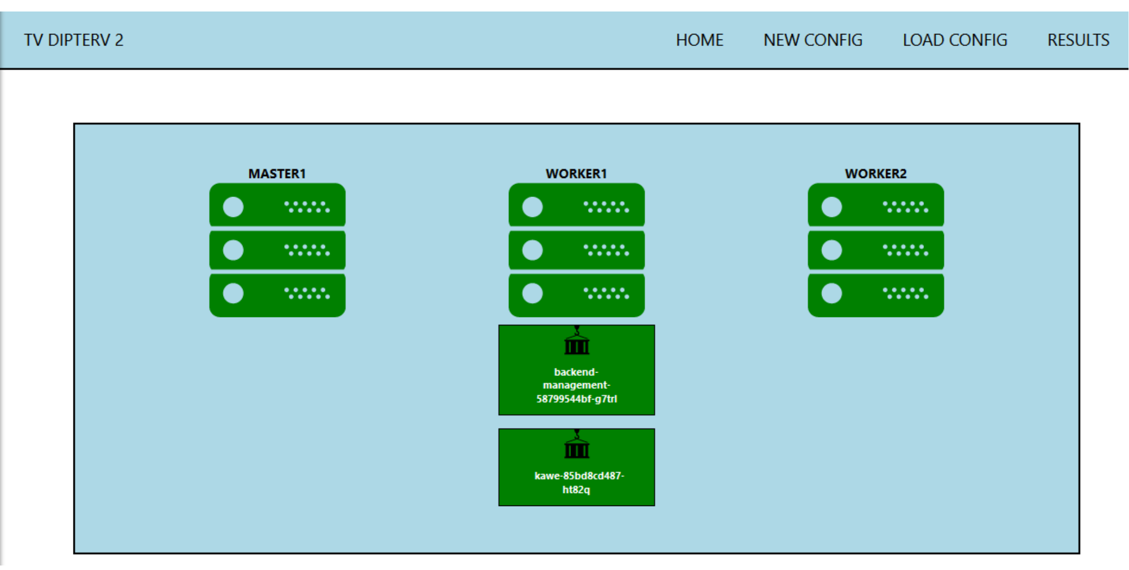
6.3.1.3 React-Toastify

A mindenki számára ismert Toast-ok a kis méretű, felugró értesítések, melyek nagyon sok alkalmazásban megtalálhatók, a felhasználót bármiféle eseményről – hibáról – értesíteni hivatott dobozkák. A React-Toastify alkalmazásával az alkalmazás „gyökér" JavaScript fájljában, az App.js-ben volt szükséges definiálni egy ToastContainer-t, illetve a komponensek közé felvettem az általam használni kívánt értesítések stílusát meghatározó objektumokat, külön függvényeket írtam az információt, sikert, hibát, figyelmeztetést jelző értesítések megjelenítésére, majd ezek után már az alkalmazás bármelyik komponenséből – vagy akár a korábban leírt React-Redux Reducerekből – meg lehetett jeleníteni a kívánt értesítést. Erre példát a klaszter állapotát mutató oldalon láthatunk, amikor is bármilyen hálózati hiba esetén piros színű dobozokban érkezik a hiba értesítés.

6.3.1.4 Kész applikáció

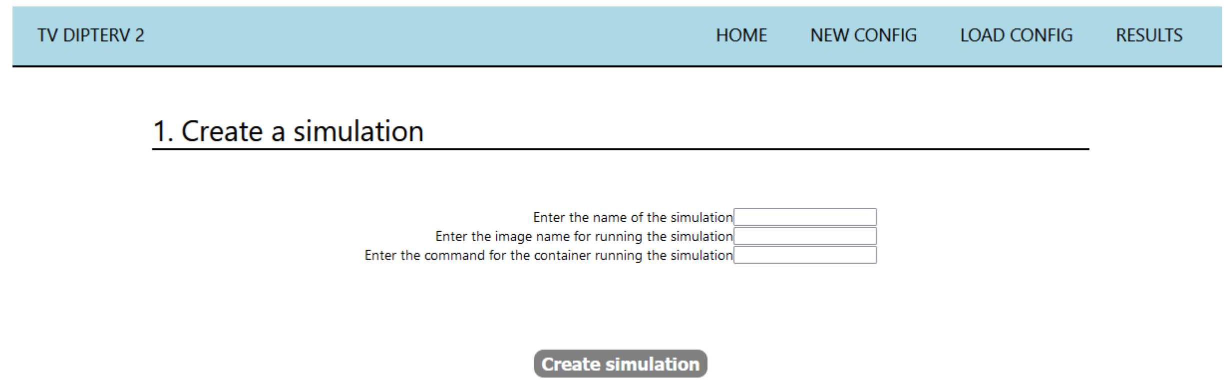
A kész applikáció segít a backend funkciók használatában. Ennek megfelelően több oldal létezik.

A Home oldalon láthatjuk a futó konténereket Node-hoz kötve, a ClusterState hívás feldolgozásával, ez a futó konténerek állapotát írja le nekünk, és láthatjuk a különböző elosztású node-okat.



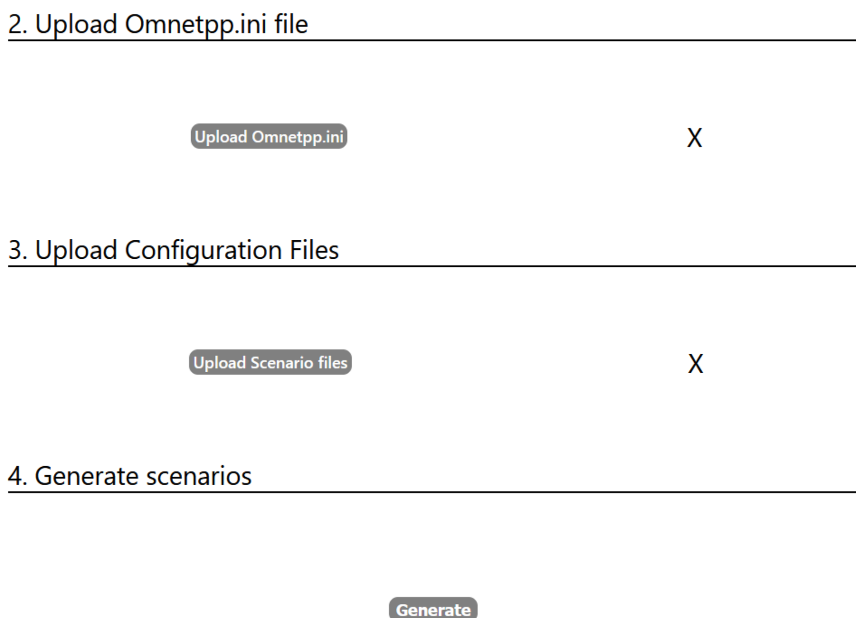
37. ábra Klaszter állapot frontend oldal

Emellett található az új szimuláció konfigurálása oldal:



38. ábra Új szimuláció létrehozása frontend oldal

A megfelelő fájlfeltöltésért felelős gombokat ezután használhatjuk lentebb, majd ha megtörtént az omnetpp.ini feltöltése akkor használhatjuk a generálás gombot:



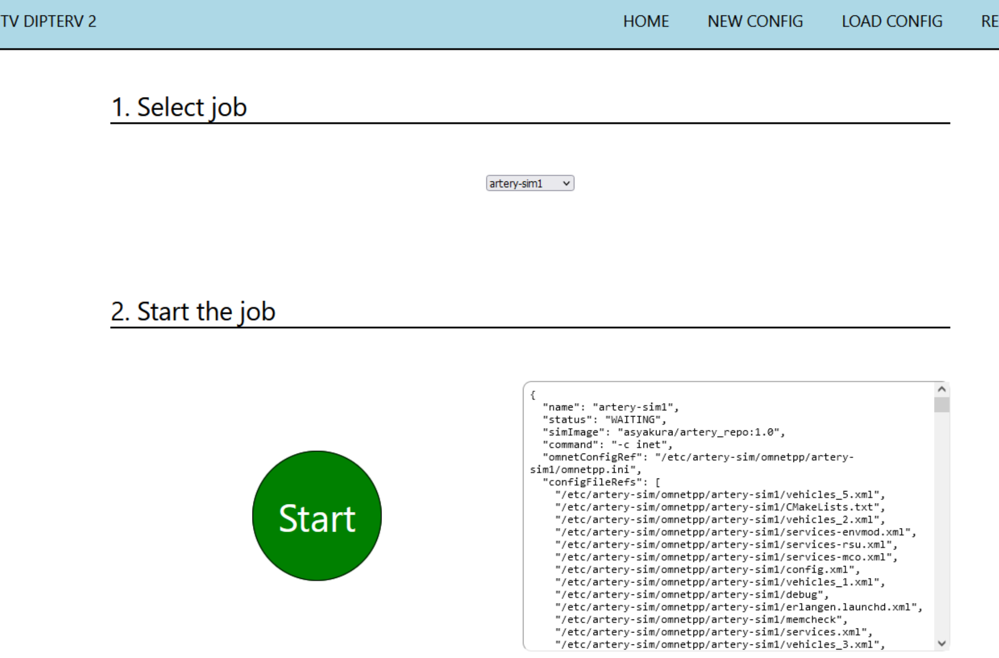
39. ábra Fájl upload és szcenárió generálás

Az oldal alján egy összegző felület található, ami alap json formátumban ad nekünk visszacsatolást a feltöltött konfigurációkról és a szimuláció adatairól, és persze egy nagy zöld gomb az indításhoz:

6. Start the job

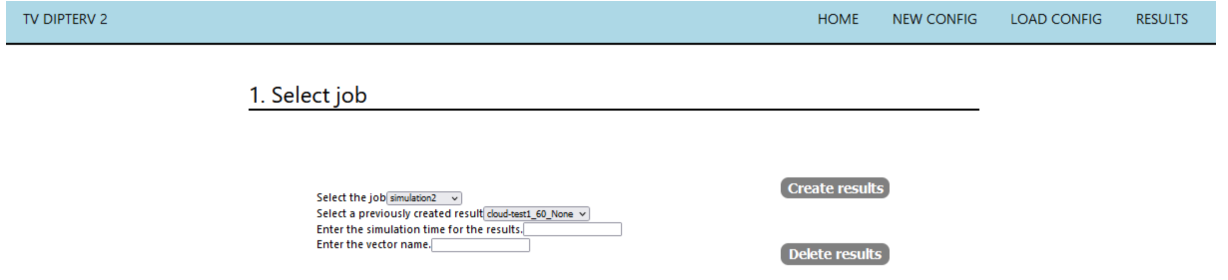
40. ábra Szimuláció összegzés

Ezután találunk egy konfiguráció újra töltő oldalt, amivel a már felvett szimulációkat futtathatjuk újra:



41. ábra Konfiguráció újra töltésé

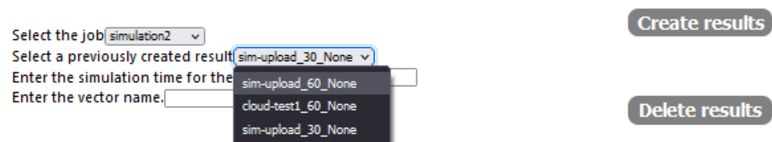
Végül pedig az eredmények vizualizálásáért felelős oldal a Results:



42. ábra Eredmény kiválasztása

Amennyiben még nincs az eredmények között a mi szimulációnk, úgy a szimuláció nevét kiválasztva megadunk egy vektor nevet és szimulációs időt, ezáltal készíthetünk egy új eredményt a lefutott szimulációnkból. Akár ugyanazon szimulációból tudunk készíteni más vektor, vagy más szimulációs időre vetített kimutatást.

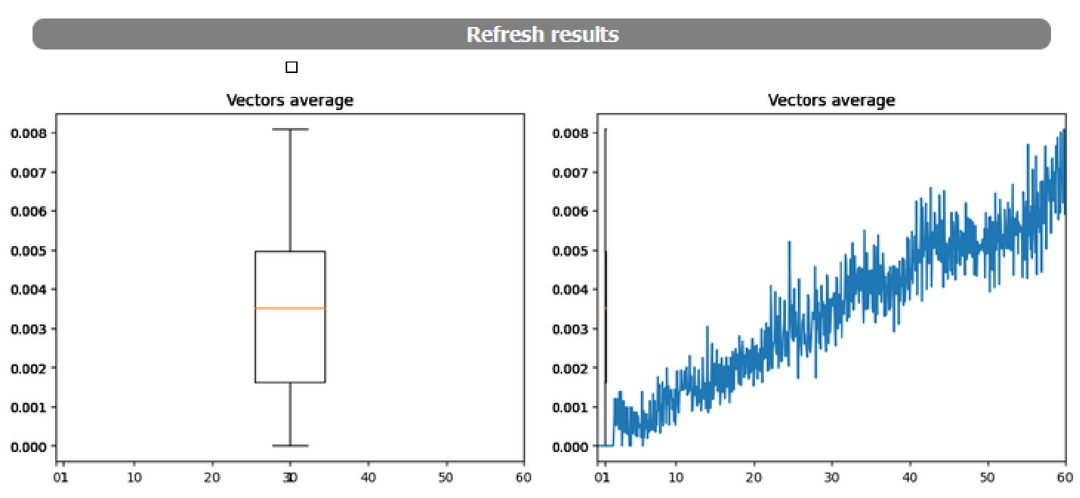
1. Select job



43. ábra Azonos szimulációk eltérő idővel való eredménye

Miután az eredménykészítés a háttérben lefut, a Refresh results gomb segítségével betölthetjük az eredményeket.

2. Results



Stats of the simulation:

median	0.003520635624538062
mean	0.0033620215877563997
mode	0

44. ábra Választott eredmények vizuálisan

6.4 Alkalmazások konténerizálása és Kubernetes integrációja

Az összes felsorolt alkalmazás a Kubernetes rendszerünkben fut. Ennek eléréséhez először mindegyiket konténerizálni kell. A konténerizálás mindegyik programnál hasonló, első lépés a megfelelő futtató környezet, alap konténer image és programok telepítése. Majd a program kód eljuttatása a konténerbe és annak installációja. Ezután pedig a futtatása a megfelelő paraméterekkel.

6.4.1 Java backend konténerizálás

A Java backendhez szükséges volt a paramétereket egy yaml konfigurációs fájlba szervezni mivel a program Kubernetes-be helyezésével a paraméterek változhatnak. Tipikusan az adatbázisra vonatkozó adatok és hasonló paraméterek különböznek ilyenkor. Ezeket később ConfigMap-ként lehet felülírni.

```
#Basic image
FROM openjdk:11.0.14-jre

#Update for apt
RUN apt update

#The current directory will be this one
WORKDIR /usr/src/management-backend

#Copy the jar files into the container
COPY ./target/*.jar /usr/src/management-backend/service.jar
COPY src/main/resources/application.yaml /usr/src/management-backend/config/application.yaml

#Start the application jar file with config location and config yaml file name
CMD java -jar /usr/src/management-backend/service.jar --spring.config.location=file:/usr/src/management-backend/config/
```

45. ábra Java Backend Dockerfile

A konténerizáláshoz egy Dockerfile-ra volt szükség, ami megadja a kép elkészítésének paramétereit, majd ezt a Java Spring Spotify Docker plugin projektbe kapcsolásával használtam a Maven install fázisban.

6.4.2 Python Django backend konténerizálás

A Python Django backend konténerizálása alapvetően szintén a konfigurációk kiszervezésével oldható meg. Itt is egy Dockerfile-ban írtam le az Image gyártás folyamatát.

```
ENV PYTHONUNBUFFERED 1

RUN mkdir /figures
RUN mkdir /csv
RUN mkdir /python_backend

WORKDIR /python_backend

COPY . .

COPY ./requirements.txt /requirements.txt

RUN pip install --upgrade pip

RUN pip3 install matplotlib
RUN pip3 install numpy
RUN pip3 install pandas

RUN pip install -r /requirements.txt

CMD python manage.py migrate && python manage.py runserver 0.0.0.0:8080
```

46. ábra Python Django backend Dockerfile

A nehézségeket itt a Python Scientific könytvárak telepítése adta, mivel ezeket a requirements.txt kigenerálásakor adott verziókkal szerette volna a konténer telepíteni. Ezt azonban felül kellett írnom mivel a nem megfelelő verziók előre build-elt wheel-ek nélkül települtek majd órákat telepítették azokat.

Ezek mellett egy Omnetpp verziót is telepítettem a konténerbe a Scavetool script használatához.

6.4.3 JavaScript SPA konténerizáció

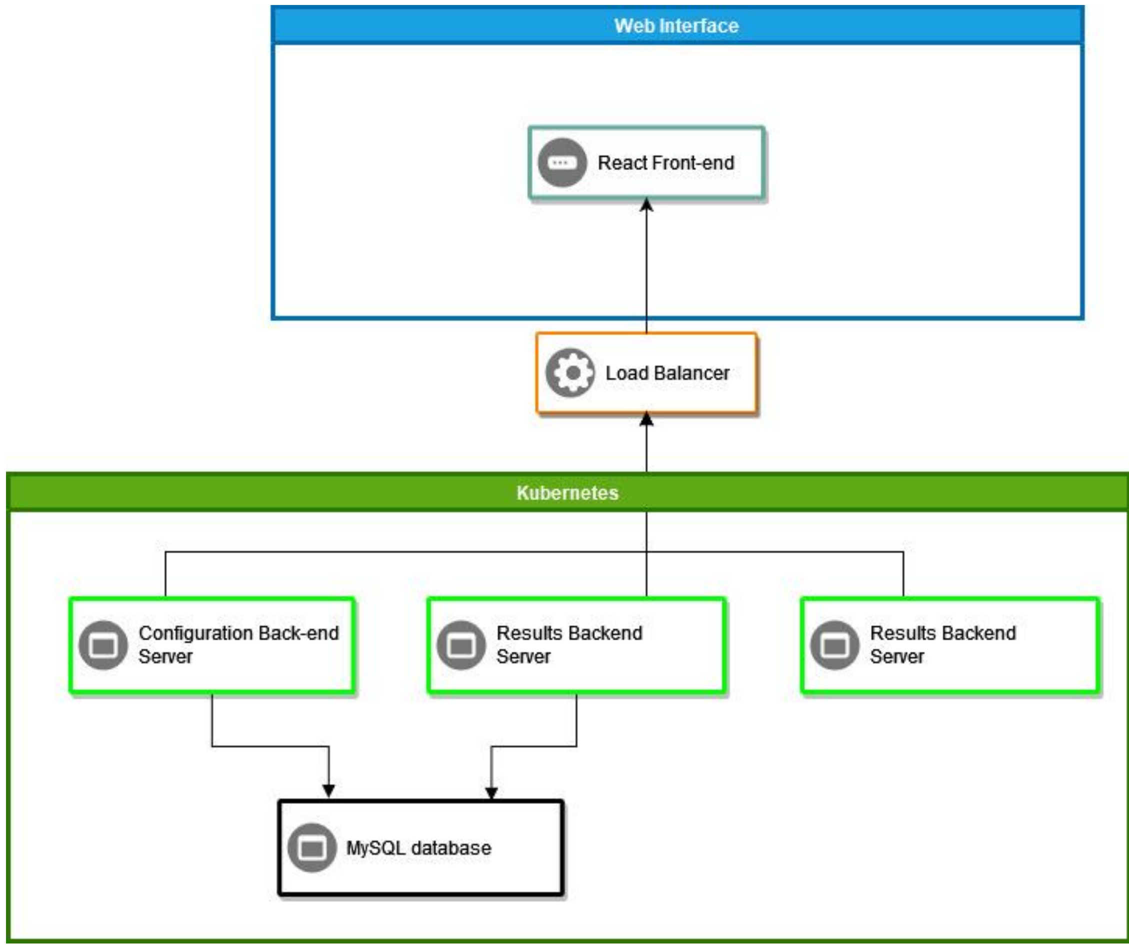
A JavasScript konténerizálásánál egy félkész megoldást használtam, ami nem production ready konténert ad nekünk, mivel tartalmazni fogja a konténer a program kódot. Ennek a megoldásnak viszont egyszerűbb a konfigurálása és debug-olása. Ezért ezt választottam. A konfigurációt itt is egy fájlba szerveztem, ezen kívül a másik fontos dolog a react script start átírása volt, mivel az SSL verzióváltás miatt nem tudott futni a konténer környezetben.

```
FROM node:current-alpine3.15
RUN mkdir /usr/app
COPY ../ /usr/app
WORKDIR /usr/app
RUN npm ci
CMD npm run start
```

47. ábra SPA Frontend Dockerfile

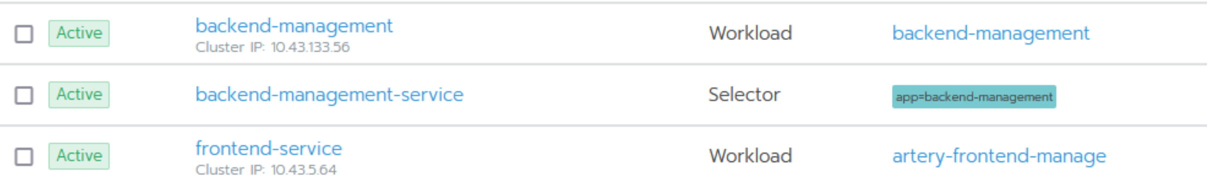
6.4.4 Kubernetes integráció

A konténereket ezután Kubernetes Deployment-ként kellett telepíteni és elérhetővé tenni. Mivel a felhőbe kerültek az alkalmazások, ezért egy saját MySQL adatbázist is telepítettem, ami kiszolgálta őket.



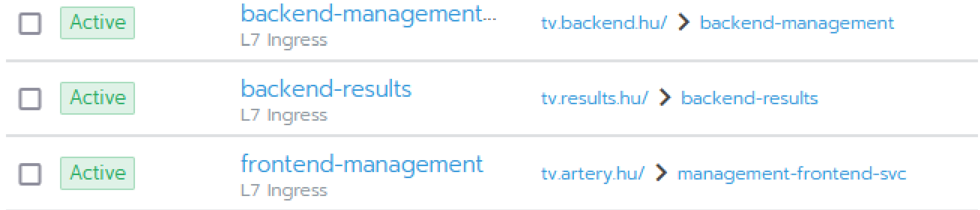
48. ábra Kubernetes telepítés architektúra

Ezután minden szolgáltatás kapott egy Service-t, ami klaszteren belül teszi őket elérhetővé.



49. ábra Kubernetes Service-k

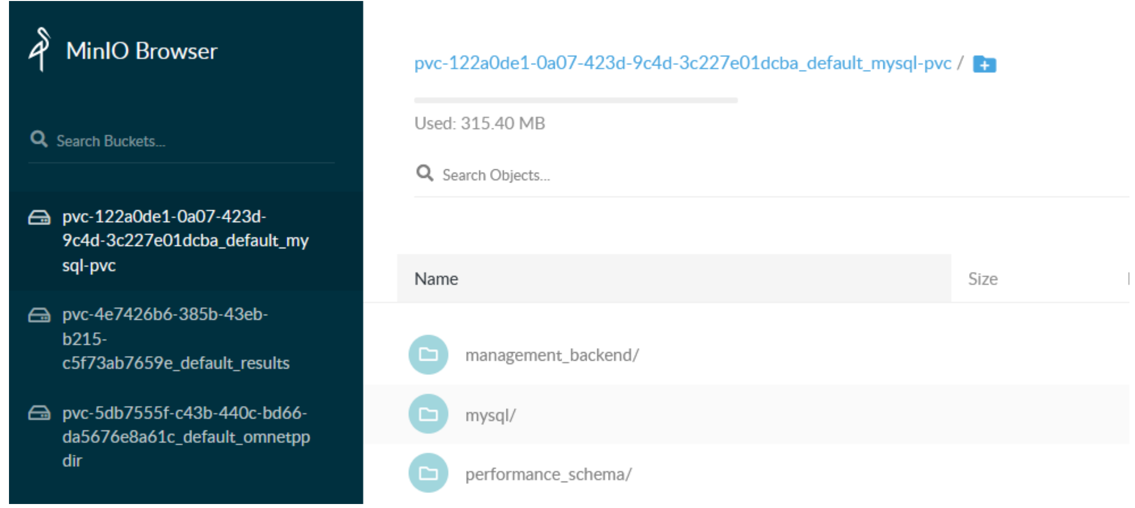
És ezeket a Service-eket felhasználva az Ingress segítségével publikusan is elérhetővé lehet tenni adott domain neveken.



50. ábra Ingress beállítások

Ezután ezeket fel kell venni a tesztelő rendszerén mint feloldható domain nevek, esetünkben egy Windows típusú rendszeren például a hosts fájlban érdemes.

A rendszerhez telepítettem még egy MinIO felületet, amivel a háttértáron elérhető fájlokat tudjuk könnyedén menedzselni. Mivel erre a mi szolgáltatásaink nem adnak megfelelő megoldást.



51. ábra MinIO felület

A fejlesztés segít pedig egy phpMyAdmin felületet is telepítettem. Amiben a két backend állapotát lehet megfigyelni és menedzselni.

A szolgáltatások publikálása után, a megfelelő konfigurációkat a ConfigMap-ok segítségével végeztem.

☐	Active	backend-management-cm	default	application.yaml
☐	Active	backend-management-configmapyaml	default	configmap.yaml
☐	Active	backend-management-kubeconfig	default	kubeconfig
☐	Active	backend-management-omnetpp	default	omnetpp.ini
☐	Active	backend-management-testjobyaml	default	testjob.yaml
☐	Active	django-directory-config	default	config.json
☐	Active	django-settings-py	default	settings.py
☐	Active	frontend-configmap	default	backend_config.json

52. ábra Konfigurációs fájlok ConfigMap-jai

Itt, mint látszik a Java backend, a Django backend, és a frontend beállításainak a fájljait tartalmazzák.

6.4.5 Kubernetes integráció kihívásai

Az alapvető környezet váltás miatt sok problémával kell megküzdeni. Az egyik ami webfejlesztésnél mint amilyen az én projektem előfordul, az a Cross Origin Resource Sharing megoldása. Ez ugyanis változik a környezettel mivel más forrás domain nevek fognak hívásokat intézni. És nem elég az alkalmazáson engedélyezni, mivel a Kubernetes Ingress objektumon szintén be kell állítani ezeket. Továbbá nem szabad elfelejteni, hogy bár a frontend alkalmazásunk a felhőben fut, a valódi felhasználó által is kezelt alkalmazás a felhasználó eszközén fog futni, ezért azt külső entitásként kell kezelni Cors, és Routing esetében.

7 5G-V2X szimuláció tervezése és megvalósítása a rendszer teszteléséhez

A szimuláció megvalósítása a példa szcenárió létrehozásával történik. A választott szimulációs kódom egy Artery, Simu5G példa. Aminek a konténerét le kell gyártani és a megfelelő szcenáriókkal elindítani.

A szcenárióm Győr városának térkép exportálása az OpenStreetMap felületéről. A térképfájlt használva több szintű forgalmat generáltam és ezeket külön SUMO útvonalfájlokban tároltam, a névkonvenciót tartva a több konfigurációhoz. Ezután elkészítettem az egyes SUMO konfigurációs fájlokat, ugyanis az omnetpp.ini fájlban ezeket iterálva tudunk más konfigurációkat futtatni.

53. ábra Sumo térkép Grafikus felületen

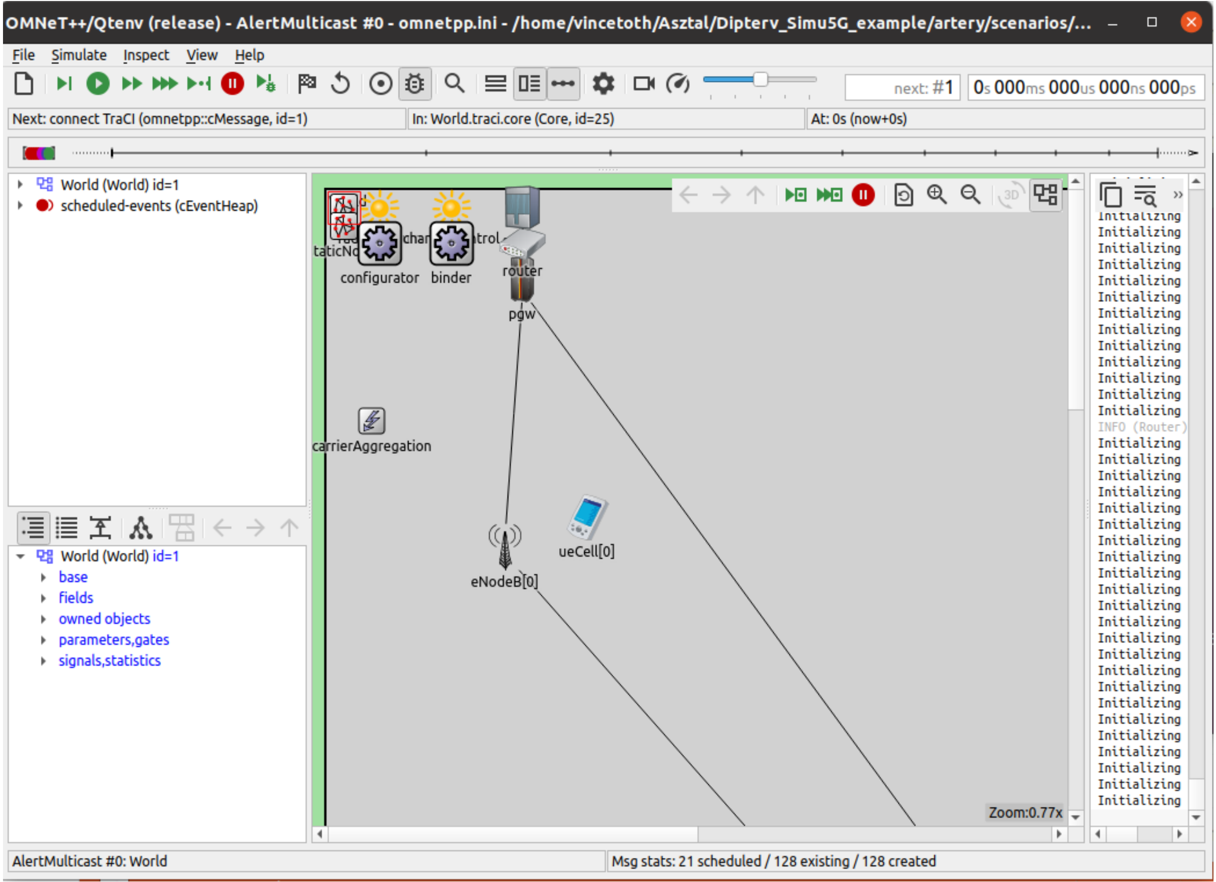
Ezután az omnetpp.ini fájlban megadva a konfigurációs fájlok sorozatát kész volt a szcenárióm. Az omnetpp.ini fájlban ezután penetrációs fájlokat iterálva és más konfigurációkat, esetünkben még fontos seed értéket beállítva tudunk több szcenáriót generálni a megadott térékép és útvonal fájlokkal.

A Simu5G telepítését a tanszéki Gitlab felületéről az Artery develop branch-ről végeztem. A megfelelő Omnet++ verzió telepítésével, a SUMO-t pedig utána. A kész telepítés után a teszteket lefuttattam, abból a Simu5G tesztek sikeresek voltak de néhány másik hibás.

```
vincetoth@toth-v-ubuntu20:~/Asztal/Dipterv_Simu5G_example/artery$ cmake --build build --target test
Running tests...
Test project /home/vincetoth/Asztal/Dipterv_Simu5G_example/artery/build
      Start  1: simu5g-example-cellular
 1/21 Test  #1: simu5g-example-cellular ..........   Passed    6.83 sec
      Start  2: simu5g-example-d2d
 2/21 Test  #2: simu5g-example-d2d ..............   Passed    7.39 sec
      Start  3: car2car-grid-cam_bsp
 3/21 Test  #3: car2car-grid-cam_bsp ............   Passed    6.75 sec
      Start  4: example-inet
```

54. ábra Simu5G tesztek

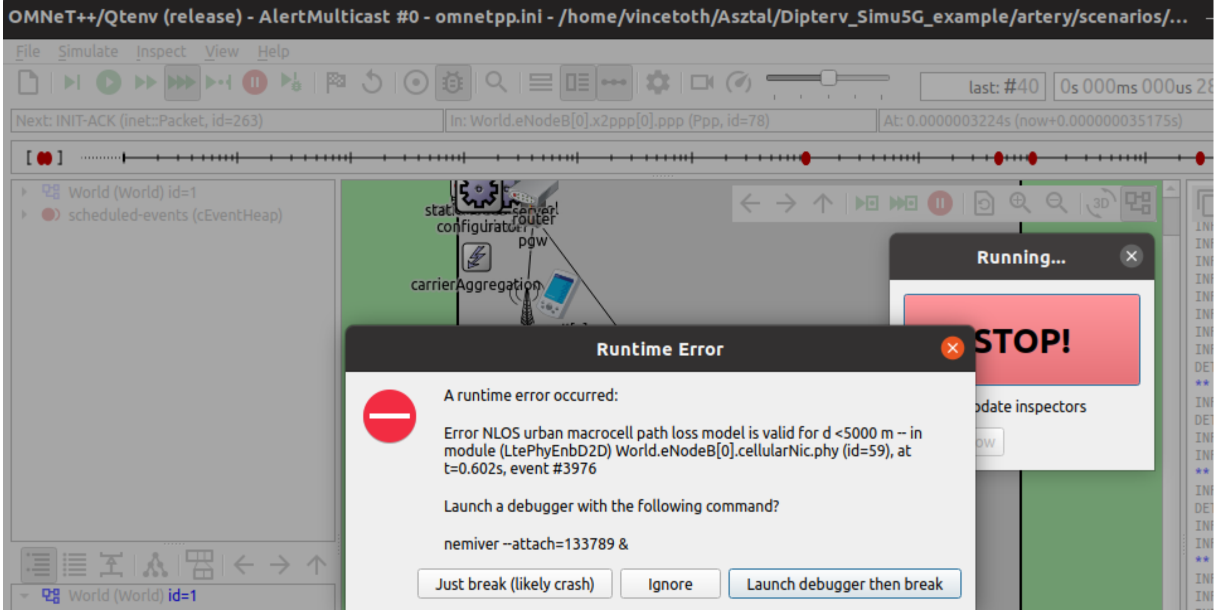
Ezután a példa szcenáriót futtatva ellenőriztem a futást. Ami sikeres volt, így abból kindulva készítettem a saját térképemmel szcenáriót. Az én feladatom elvégzéséhez, mindössze a térképek konfigurációját, ezzel együtt a forgalom és a seed érték beállítását kellett tesztelnem és igazolnom, ezt az alap program futtatása is igazolta.



55. ábra Simu5G Omnet++ felületen

Az 5g Modellt ennek megfelelően két darab eNodeB alkotta, ezek külön lokációban voltak. A Packet Network Data Gateway-en keresztül kapcsolódtak a router-hez, ezeken kívűl egy ueCell nevű eszköz volt még jelen. A járműveim, amik a Győr városi térkép konfigurációjában voltak, ehhez adódtak, mint node-ok.

Ez viszont hibát okozott a futásban, mivel az LTE és Simu5G verzióval ellátott Artery szimulációk modelljei a celluláris hálózatokra, hibára fut öt ezer méter feletti távolságok esetén.



56. ábra Szimulátor hiba 5000 méter felett

A szimulátor működése így korlátozott, mint kiderült. Viszont a hiba megjelenésével, igazolható, hogy a saját szcenárióm beállításai érvényre léptek. Így a Kubernetes keretrendszer is kompatibilis ezzel a verzióval.

Az omnetpp.ini fájlban a konfiguráció engedélyezésével pedig vektor fájlokat is tudtam kinyerni a szimulációból. Ezeket pedig az eredmény feldolgozó backend szolgáltatás tudja feldolgozni, tehát ez is kompatibilis minden Artery verzióval.

```
version 3
run AlertMulticast-0-20220529-21:08:23-135675
attr configname AlertMulticast
attr datetime 20220529-21:08:23
attr datetimef 20220529-210823
attr experiment AlertMulticast
attr inifile omnetpp.ini
attr iterationvars ""
attr iterationvarsd ""
attr iterationvarsf ""
attr measurement ""
attr network artery.simu5g.World
attr processid 135675
attr repetition 0
attr replication #0
attr resultdir results
attr runnumber 0
attr seedset 0
config *.eNodeB[*].nicType "\"LteNicEnbD2D\""
config *.eNodeB[*].**.amcMode "\"D2D\""
```

57. ábra Eredmények Simu5G futásból

A szimulátorhoz egy Dockerfile szükséges, hogy a konténerbe telepítse a megfelelő Omnet++ verziót, a SUMO szimulátort és az Artery programot. Mindezt a saját kódunkat felhasználva, hogy az alkalmazásban létrejött változtatások életbe lépjenek.

A keretrendszer a Dockerfile-t nem tudja kiszolgálni minden Omnet és Artery verzióhoz, így a fejlesztő feladata annak megalkotása. A keretrendszer pedig annyi megkötést tesz, hogy a szcenárió konfigurációs fájljai, az omnetpp.ini fájl és az eredmények külön helyen legyen elérhetőek az image-ben. Ezek helye ugyanis konfigurálható a konfigurációs és eredmény feldolgozó backend Kubernetes ConfigMap-jai által.

8 Szimuláció CI/CD

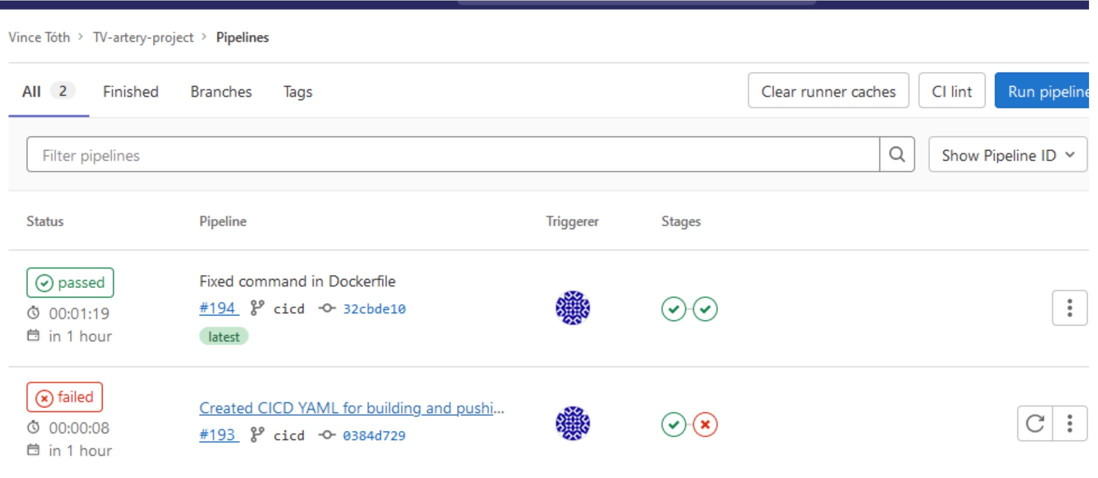
A szimulátorunk előzőleg leírt konténerizálását a Gitlab verziókezelőből automatikusan szeretnénk végrehajtani, az elkészült Image-t eljuttatni a megfelelő konténer Image tárolóba, esetünkben a Docker Hub-ra.

Ehhez a Git branch-re egy gitlab-ci.yaml fájlt kell létrehoznunk ami leíró módon hajtja végre a feladatainkat. Ennek a dolga, Docker build parancsot és Docker Push parancsot hívni, ez egy minimális CI/CD, ami a konténer Image gyártásáért felel. A valós CI/CD rendszerek összetett rendszerekhez készülnek, ahol több tesztelési és integrációs fázis is létezhet.

```
build_docker:
  stage: build_docker
  tags:
    - ci
    - test
  before_script:
    - echo "$DOCKER_PASSWORD" | docker login --username $DOCKER_USER --password-stdin
  dependencies:
    - checks
  needs:
    - checks
  script:
     - echo "Building docker image"
     - docker build -t asyakura/testrepo:$CI_CONCURRENT_ID .
     - docker push asyakura/testrepo:$CI_CONCURRENT_ID
```

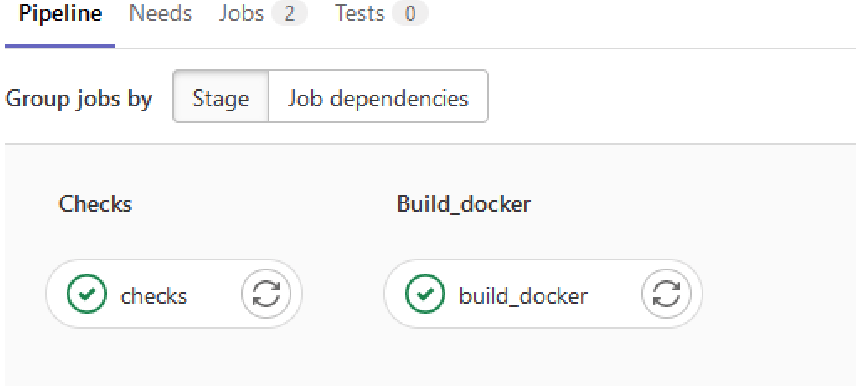
58. ábra Gitlab CI/CD fájl fontos része

A fentebb látott kódrészlet a valós munkát végző része a CI/CD-nek. Emellett egy egyszerű ellenőrzés volt még a CI/CD része. A branch-re push-olva a CI/CD lefut és végrehajtja a Docker build és push parancsokat amennyiben a megfelelő Dockerfile megtalálható a Repository gyökér könyvtárban.



59. ábra CI/CD futás után

Jobban szemügyre véve láthatjuk a két részfeladatot is. Amik egymás után lefutnak.



60. ábra Több CI/CD stage

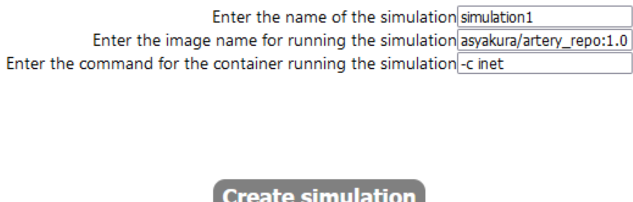
A CI/CD továbbfejleszthető lenne, a konfigurációs backend szolgáltatás automatikus konfigurálására, de ez kívül esik a mostani feladaton.

9 Szimulációs folyamat automatizált működése

A szimuláció futtatásához szükséges, hogy a CI/CD-vel vagy kézzel elkészített és feltöltött konténer image-fájl elérhető legyen. Ezután, ha rendelkezünk a megfelelő konfigurációs fájlokkal megkezdhetjük a futtatást.

Első lépés a szimuláció létrehozása a felületen, kitölteni a három szöveg mezőt értelemszerűen. A szimuláció neve csak korlátozottan a kis betűkből, kötőjelekből és számokból állhat, ez Kubernetes megkötés. A másik két értékre nincs megkötés.

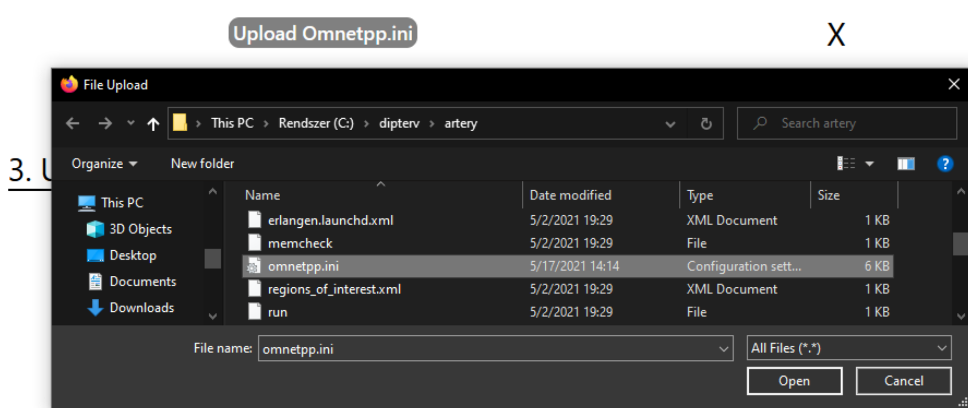
1. Create a simulation



61. ábra Szimuláció létrehozás

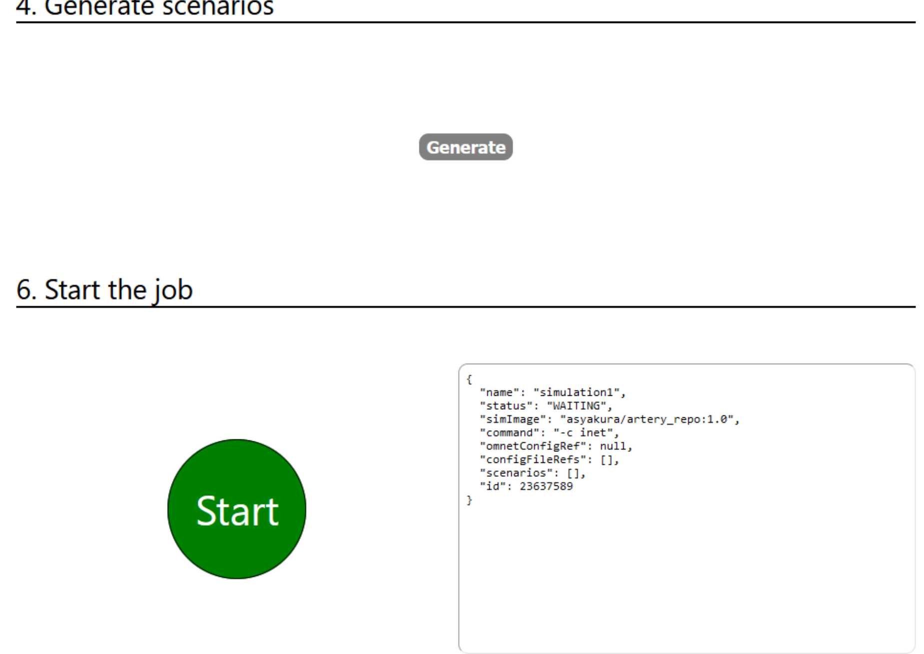
A Create simulation gombra kattintva létrejön a szimulációnk. Ezután feltölthetünk hozzá fájlokat. Az omnetpp.ini és konfig fájlok feltöltése azonos, de omnetpp.ini fájlból csak egy engedélyezett.

2. Upload Omnetpp.ini file



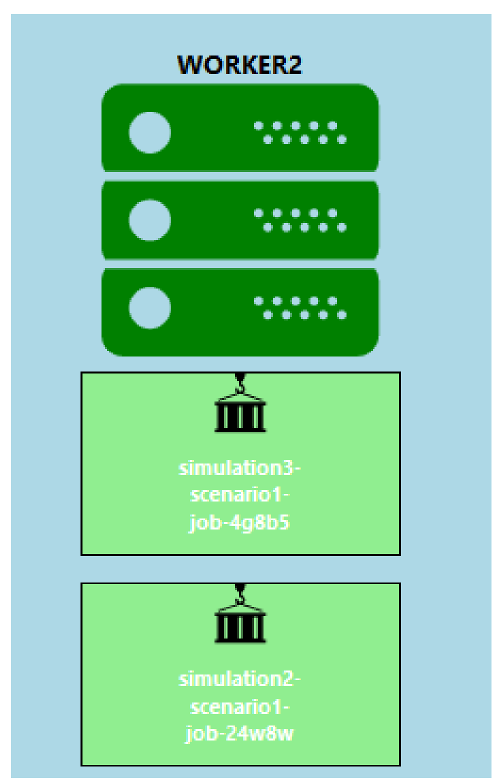
62. ábra Upload files

A fájlok feltöltése után generálhatjuk a szcenáriókat és futtathatunk, ebben a sorrendben.



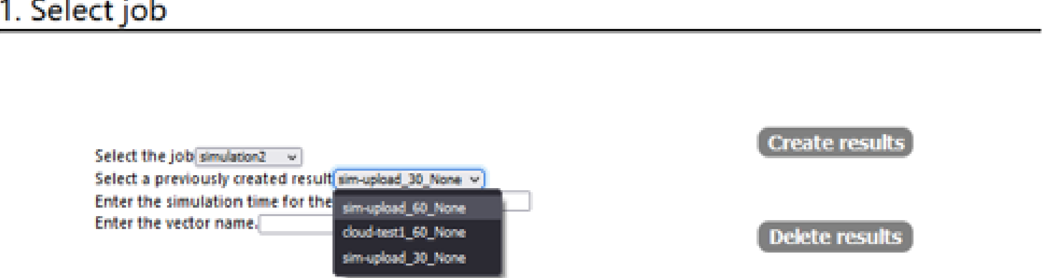
63. ábra Start szimuláció

A sikeres indítás esetén mind Rancher mind a mi vizualizációs felületünkön megjelennek a szcenáriók a szimulációnk nevével.



64. ábra Szcenáriók Pod-jai

Amennyiben minden szcenárió értelem szerint világos zöld lesz, kérhetünk eredmény vizsgálatot a másik felületen.

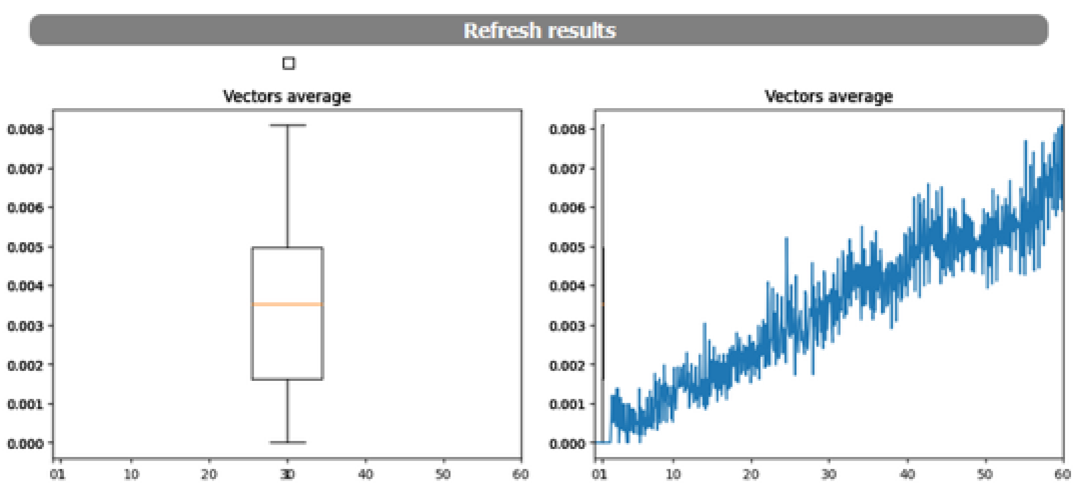


65. ábra Eredmény értékelés készítése

A job-ot kiválasztva tudunk szimulációt választani. Megadhatjuk a szimuláció vizsgálatának idejét, alap értéken ez hatvan másodperc. Illetve a keresett vektor nevét is. De amennyibe nincs vektor név, ami érdekelne, mert esetleg csak egy féle vektort hagytunk a szimuláció eredményében. Akkor üresen is hagyható.

Ezután meg kell várjuk ameddig elkészül a feldolgozás, majd a Refresh results gomb betölti az eredményeket.

2. Results



Stats of the simulation:

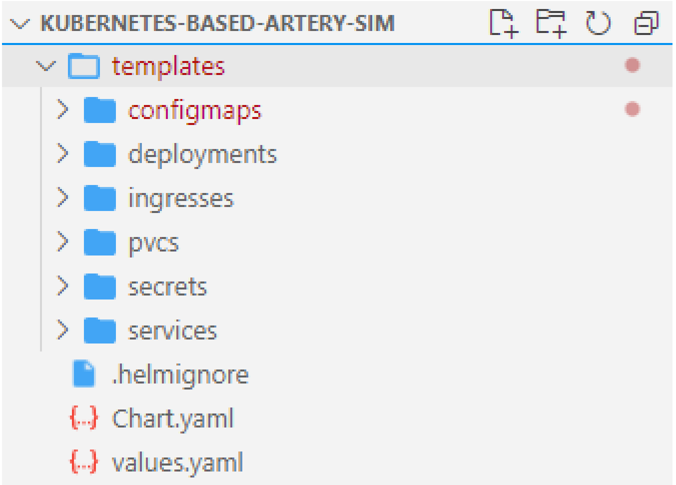
median	0.003520635624538062
mean	0.0033620215877563997
mode	0

66. ábra Eredmények betöltve

Ezzel kész a szimuláció és a kiértékelése. A folyamat eredményeképpen a szimulációnk megmarad a rendszerben, így az újra futtatható. Illetve a fájlrendszer és az adatbázis mozgatása után máshol is betölthető.

10 Telepítés

A telepítés megkönnyítése végett, a felhőben lévő szolgáltatásunkat egy Helm Chart-ba csomagoltam. Ez tartalmaz minden yaml-fájlt ami leírja az alkalmazásokat. Az elkészítéséhez először Helm create paranccsal elkészítek egy template fájlszerkezetet. Majd minden Kubernetesben futó alkalmazás komponenst letöltök yaml formátumban.



67. ábra Helm mappa szerkezet

Majd kitörlöm a státuszt jellemző és számunkra nem fontos alapbeállításokat leíró sorokat. Ezek ugyanis nem adnak hozzá a Helm Chart-hoz viszont elrontják az olvashatóságot és tele szemetelik a yaml fájlokat. Az összes leíró fájlt ezután beteszem a Helm Chart mappaszerkezetbe és megírom a Chart.yaml és values.yaml fájlokat, amik leírják a Chart-ot magát és a yaml fájlokban hivatkozott változók aktuális értékét. A Helm lint paranccsal igazolhatjuk a formátum helyességet.

Ezután a szolgáltatás egy Helm install paranccsal telepíthető, bármely Kubernetes rendszerbe. Persze az infrastruktúra sajátosságaira figyelemmel kell lenni, esetünkben a kritikus részek az Ingress beállítások, és a StorageCalss megoldás. Az Ingress-en a Cors beállításokat kell életre hívni, ehhez a klaszterben telepített Ingress alap beállításaiban kell engedélyezni a snippet-ek használatát. A StorageClass pedig nálunk LocalPath Provisioner segítségével működik, azonban ennek módosítása is szükséges amennyiben hasonló elosztott tárhely megoldás működik a host gépek alatt. Más StorageClass megoldásokra elméletileg működik, a Provisioner átírása után.

11 Összefoglalás

Eredmények értékelése

Az elkészült rendszer mérete, mint feladat jóval meghaladta az elképzeléseimet. Az egyes komponensek kialakítása emiatt gyakran csak a cél funkciókra terjed ki. Emiatt bőséggel akad fejlesztésre lehetőség.

A végeredmény, hogy az elkészült rendszer támogatja a fejlesztőt CI/CD megoldással, amit a Gitlab programmal lehet integrálni. A fejlesztő ezáltal kap egy becsomagol konténerizált szimulátort, amit a felhő infrastruktúrában használhat. A rendszer konfigurációs komponense, lehetővé teszi a szimulátor használatát olyan fokig, mint amit a lokális futtatásnál megszokhattunk. A szimulációk újra futtathatóak, és a konfigurációk is megtekinthetőek.

Az eredmények feldolgozása automatikus. Az eredményeket vizuálisan és szövegesen is megjeleníti, ezáltal lehetőséget adva a szimulátor viselkedésének vizsgálatára. A webfelület, képes a backend szolgáltatások funkcióit elérhetővé tenni a felhasználó számára. Emellett vizuálisan megjeleníti az eredményeket és a klaszterben futó szimulációkat, ezáltal valós időben mutatva a felhő szimulációs rendszer futását.

A Kubernetes rendszert a virtuális host gépektől kezdve létrehoztam, és konfiguráltam, hogy felhő infrastruktúrában legyen végrehajtható a keretrendszer és a szimulációk is. A szimulációs keretrendszerhez, Helm telepítési lehetőséget készítettem, ezáltal bármilyen Kubernetes rendszerbe integrálható.

A Gitlab rendszerébe egy egyszerű CI/CD megírás sikeres volt. Egy Dockerfile-t igényelve bármilyen Gitlab projekben használható a Runner telepítése után.

A Simu5G Artery szcenáriómat a példa alapján megalkottam. A SUMO-t saját térkép fájllal és generált forgalommal konfiguráltam. Ezek után a Simu5G példába injektál saját szcenáriót teszteltem. Az eredmények alapján a rendszer bármilyen Artery szimulációt tud futtatni, minimális, de konfigurálható megkötésekkel és saját Dockerfile megírása után.

Továbbfejlesztési lehetőségek

A Frontend fejlesztéseit a megjelenésén túl mindig a backend szolgáltatások funkcióihoz kell igazítani, így az azokon ajánlott fejlesztéseket folyamatosan integrálni kell.

A konfigurációs backend szolgáltatás hibakezelése egy komplex és szerteágazó probléma, a Spring API hibakezelése megoldott viszont a Kubernetes operációs feladatok hosszabb átgondolást és tesztelést igényelnek, hogy az felhasználó kényelmesen használhassa.

A konfigurálást az omnetpp.ini fájl alapján tesszük, ugyanakkor ennek manipulálásával a felületen is lehetne új konfigurációkat létrehozni a meglévők átírásával.

A futás végeztével automatikus észleléssel indíthatóak lennének az eredmények feldolgozásai.

A CI/CD kiegészítése, valamilyen automatikus Dockerfile generálás lehetőségével.

Az eredmények vizuális megjelenítésére más grafikonok implementálása is lehetséges.

Az eredmények feldolgozása után az eredmények vizualizálása és vizsgálata szintén bővíthető, hogy kettő vagy több szimulációt egy ábrán megjelenítve jobban összevethetőek legyenek.

Az eredményeket egy közös adatbázisban tárolva később nagyobb jelentéseket lehet több szimuláció tapasztalatai alapján gyártani.

A Kubernetes rendszerben használt StorageClass megoldás integrálását implementálni más StorageClass rendszerekre is.

Köszönetnyilvánítás

Köszönöm a kitartó és hozzáértő munkát konzulensemnek, dr. Bokor Lászlónak. Köszönöm a szakértő segítséget Wippelhauser Andrásnak. Végül köszönöm a Datatronic Kft.-nek a lehetőséget, hogy valós infrastruktúrán dolgozhattam, és külső konzulensemnek Csányi Zsoltnak a munkáját.

Szójegyzék

5G	Ötödik generációs mobil- illetve vezeték nélküli hálózat
ADAS	Advanced Driving Assistance System
API	Application Programming Interface
AWS	Amazon Web Services
C++	C++ Programozási nyelv
CD	Continouos Delivery
CI	Continouos Integration
CI/CD	Continouos Integration/Continouos delivery
CNCF	Cloud Native Computing Foundation
CORS	Cross Origin Resource Sharing
CRUD	Create Read Update Delete
CSS	Cascaded Style Sheet
CSV	Comma Separated Values (fájlformátum)
DOM	Document Object Model
DTO	Data Transfer Object
DevOps	Development + Operations
ETCD	Elosztott, konzisztens kulcs-érték tár, konfigurációhoz, ütemezéshez használatos. Neve a /etc UNIX mappa nevéből és a "d" mint "distributed" (elosztott) szavakból áll össze.
ETSI	European Telecommunications Standards Institute
FS	File System
HTML	HyperText Markup Language
HiL	Hardware in the Loop
IBM	International Business Machines (vállalat)

IEEE	Institute of Electrical and Electronics Engineers
INET	Nyílt forráskódú keretrendszer az OMNeT++ szimulációs környezethez
IP	Internet Protocol
ITS-G5	Intelligent Transport Systems Az 5GHz-es frekvencia sváon
JPA	Java Persistence API
JS	JavaScript
JSON	JavaScript Object Notation
JSX	JavaScript XML
K3S	Egy lightweight Kubernetes disztribúció, a Kubernetes K8S rövidítéséhez képest a 3 a lightweight jellegére utal.
KVM	Kernel based Virtual Machine
MAC	Media Access Control
MySQL	SQL alapú adatbázis kezelő rendszer neve.
OVIRT	Oracle linux VIRTualization manager
Omnet++	Objective Modular Network Testbed in C++
PV	Persistent Volume
PVC	Persistent Volume Claim
REST	Representational State Transfer
RKE	Rancher Kubernetes Engine
S3	Amazon Simple Storage Service (S3) objektum
SDK	Software Development Kit
SELinux	Security-Enhanced Linux
SPA	Single Page Application
SQL	Structured Query Language
SSH	Secure Shell

SSL	Secure Sockets Layer
SUMO	Simulation of Urban MObility
TCP	Transmission Control Protocol
TraCI	Traffic Control Interface
UDP	User Datagram Protocol
UI	User Interface
V2X	Vehicle-to-Everything
VANET	Vehicular Ad Hoc Networks
VM	Virtual Machine
VeiNS	Vehicles in Network Simulation
YAML	YAML Ain't Markup Language

Ábrajegyzék

1. ábra Artery moduláris rendszer [2] 10
2. ábra Szerkesztett autópálya térkép[3] 11
3. ábra SUMO grafikus interfész[3] 12
4. ábra Omnet++ grafikus felület 13
5. ábra Omnet++ eredmény vizualizációs felület [12] 14
6. ábra Vizualizált szimulációs eredmény[12] 14
7. ábra Egyszerűsített virtualizált szimulátor futtatás 16
8. ábra Mikroszolgáltatások és monolitok[15] 17
9. ábra Mikroszolgáltatások felépítése [15] 18
10. ábra Deployment és Pod-ok kapcsolata[20] 21
11. ábra Ingress és Service kapcsolata a Pod-okkal[19] 23
12. ábra Szeparált konténerizálás 25
13. ábra Rendszer komponensek 29
14. ábra CI/CD folyamat 30
15. ábra Webalkalmazás felület lehetőségei. 32
16. ábra Grafana monitoring ábra[21] 33
17. ábra Scheduler Algoritmus példa ábra. 36
18. ábra Deployment szimulációs konfiguráció 37
19. ábra Jobs szimulációs konfiguráció. 38
20. ábra Ovirt menedzser grafikus felület[25] 39
21. ábra Pythond Ovirt SDK példa 40
22. ábra Három node konfigurálása RKE yaml fájllal 43
23. ábra Kubernetes Node architectúra 44
24. ábra Rancher bejelentkezés 45
25. ábra Klaszter áttekintés 46
26. ábra Klaszterben futó Rancher 47
27. ábra Rancher beépített applikációk 48
28. ábra Klaszter állapot adatmodell 49
29. ábra Szimulációk adatmodellje 50
30. ábra Szcenárió generálás 52
31. ábra Elosztott tárhely a szimulációknak 53

32. ábra Szcenárió Job futás 54
33. ábra Szcenárió logjai 54
34. ábra Szimuláció aggregált box plot 56
35. ábra Szimuláció aggregált vonalas ábra 57
36. ábra Eredmény összegző adatok 57
37. ábra Klaszter állapot frontend oldal 60
38. ábra Új szimuláció létrehozása frontend oldal 60
39. ábra Fájl upload és szcenárió generálás 61
40. ábra Szimuláció összegzés 61
41. ábra Konfiguráció újra töltésé 62
42. ábra Eredmény kiválasztása 62
43. ábra Azonos szimulációk eltérő idővel való eredménye 63
44. ábra Választott eredmények vizuálisan 63
45. ábra Java Backend Dockerfile 64
46. ábra Python Django backend Dockerfile 65
47. ábra SPA Frontend Dockerfile 66
48. ábra Kubernetes telepítés architektúra 66
49. ábra Kubernetes Service-k 67
50. ábra Ingress beállítások 67
51. ábra MinIO felület 67
52. ábra Konfigurációs fájlok ConfigMap-jai 68
53. ábra Sumo térkép Grafikus felületen 69
54. ábra Simu5G tesztek 70
55. ábra Simu5G Omnet++ felületen 70
56. ábra Szimulátor hiba 5000 méter felett 71
57. ábra Eredmények Simu5G futásból 72
58. ábra Gitlab CI/CD fájl fontos része 73
59. ábra CI/CD futás után 74
60. ábra Több CI/CD stage 74
61. ábra Szimuláció létrehozás 75
62. ábra Upload files 75
63. ábra Start szimuláció 76
64. ábra Szcenáriók Pod-jai 77

65. ábra Eredmény értékelés készítése 77
66. ábra Eredmények betöltve 78
67. ábra Helm mappa szerkezet... 79

Irodalomjegyzék

[1] R. Riebl, „Artery V2X Simulation Framework”. http://artery.v2x-research.eu/ (elérés 2021. december 19.).

[2] R. Riebl, H.-J. Gunther, C. Facchi, és L. Wolf, „Artery: Extending Veins for VANET applications”, in *2015 International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*, Budapest, Hungary, jún. 2015, o. 450–456. doi: 10.1109/MTITS.2015.7223293.

[3] V. Tóth, „C-ITS szolgáltatások modellezése HiL szimulációs környezetben”. Elérés: 2021. december 19. [Online]. Elérhető: https://diplomaterv.vik.bme.hu/hu/Theses/CITS-szolgaltatasok-modellezese-HiL/Attachment/49891

[4] Autotalks, „Seoul City Transforms Transport System by Deploying V2X Systems Powered by Autotalks”. [Online]. Elérhető: https://auto-talks.com/seoul-city-transforms-transport-system-by-deploying-v2x-systems-powered-by-autotalks/

[5] Eclipse Foundation, „SUMO: Simulation of Urban MObility”. https://www.eclipse.org/sumo/ (elérés 2021. december 19.).

[6] German Aerospace Center (DLR) and others, „SUMO User Documentation”. https://sumo.dlr.de/docs/ (elérés 2021. december 19.).

[7] A. Wegener, M. Piórkowski, M. Raya, H. Hellbrück, S. Fischer, és J.-P. Hubaux, „TraCI: an interface for coupling road traffic and network simulators”, in *Proceedings of the 11th communications and networking simulation symposium on - CNS ’08*, Ottawa, Canada, 2008, o. 155. doi: 10.1145/1400713.1400740.

[8] A. Varga, „OMNeT++”, in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Güneş, és J. Gross, Szerk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, o. 35–59. doi: 10.1007/978-3-642-12331-3_3.

[9] L. Mészáros, A. Varga, és M. Kirsche, „INET Framework”, in *Recent Advances in Network Simulation*, A. Virdis és M. Kirsche, Szerk. Cham: Springer International Publishing, 2019, o. 55–106. doi: 10.1007/978-3-030-12842-5_2.

[10] OpenSim Ltd., „OMNeT++ Discrete Event Simulator”. https://omnetpp.org/ (elérés 2021. december 19.).

[11] OpenSim Ltd., „OMNeT++ Tutorials and Technical Articles”. https://docs.omnetpp.org/ (elérés 2021. december 19.).

[12] 2019 OpenSim Ltd, „Omnet++ Visualizing the Results”. [Online]. Elérhető: https://docs.omnetpp.org/tutorials/tictoc/part6/

[13] C. Sommer *és mtsai.*, „Veins: The Open Source Vehicular Network Simulation Framework”, in *Recent Advances in Network Simulation*, A. Virdis és M. Kirsche, Szerk. Cham: Springer International Publishing, 2019, o. 215–252. doi: 10.1007/978-3-030-12842-5_6.

[14] M. Luksa, *Kubernetes in Action*. Manning, 2017. Elérés: 2021. december 19. [Online]. Elérhető: https://books.google.hu/books?id=WTgzEAAAQBAJ

[15] R. Gnatyk, „Microservices vs Monolith: which architecture is the best choice for your business?” https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/ (elérés 2021. december 19.).

[16] Citrix Systems, „What is containerization and how does it work?” https://www.citrix.com/solutions/app-delivery-and-security/what-is-containerization.html (elérés 2021. december 19.).

[17] D. Vohra, „Hello Kubernetes. In: Kubernetes Microservices with Docker.” Apress, Berkeley, CA., 2016. Elérés: 2021. december 19. [Online]. Elérhető: https://doi.org/10.1007/978-1-4842-1907-2_2
[18] I. Eldridge, „What Is Container Orchestration?” https://newrelic.com/blog/best-practices/container-orchestration-explained (elérés 2021. december 19.).
[19] The Kubernetes Authors, „Kubernetes Documentation”. [Online]. Elérhető: https://kubernetes.io/docs/home/
[20] M. Hoogendoorn, „How Kubernetes Deployments Work”, *The New Stack*. https://thenewstack.io/kubernetes-deployments-work/
[21] Grafana Labs, „Grafana docs”. https://grafana.com/docs/ (elérés 2021. december 19.).
[22] The Linux Foundation, „Kubernetes Pods”. https://kubernetes.io/docs/concepts/workloads/pods/ (elérés 2021. december 19.).
[23] MinIO, Inc., „Minio official”. https://min.io/
[24] T. Menouer, „KCSS: Kubernetes container scheduling strategy”, *J. Supercomput.*, köt. 77, sz. 5, o. 4267–4293, 0 2021, doi: 10.1007/s11227-020-03427-3.
[25] G. Sheremeta és S. Bonazzola, „oVirt official site”. https://www.ovirt.org/
[26] S. Bonazzola, „oVirt Python SDK Documentation”. https://www.ovirt.org/documentation/doc-Python_SDK_Guide/
[27] Rancher, „Rancher documentation”. https://rancher.com/docs/k3s/latest/en/
[28] etcd Authors, „etcd documentation”. https://etcd.io/docs/v3.5/
[29] Kubernetes Java Client Authors, „Kubernetes Java Client”. https://github.com/kubernetes-client/java

Függelékek

Rke cluster.yaml

```
nodes:
    - address: 10.2.108.102
      user: Rancher
      role:
        - controlplane
        - etcd
        - worker
      hostname_override: master1
      taints:
        - key: CriticalAddonsOnly
          value: True
          effect: NoExecute
    - address: 10.2.108.100
      user: Rancher
      role:
        - worker
      hostname_override: worker1
    - address: 10.2.108.101
      user: Rancher
      role:
        - worker
      hostname_override: worker2

cluster_name: local

addon_job_timeout: 120
```

Gitlab CI/CD yaml:

```
variables:
  GIT_STRATEGY: clone
  DOCKER_PASSWORD: $DOCKER_PASSWORD
  DOCKER_USER: $DOCKER_USER
stages:
  - checks
  - build_docker

checks:
  stage: checks
  tags:
    - ci
    - test
  script:
    - echo "Checking code before deploy and build"

build_docker:
  stage: build_docker
  tags:
    - ci
    - test
```

```
  before_script:
    - echo "$DOCKER_PASSWORD" | docker login --username $DOCKER_USER --password-
stdin
  dependencies:
    - checks
  needs:
    - checks
  script:
     - echo "Building docker image"
     - docker build -t asyakura/testrepo:$CI_CONCURRENT_ID .
     - docker push asyakura/testrepo:$CI_CONCURRENT_ID
```

Python backend Dockerfile:

```
# The image you are going to inherit your Dockerfile from
FROM python:3.8

ARG VERSION=5.6.2
WORKDIR /root
RUN apt-get update && apt-get install -y \
    bison \
    build-essential \
    flex \
    libxml2-dev \
    wget \
    zlib1g-dev \
    && rm -rf /var/lib/apt/lists/*
RUN     wget     https://github.com/omnetpp/omnetpp/releases/download/omnetpp-
$VERSION/omnetpp-$VERSION-src-core.tgz \
    --progress=bar:force:noscroll -O omnetpp-src-core.tgz && \
    tar xf omnetpp-src-core.tgz && \
    rm omnetpp-src-core.tgz && \
    mv omnetpp-$VERSION /omnetpp
WORKDIR /omnetpp
ENV PATH /omnetpp/bin:$PATH
RUN ./configure WITH_QTENV=no WITH_OSG=no WITH_OSGEARTH=no && \
    make -j $(nproc) base MODE=release

RUN make -j $(nproc) base MODE=debug
COPY ./scavetool /usr/local/sbin/scavetool
RUN chmod 777 /usr/local/sbin/scavetool
RUN scavetool

ENV PYTHONUNBUFFERED 1

RUN mkdir /figures
RUN mkdir /csv
RUN mkdir /python_backend

WORKDIR /python_backend

COPY . .

COPY ./requirements.txt /requirements.txt
```

```
RUN pip install --upgrade pip

RUN pip3 install matplotlib
RUN pip3 install numpy
RUN pip3 install pandas

RUN pip install -r /requirements.txt

CMD python manage.py migrate && python manage.py runserver 0.0.0.0:8080
```

JavaScript frontend Dockerfile:

```
FROM node:current-alpine3.15
RUN mkdir /usr/app
COPY ../ /usr/app
WORKDIR /usr/app
RUN npm ci
CMD npm run start
```

Java backend Dockerfile:

```
#Basic image
FROM openjdk:11.0.14-jre

#Update for apt
RUN apt update

#The current directory will be this one
WORKDIR /usr/src/management-backend

#Copy the jar files into the container
COPY ./target/*.jar /usr/src/management-backend/service.jar
COPY src/main/resources/application.yaml /usr/src/management-backend/config/application.yaml

#Start the application jar file with config location and config yaml file name
CMD java -jar /usr/src/management-backend/service.jar --spring.config.location=file:/usr/src/management-backend/config/ --spring.config.name=application
```

Python results_maker.py

```
import json
import subprocess
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statistics
import logging

from base.models import Simulation

logging.basicConfig(filename='artery_python_backend.log', filemode='w',
                    level=logging.INFO,
                    format='%(asctime)s-%(process)d-%(levelname)s            -
%(message)s',
                    datefmt='%d-%b-%y %H:%M:%S')
f = open('/directory_config/config.json')
config = json.load(f)

FIGURES_PATH = config["FIGURES_PATH"]
CSV_PATH = config["CSV_PATH"]
RESULTS_BASE_DIR = config["RESULTS_BASE_DIR"]


def parse_if_number(s):
    try:
        return float(s)
    except:
        return True if s == "true" else False if s == "false" else s if s else
None


def parse_ndarray(s):
    return np.fromstring(s, sep=' ') if s else None


def get_vecs(file_path):
    logging.info("Loading vectors from %s.", file_path)
    vecs = pd.read_csv(file_path, converters={
        'attrvalue': parse_if_number,
        'binedges': parse_ndarray,
        'binvalues': parse_ndarray,
        'vectime': parse_ndarray,
        'vecvalue': parse_ndarray})
    return vecs[vecs.type == 'vector']


def get_average_vec_for_scenario(run_name, vectors, sim_time=60.0):
    ###
    # Gives back the average value for all the nodes, for all the time fragments
in the simulation
    # ###
    myvectors = vectors[vectors.run == run_name]
    time = 0.0
    my_dict = {}
    live_nodes = {}
    while time <= sim_time:
```

```
        value = 0
        time += 0.1
        my_dict[round(time, 1)] = value
        live_nodes[round(time, 1)] = value
    for x in range(len(myvectors)):
        myvector = myvectors.iloc[x]
        index_of_item = 0
        for item in myvector.vectime:
            if item <= sim_time:
                my_dict[item] += myvector.vecvalue[index_of_item]
                live_nodes[item] += 1
                index_of_item += 1
    # print(my_dict)
    # print(live_nodes)
    time = 0
    my_sums = []
    while time <= sim_time:
        time += 0.1
        if live_nodes[round(time, 1)] == 0:
            my_sums.append(0)
        else:
            my_sums.append(my_dict[round(time, 1)] / live_nodes[round(time, 1)])
    return my_sums


def get_average_for_sim(vectors, sim_time=60.0, vector_name=None):
    ###
    # Gets the average vector values for a simulation
    # ###
    logging.info("Getting summary data from simulation.")
    if vector_name:
        print("Filtering for vector name: " + vector_name)
        vectors = vectors[vectors.name == vector_name]
    average_vectors = {}
    runs = vectors.run.unique()
    for run in runs:
        average_vectors[run]   =   get_average_vec_for_scenario(vectors=vectors,
run_name=run, sim_time=sim_time)
    time = 0
    my_dict = {}
    my_sums = []
    if len(average_vectors) == 0:
        return my_sums
    while time <= sim_time:
        value = 0
        my_dict[round(time, 1)] = value
        time += 0.1
    for k in average_vectors:
        vector_list = average_vectors[k]
        time = 0
        for x in vector_list:
            my_dict[round(time, 1)] += x
            time += 0.1
    time = 0

    while time <= sim_time:
        my_sums.append(my_dict[round(time, 1)] / len(average_vectors))
        time += 0.1
    logging.info("Returning summary data from simulation.")
```

```
    return my_sums


def     run_scavetool(directory_path,     out_put_directory="/var/lib/artery/",
simulation_name="no_name"):
    print("Running Scavetool.")
    logging.info("Running scavetool in directory: %s, for simulation: %s, putting
the output to: %s",
                 directory_path, simulation_name, out_put_directory)
    command = 'scavetool x ' + directory_path + '/*vec ' + directory_path + '/*sca
-o ' \
              + out_put_directory + simulation_name + '.csv'
    process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE)
    return process


def get_box_plot(sim_name, sums, result_name):
    print("Getting box plot data.")
    logging.info("Making box plot for simulation: %s.", sim_name)
    data = [sums]
    data_to_plot = data
    fig = plt.figure(1, figsize=(10, 10))
    ax = fig.add_subplot(111)
    bp = ax.boxplot(data_to_plot)
    fig.savefig(FIGURES_PATH + result_name + '_box.png', bbox_inches='tight')
    return True


def get_vector_plot(sim_name, sums, result_name, sim_time=60.0):
    print("Getting vector plot data.")
    logging.info("Making plot for simulation: %s.", sim_name)

    simtimes = []
    if len(sums) == 0:
        plt.plot(simtimes, sums, drawstyle='default')
        plt.savefig(FIGURES_PATH     +     result_name     +     '_vector.png',
bbox_inches='tight')
        return True
    for x in range(int(sim_time) * 10):
        simtimes.append(round(x * 0.1, 1))
    plt.plot(simtimes, sums, drawstyle='default')
    plt.title("Vectors average")
    plt.xlim(0, int(sim_time))
    plt.savefig(FIGURES_PATH + result_name + '_vector.png', bbox_inches='tight')
    # plt.show()
    return True


def get_string_data(sim_name, sums, result_name):
    print("Getting String data.")
    logging.info("Making stats for simulation: %s.", sim_name)
    stats = {}
    if len(sums) == 0:
        stats['info'] = "No dataset."
        stats_file = open(FIGURES_PATH + result_name + '_stats.json', "w")
        json.dump(stats, stats_file)
        stats_file.close()
        return stats
    stats['median'] = statistics.median(sums)
```

```
    stats['mean'] = statistics.mean(sums)
    stats['mode'] = statistics.multimode(sums)
    stats_file = open(FIGURES_PATH + result_name + '_stats.json', "w")
    json.dump(stats, stats_file)
    stats_file.close()
    return stats


def       make_all_results(csv_path,       sim_name="no_name",       sim_time=60,
vector_name=None, result_name="no_name_result"):
    logging.info("Making all results with sim time:" + str(sim_name) + " | and
with sim_name:" + str(sim_name)
                 + " | and with vector name:" + str(vector_name) + " with the csv
directory being:" + str(csv_path)
                 )
    print("Making all results.")
    vectors = get_vecs(csv_path + sim_name + ".csv")
    sums      =      get_average_for_sim(vectors=vectors,      sim_time=sim_time,
vector_name=vector_name)
    get_vector_plot(sim_name=sim_name,       sums=sums,       sim_time=sim_time,
result_name=result_name)
    get_box_plot(sim_name=sim_name, sums=sums, result_name=result_name)
    data_dict       =       get_string_data(sim_name=sim_name,       sums=sums,
result_name=result_name)
    print("Results making finished.")
    return data_dict


def make_results_for_sim(simulation_id):
    logging.info("Making   simulation   results   for   simulation   id:   %s.",
simulation_id)
    simulation = Simulation.objects.get(id=simulation_id)
    temp_simulation = simulation
    process       =       run_scavetool(directory_path=RESULTS_BASE_DIR       +
simulation.sim_name + "/",
                            out_put_directory=CSV_PATH,
                            simulation_name=simulation.sim_name)
    process.wait()
    if process.returncode != 0:
        logging.error("Scavetool process exited with none 0 code for simulation
id: %s", simulation_id)
        return 0
    temp_simulation.state = "SCAVETOOL_DONE"
    simulation.save()
    logging.info("Scavetool finished for simulation id: %s.", simulation_id)
    data_dict               =               make_all_results(csv_path=CSV_PATH,
sim_name=simulation.sim_name, sim_time=simulation.sim_time,
                                 vector_name=simulation.vec_name,
result_name=simulation.name)
    logging.info("Plot making finished for simulation id: %s. Average stats: %s.",
simulation_id, str(data_dict))
    temp_simulation.state = "FINISHED"
    simulation.save()
```