



**Budapesti Műszaki és Gazdaságtudományi Egyetem**  
Villamosmérnöki és Informatikai Kar  
Hálózati Rendszerek és Szolgáltatások Tanszék

# Istio operator CLI fejlesztése

SZAKDOLGOZAT

*Készítette*  
Csepi Árpád

*Konzulens*  
dr. Farkas Károly  
Gyurácz Kristóf

2023. június 1.



## SZAKDOLGOZAT FELADAT

**Csepi Árpád**

üzemmmérnök-informatikus hallgató részére

### Istio Operator CLI fejlesztése, bővítése

Az Istio egy nyílt platform mikroszolgáltatások összekapcsolására, kezelésére és biztonságossá tételére, így egyre inkább a Kubernetes szolgáltatásháló kiépítésének szabványává válik. Az Istio számos összetevőre és meglehetősen összetett telepítési sémára épül, amely összetevők telepítése, frissítése és működtetése a platform alapos ismeretét igényli.

A Cisco által fejlesztett Istio Operator célja, hogy ezeket a feladatokat automatizálja, egyszerűsítse, és lehetővé tegye a népszerű szolgáltatásháló használati eseteket – mint például a multi cluster összevonás, több átjáró támogatása vagy erőforrás-egyeztetés – egyszerű, magasabb szintű absztrakciók bevezetésével.

A hallgató feladata egy parancssori (CLI) felület fejlesztése és bővítése a Cisco Istio Operator szoftverhez Go nyelven, valamint a felület integrálása a meglévő istio-operator projektbe. A hallgató feladatának a következőkre kell kiterjednie:

- Igazodva a meglévő projekt struktúrájához adaptálja a korábban létrehozott CLI felületet, és bővítse annak funkcionalitását multi-cluster telepítési lehetőséggel;
- Fejlesszen a CLI-hez teszteket a már meglévő és az újonnan fejlesztett funkciók tesztelésére, és ezeket integrálja a meglévő CLI rendszerhez;
- Dokumentálja a CLI működését, valamint a tesztek eredményét.

**Egyetemi témavezető:** Dr. Farkas Károly, egyetemi docens, BME-VIK HIT tanszék

**Ipari konzulens:** Gyurácz Kristóf, Software engineer, Cisco Systems Magyarország Kft.

Budapest, 2023. február 1.

Dr. Imre Sándor

egyetemi tanár  
tanszékvezető

**Témavezetői vélemények:**

Egyetemi témavezető: ☐ Beadható, ☐ Nem beadható, dátum:

aláírás:

# Tartalomjegyzék

<b>Kivonat</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>1. Bevezetés</b>	<b>1</b>
<b>2. Konténerizációs technológiák</b>	<b>3</b>
2.1. A monolitikus alkalmazásokról a mikroszolgáltatásokra való áttérés . . . . .	3
2.2. Konténerek előnyei . . . . .	3
2.3. Docker platform . . . . .	4
<b>3. Konténer alapú alkalmazáskezelő szoftverek</b>	<b>6</b>
3.1. Kubernetes bemutatása . . . . .	6
3.2. A Kubernetes működésének megértése . . . . .	7
3.2.1. A Control Plane . . . . .	7
3.2.2. A worker node-ok . . . . .	8
3.2.3. A podok bevezetése . . . . .	8
3.2.4. A Service bemutatása . . . . .	9
3.2.5. Konténeralkalmazások konfigurálása, ConfigMap bemutatása . . . . .	9
3.2.6. A Secrets bemutatása . . . . .	9
3.2.7. Kliens könyvtárak az API-kiszolgálóval való kommunikációhoz . . . . .	10
3.2.8. Egyéni API-objektumok definiálása . . . . .	11
3.2.9. Az operátor bemutatása . . . . .	11
3.3. Kubernetes in Docker . . . . .	13
<b>4. Helm csomagkezelő</b>	<b>14</b>
4.1. Helm chart-ok . . . . .	14

4.2.	Helm működése . . . . .	15
4.3.	Helm repository . . . . .	15
4.4.	Mikor érdemes használni . . . . .	15
<b>5.</b>	<b>Service Mesh</b>	<b>17</b>
5.1.	Istio . . . . .	17
5.2.	Istio operátor . . . . .	18
5.3.	Cisco Istio operátor . . . . .	18
5.4.	Multi cluster . . . . .	19
5.4.1.	Több klaszteres architektúra tervek . . . . .	20
5.4.2.	Kubernetes-központú vs. hálózat-központú konfiguráció . . . . .	20
5.4.3.	A több klaszteres architektúra előnyei . . . . .	21
5.4.4.	A több klaszteres Kubernetes kihívásai . . . . .	23
<b>6.</b>	<b>Felhőszolgáltatások</b>	<b>24</b>
6.1.	A felhőalapú számítástechnika jellemzői . . . . .	24
6.2.	A felhőalapú telepítések típusai . . . . .	25
6.2.1.	Publikus felhő . . . . .	25
6.2.2.	Privát felhő . . . . .	25
6.2.3.	Hibrid felhő . . . . .	25
6.3.	Felhőalapú számítástechnikai szolgáltatások . . . . .	25
6.3.1.	Infrastruktúra mint szolgáltatás . . . . .	26
6.3.2.	Platform mint szolgáltatás . . . . .	26
6.3.3.	Szoftver mint szolgáltatás . . . . .	26
6.3.4.	Funkció mint szolgáltatás . . . . .	26
6.4.	A felhőalapú számítástechnika előnyei . . . . .	27
<b>7.</b>	<b>Go programozási nyelv</b>	<b>29</b>
7.1.	A programozási nyelv . . . . .	29
7.2.	Csomagkezelő . . . . .	30
7.3.	Tesztelés . . . . .	31
<b>8.</b>	<b>Elosztott verziókezelő rendszerek</b>	<b>33</b>
8.1.	Git . . . . .	33

8.2. GitHub . . . . .	34
8.2.1. GitHub Actions ismertető . . . . .	34
8.2.2. A GitHub Actions elemei . . . . .	35
8.2.3. Workflows . . . . .	35
8.2.4. Events . . . . .	35
8.2.5. Jobs . . . . .	36
8.2.6. Actions . . . . .	36
8.2.7. Runners . . . . .	36
8.2.8. Egy példa workflow . . . . .	36
<b>9. KLI CLI</b>	<b>38</b>
9.1. Tervezés . . . . .	38
9.2. Model-View-Controller ismertetése . . . . .	38
9.3. Megvalósítás . . . . .	38
9.3.1. Front-end . . . . .	38
9.3.2. Back-end . . . . .	39
9.4. Tesztelés . . . . .	40
9.5. Kritikai elemzése . . . . .	40
9.6. Továbbfejlesztési lehetőségek . . . . .	40
9.6.1. Operátor frissítése . . . . .	40
9.6.2. Log rendszer . . . . .	40
9.6.3. Virtuálisgép integráció megvalósítása . . . . .	40
9.6.4. Klaszterek létrehozása telepítés előtt . . . . .	41
9.6.5. Automatizált post-install . . . . .	41
<b>10.Összefoglaló</b>	<b>42</b>
<b>Köszönetnyilvánítás</b>	<b>43</b>
<b>Irodalomjegyzék</b>	<b>44</b>

## HALLGATÓI NYILATKOZAT

Alulírott *Csepi Árpád*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot 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 autentikált 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.

Budapest, 2023. június 1.

---

*Csepi Árpád*  
hallgató

# Kivonat

Ennek a szakdolgozatnak a célja, hogy ismertesse a mai világ nélkülözhetetlen felhő-alapú számítástechnikai rendszerek automatizált telepítését, menedzselését és működését a Kubernetes szoftver segítségével, illetve ingyenes és nyílt forráskódú megoldást kínáljon ezekre.

Az automatizált telepítés és klaszterek összekapcsolásának segítésére készítettem egy konzolos applikációt Go programozási nyelv segítségével. Ez a felhasználóbarát megoldás elősegíti, hogy több a téma iránt érdeklődő embereknek nyújtson egy eszközt, ami könnyen használható, átlátható és módosítható.

A szakdolgozatom első szakaszában ismertetem azoknak a technológiáknak a hátterét, melyekre a program működésének megértéséhez szükség van.

Ezután bemutatom a programom tervezésének folyamatát, majd részletesen leírom az elkészített program szerkezeti felépítését és működését. Itt rávilágítok a program megvalósított funkcióira, a moduláris felépítésére, kiemelve a modulok legfontosabb részleteit. Bemutatom a tesztelési módszereket és dokumentálom az eredményeit.

A szakdolgozat végén összegzem a tanulságokat, elért eredményeket és további lehetséges fejlesztési lehetőségeket vázolok fel.

# Abstract

The purpose of this thesis is to describe the automated deployment, management and operation of cloud computing systems that are essential in today's world using Kubernetes software, and to provide a free and open-source solution for these.

To help with automated deployment and cluster attaching, I created a console application using the Go programming language. This user-friendly solution helps to provide several people interested in the topic with a tool that is easy to use, transparent and modifiable.

In the first section of my thesis, I will describe the background of the technologies that are needed to understand how the program works.

I will then describe the process of designing my program, and then describe in detail the structure and operation of the program I have created. Here I will highlight the implemented features of the program, its modular structure, highlighting the most important details of the modules. I will present the testing methods and document the results.

At the end of the thesis, I summarise the lessons learned, the results achieved and outline further possible improvements.



# 1. fejezet

## Bevezetés

A mai rohanó világban a modern alkalmazások egyre összetettebbé válnak. Ezek a mikroszolgáltatások egy elosztott architektúrára épülnek, amelyek különböző platformokon és környezetekben futnak. Ez a komplexitás olyan kihívásokat hoz magával, mint a szolgáltatások közötti kommunikáció kezelése, biztonságának biztosítása, és az egyes szolgáltatások teljesítményének felügyelete. Az Istio service mesh termék ezeket a kihívásokat segíti áthidalni olyan funkciókkal amelyek egyszerűsítik a mikroszolgáltatás-alapú alkalmazások és klaszterek kezelését. Cisco által fejlesztett Istio operátor Calisti Service Mesh Manager termék része, mely felhasználóbarát telepítést, klaszter menedzselést és megfigyelést tesz lehetővé.

A Calisti Service Mesh Manager fizetős termék mellé kínálok egy ingyenes és teljesen nyílt forráskódú alternatívát. Az én általam fejlesztett program ugyanazt az Istio operátort használva implementál bizonyos funkciókat, mellyel felhasználóbarát és automatizált lehetőséget kínálok telepítésre, törlésre, és több klaszter összekapcsolására. A célom ezzel a programmal, hogy több emberhez eljusson a mikroszolgáltatás alapú szoftveres megoldások és képesek legyenek további fejlesztésekkel bővíteni és testreszabni azt.

A szakdolgozat fejezeteit ajánlott sorban elolvasni, mert a fejezetek támaszkodnak előző fejezetekben leírt technológiákra. A témakörök amikről szó lesz röviden összefoglalva:

- Konténerek használatának előnyei mikroszolgáltatások üzemeltetésekor
- A népszerű Docker platform megismertetése, konténerképek rétegeinek bemutatása
- Kubernetes alkalmazáskezelő működésének illusztrálása, master-slave architektúra és a különböző típusú erőforrások leírása
- Kubernetes in Docker milyen lehetőségeket kínál ha valaki lokális gépen szeretne Kubernetes klasztert létrehozni.
- Helm csomagkezelő ki lesz fejtve miért is nagyon hasznos egy jó csomagkezelő a Kubernetes klaszterünknek.
- Service Mesh fejezetben az Istio szolgáltatásháló megoldásról lesz szó, ami segít menedzselni a mikroszolgáltatásainkat akár több klaszteren keresztül is.
- Boncolva lesznek a felhő telepítésének típusai és a szolgáltatások típusai, amin futni fognak majd a mikroszolgáltatások a kubernetes segítségével.

- A Go programozási nyelv ismertetésre kerül, mivel ez a legelterjedtebb programozási nyelv a mikroszolgáltatások fejlesztéséhez és mivel ebben készült az én szakdolgozatom munkája is.
- Verziókezelő rendszerek bemutatásra kerülnek, mivel a legjobb megoldás a piacon a nyílt forráskódú szoftverek fájlainak követésére.
- A GitHub Actions tesztelési lehetőség ismertetve lesz példával illusztrálva.
- Miután ez a sok technológiai ismeret át lett adva szó lesz a fejlesztett program felépítésének tervezéséről, kivitelezéséről, teszteléséről, kritikájáról és további fejlesztési potenciáljairól.

## 2. fejezet

# Konténerizációs technológiák

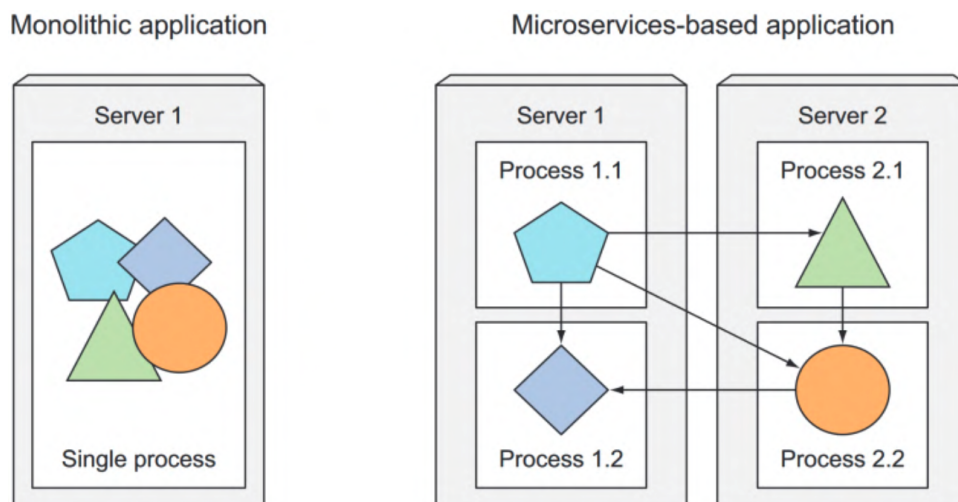
### 2.1. A monolitikus alkalmazásokról a mikroszolgáltatásokra való áttérés

A monolitikus alkalmazások olyan komponensekből állnak, amelyek mindegyike szorosan összekapcsolt és egyetlen egységként kell fejleszteni, telepíteni és kezelni, mivel mind egyetlen operációs rendszerfolyamatként futnak. Az alkalmazás egy részének módosítása az egész alkalmazás újratelepítését igényli, és idővel a részek közötti határok hiánya a részek közötti korlátlanul növekvő komplexitáshoz és következésképpen az egész rendszer minőségének romlásához vezet, mivel a részek közötti függőségek korlátlanul növekednek. A rendszer növekvő terhelésének kezeléséhez vagy vertikálisan kell skálázni a szervereket több CPU, memória és egyéb szerverelem hozzáadásával, vagy horizontálisan kell skálázni az egész rendszert további szerverek felállításával és az alkalmazás több példányának futtatásával. Bár a vertikális skálázás általában nem igényel semmilyen változtatást az alkalmazáson, viszonylag gyorsan drágul, és a gyakorlatban mindig van egy felső korlátja. A horizontális skálázás viszont viszonylag olcsó hardveresen, de nagy változtatásokat igényelhet az alkalmazás kódjában, és nem mindig lehetséges - az alkalmazás bizonyos részei rendkívül nehezen vagy szinte lehetetlenül horizontálisan skálázni (például relációs adatbázisok). Ha egy monolitikus alkalmazás bármely része nem skálázható, akkor az egész alkalmazás skálázhatatlan lesz, hacsak nem lehet valahogy felosztani a monolitot.

Ezek és más problémák arra kényszerítettek minket, hogy elkezdjük az összetett monolitikus alkalmazásokat kisebb, egymástól függetlenül telepíthető komponensekre, úgynevezett mikroszolgáltatásokra bontani. Minden mikroszolgáltatás önálló folyamatként fut (lásd az 1. ábrát), és egyszerű, jól definiált interfészekon (API-kon) keresztül kommunikál más mikroszolgáltatásokkal. [8]

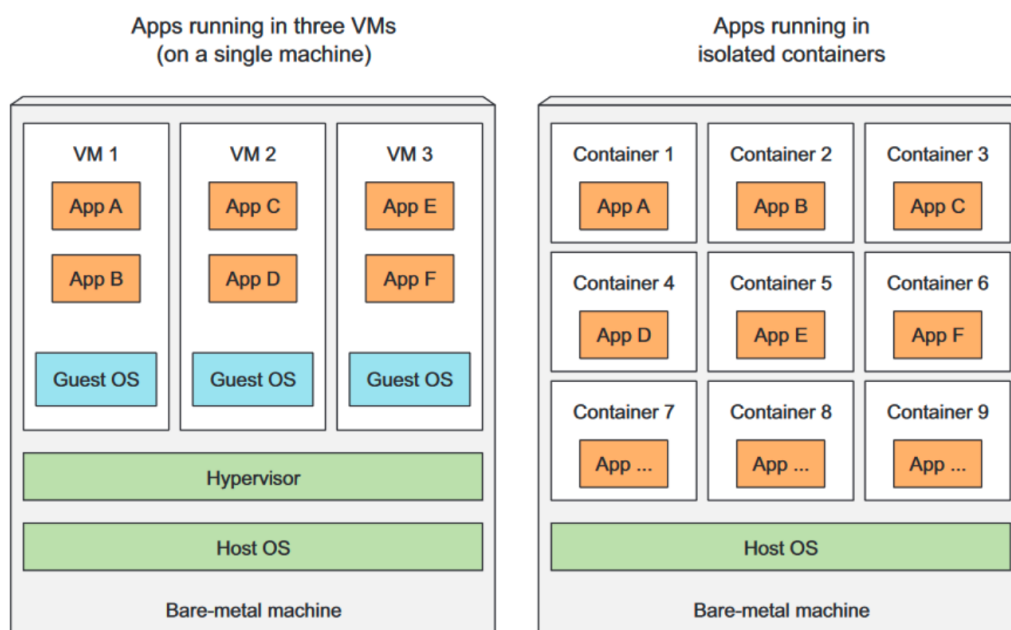
### 2.2. Konténerek előnyei

Ahelyett, hogy virtuális gépeket használnának az egyes mikroszolgáltatások környezetének elkülönítésére, a fejlesztők a Linux konténertechnológiákhoz fordulnak. Ezek lehetővé teszik, hogy több szolgáltatást futtassanak ugyanazon a számítógépen, miközben nem csak más környezetet állítanak fel mindegyiknek, hanem el is szigetelik őket egymástól, hasonlóan a virtuálisgépekhez, de sokkal kevesebb pazarlással. Egy konténerben futó folyamat a számítógép operációs rendszerén belül fut, mint az összes többi folyamat (ellentétben



**2.1. ábra.** Komponensek egy monolitikus alkalmazáson belül és az önálló mikroszolgáltatások [8]

a virtuálisgépekkel, ahol a folyamatok külön operációs rendszerben futnak). A konténerben lévő folyamat azonban továbbra is el van szigetelve a többi folyamattól. Magának a folyamatnak úgy tűnik, mintha csak ő futna a gépen és annak operációs rendszerében. [8]

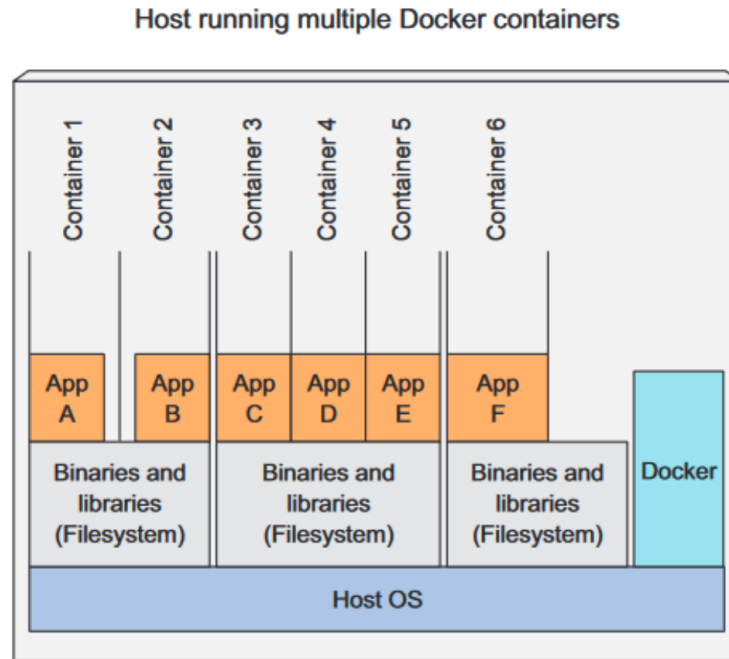


**2.2. ábra.** Virtuálisgépek használata alkalmazáscsoportok elkülönítésére és az egyes alkalmazások elkülönítése konténerekkel [8]

## 2.3. Docker platform

Bár a konténertechnológiák már régóta léteznek, a Docker konténerplatform elterjedésével váltak szélesebb körben ismertté. A Docker volt az első konténerrendszer, amely a konténereket könnyen hordozhatóvá tette különböző gépek között. Leegyszerűsítette azt a

folyamatot, hogy nem csak az alkalmazást, hanem annak összes könyvtárát és egyéb függőséget, sőt a teljes operációs rendszer fájlrendszerét is egy egyszerű, hordozható csomagba csomagolja, amellyel az alkalmazás bármely más, Dockert futtató gépen elérhetővé tehető. A Docker-alapú konténerképek rétegekből állnak, amelyek több képen is megoszthatók és újrafelhasználhatók. Ez azt jelenti, hogy egy képnek csak bizonyos rétegeit kell letölteni, ha a többi réteg már korábban letöltésre került egy másik konténer futtatása során, amely ugyanezeket a rétegeket tartalmazza. [8]



**2.3. ábra.** Docker konténerek futás közben [8]

## 3. fejezet

# Konténer alapú alkalmazáskezelő szoftverek

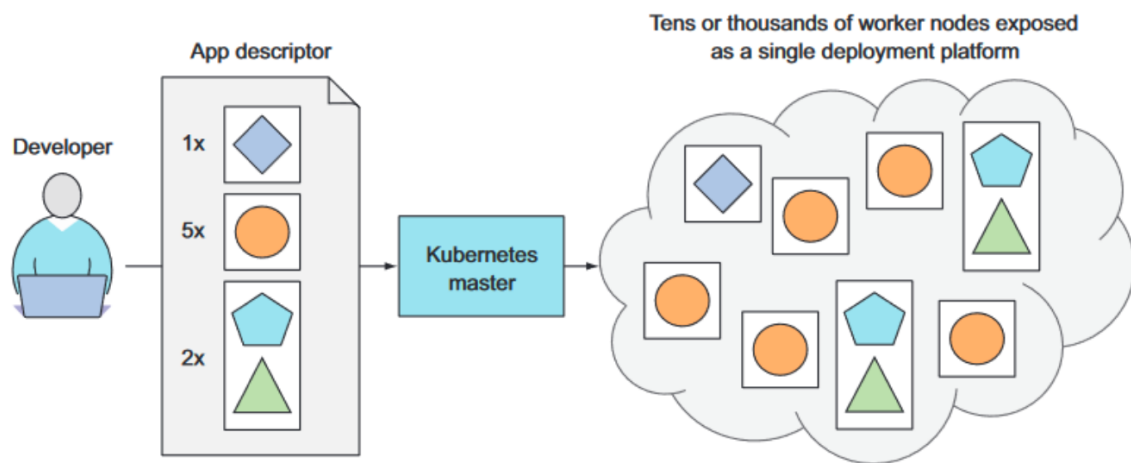
### 3.1. Kubernetes bemutatása

Láthattuk, hogy ahogy nő a rendszerben a telepíthető alkalmazáskomponensek száma, egyre nehezebb lesz mindegyiket kezelni. A Google volt valószínűleg az első vállalat, amely felismerte, hogy sokkal jobb módszerre van szüksége a szoftverkomponensek telepítéséhez és kezeléséhez, valamint a globálisan skálázható infrastruktúrájához. Egyike annak a kevés vállalatnak a világon, amely több százezer szervert üzemeltet, és amelynek a telepítések kezelésével kell foglalkoznia ilyen hatalmas léptékben. Ez arra kényszerítette őket, hogy olyan megoldásokat dolgozzanak ki, amelyekkel több ezer szoftverkomponens fejlesztése és telepítése kezelhetővé és költséghatékonyá tehető.

A Kubernetes egy olyan szoftverrendszer, amely lehetővé teszi, hogy egyszerűen telepítsünk és kezeljünk rajta konténeres alkalmazásokat. A Linux konténerek funkcióira támaszkodik, hogy heterogén alkalmazásokat futtasson anélkül, hogy ismernie kellene ezen alkalmazások belső részleteit, és anélkül, hogy ezeket az alkalmazásokat manuálisan kellene telepíteni az egyes hosztokon. Mivel ezek az alkalmazások konténerekben futnak, nem befolyásolják az ugyanazon a szerveren futó többi alkalmazást, ami kritikus fontosságú, ha teljesen különböző szervezetek alkalmazásait futtatja ugyanazon a hardveren. Ez a felhőszolgáltatók számára kiemelkedő fontosságú, mivel a hardverük lehető legjobb kihasználására törekszenek, miközben a hosztolt alkalmazások teljes elszigeteltségét is fenn kell tartaniuk. A Kubernetes lehetővé teszi, hogy szoftveralkalmazásait több ezer számítógépcsomóponton futtassa, mintha ezek a csomópontok mind egyetlen, hatalmas számítógép lenne. Absztrahálja a mögöttes infrastruktúrát, és ezáltal egyszerűsíti a fejlesztést, a telepítést és a kezelést mind a fejlesztő, mind az üzemeltetési csapatok számára. Az alkalmazások telepítése a Kubernetes segítségével mindig ugyanaz, függetlenül attól, hogy a klaszter (későbbiekben cluster) csak néhány csomópontot (későbbiekben node) tartalmaz vagy több ezret. A cluster mérete egyáltalán nem számít. A további node-ok egyszerűen a telepített alkalmazások számára rendelkezésre álló erőforrások további mennyiségét jelentik. [8]

## 3.2. A Kubernetes működésének megértése

A rendszer egy fő csomópontból (későbbiekben master node) és tetszőleges számú munkás csomópontból (későbbiekben worker node) áll. Amikor a fejlesztő elküldi az alkalmazások listáját a master-nek, a Kubernetes telepíti azokat a worker node-okra. (lásd a 4. ábrát). Az, hogy egy komponens melyik worker node-on landol, nem számít (és nem is szabadna számítania) - sem a fejlesztőnek, sem a rendszergazdának. A fejlesztő megadhatja, hogy bizonyos alkalmazásoknak együtt kell futniuk, és a Kubernetes ugyanarra a worker node-ra telepíti őket. Mások szétszóródnak a cluster-ben, de ugyanúgy tudnak egymással kommunikálni, függetlenül attól, hogy hova telepítik őket. [8]



**3.1. ábra.** A Kubernetes az egész adatközpontot egyetlen telepítési platformként teszi elérhetővé. [8]

Hardveres szinten egy Kubernetes cluster sok node-ból áll, amelyek két típusra oszthatók:

- A master node, amely a Kubernetes vezérlőjének (későbbiekben Control Plane) ad otthont, amely az egész Kubernetes rendszert irányítja és kezeli.
- A worker node, amely a tényleges telepített alkalmazásokat futtatja.

### 3.2.1. A Control Plane

A Control Plane az, ami a cluster-t irányítja és működésre készíti. Több komponensből áll, amelyek egyetlen master node-on futhatnak, vagy több node-ra oszthatók és replikálhatók a magas rendelkezésre állás biztosítása érdekében. Ezek az összetevők a következők:

- Az API szerver, amellyel Ön és a többi control plane komponens kommunikál.
- Az ütemező (későbbiekben Scheduler), amely ütemezi az alkalmazásokat (az alkalmazás minden telepíthető komponenséhez hozzárendel egy worker node-ot).
- A vezérlő menedzser (későbbiekben Controller Manager), amely cluster szintű funkciókat lát el, mint például a komponensek replikálása, a munkás node-ok nyomon követése, a node-ok hibáinak kezelése stb.

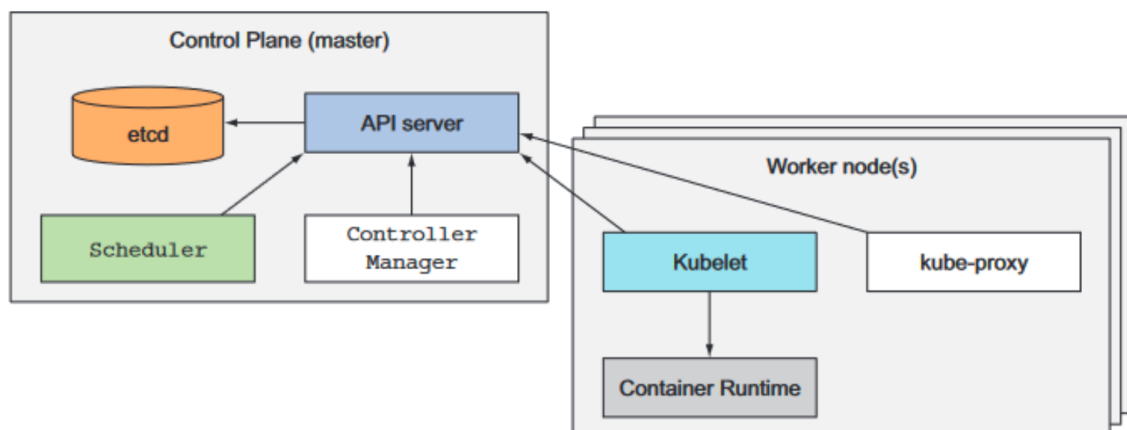
- etcd, egy megbízható elosztott adattároló, amely tartósan tárolja a cluster konfigurációját.

A Control Plane összetevői tartják és vezérlik a cluster állapotát, de nem futtatják az alkalmazásokat. Ezt a worker node-ok végzik. [8]

### 3.2.2. A worker node-ok

A worker node-ok azok a gépek, amelyek a konténerizált alkalmazásokat futtatják. Az alkalmazások futtatásának, felügyeletének és szolgáltatásnyújtásának feladatát a következő komponensek végzik: [8]

- Docker, rkt vagy más konténer futtató, amely futtatja a konténereket.
- A Kubelet kommunikál az API-kiszolgálóval és kezeli a konténereket a node-ján.
- A Kubernetes Service Proxy (kube-proxy), amely elosztja a hálózati forgalmat az alkalmazáskomponensek között.

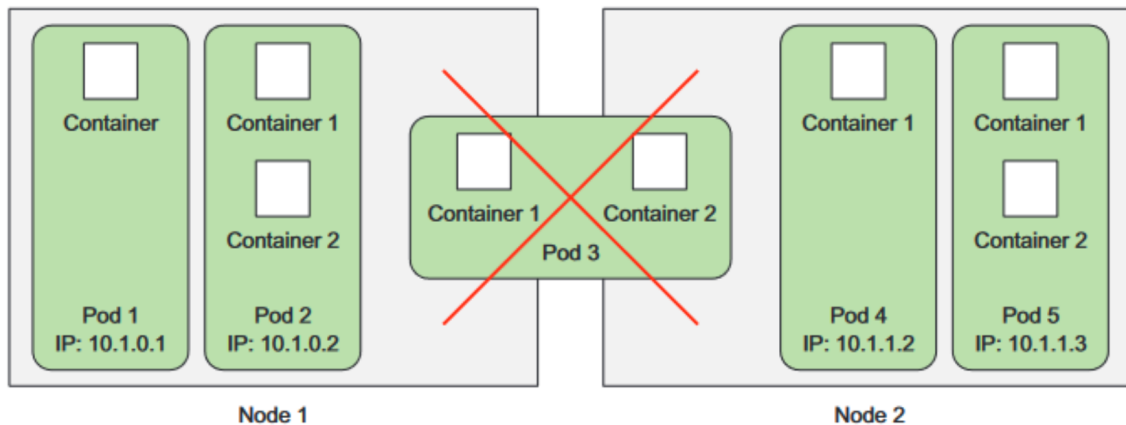


3.2. ábra. A Kubernetes cluster-t alkotó komponensek [8]

### 3.2.3. A podok bevezetése

A Kubernetes nem foglalkozik közvetlenül az egyes konténerekkel. Ehelyett a több, együtt elhelyezett konténer koncepcióját használja. Ezt a konténerekből álló csoportot Podnak nevezzük. A pod egy vagy több szorosan összefüggő konténer csoportja, amelyek mindig ugyanazon a worker node-on és ugyanazon a Linux névtereken futnak együtt. Minden pod olyan, mint egy különálló logikai gép, saját IP-vel, hostnévvel, folyamatokkal, amely egyetlen alkalmazást futtat. Az alkalmazás lehet egyetlen folyamat, amely egyetlen konténerben fut, vagy lehet egy fő alkalmazási folyamat és további támogató folyamatok, amelyek mindegyike saját konténerben fut. Egy podban lévő összes konténer látszólag ugyanazon a logikai gépen fut, míg a többi podban lévő konténerek, még ha ugyanazon a worker node-on futnak is, látszólag másikon futnak. [8]





**3.3. ábra.** Egy pod minden konténere ugyanazon a node-on fut.  
Egy pod soha nem terjed ki két node-ra. [8]

### 3.2.4. A Service bemutatása

A Kubernetes Service egy olyan erőforrás, amelyet azért hozunk létre, hogy egyetlen, állandó belépési pont legyen egy podok csoportjához, amelyek ugyanazt a szolgáltatást nyújtják. Minden szolgáltatásnak van egy IP-címe és portja, amelyek a szolgáltatás létezése alatt soha nem változik. Az felhasználók kapcsolatot nyithatnak az adott IP-címre és portra és ezek a kapcsolatok az adott szolgáltatást támogató podok egyikéhez kerülnek továbbításra. Így a szolgáltatás felhasználóinak nem kell ismerniük a szolgáltatást nyújtó egyes podok helyét és címét, így ezek a podok bármikor áthelyezhetők a cluster-en belül. [8]

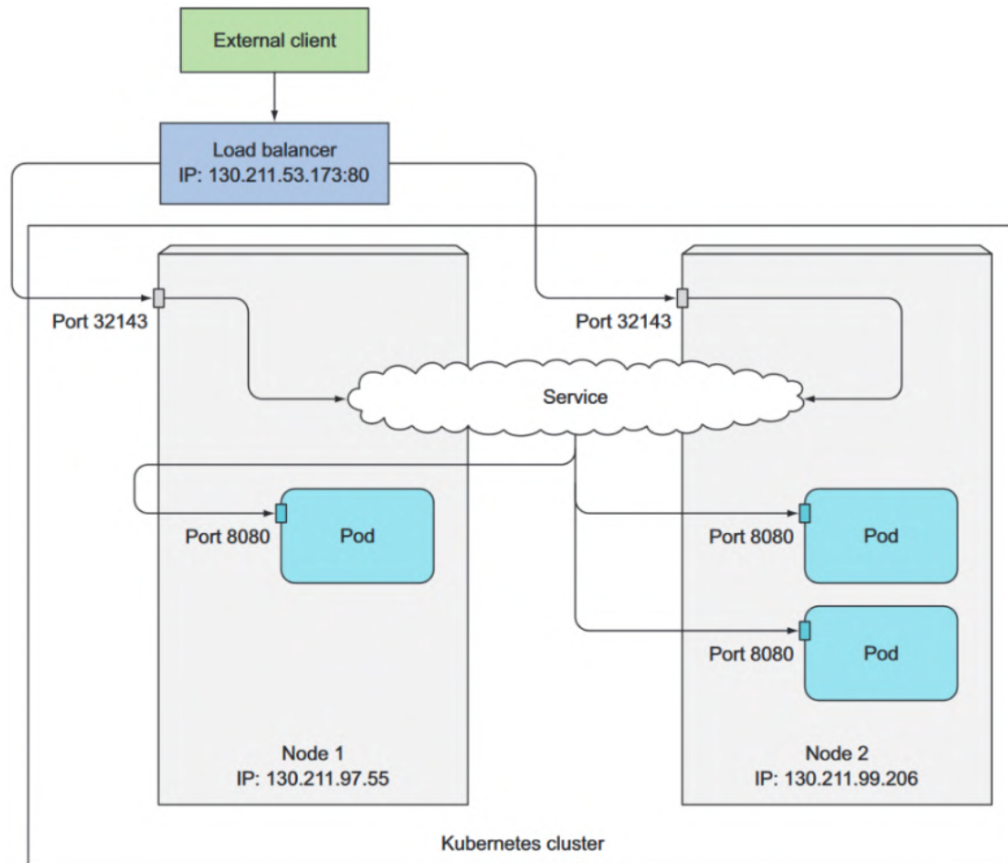
### 3.2.5. Konténeralkalmazások konfigurálása, ConfigMap bemutatása

Ha kihagyjuk azt a tényt, hogy a konfigurációt magába az alkalmazásba süthetjük, egy új alkalmazás fejlesztésének megkezdésekor általában úgy kezdjük, hogy az alkalmazást parancssori argumentumokon keresztül konfiguráljuk. Ezután, ahogy a konfigurációs lista opciók listája növekszik, a konfigurációt egy konfigurációs fájlba helyezhetjük át. A konfigurációs opciók átadásának egy másik módja, amely széles körben népszerű a konténeres alkalmazásokban, a környezeti változókon keresztül történik. Egy egyszerűbb megoldás azonban lehetővé teszi, hogy a konfigurációs adatokat egy legfelső szintű Kubernetes- erőforrásba helyezze, és azt, valamint az összes többi erőforrás-definíciót ugyanabban a Git-repositoryban vagy bármilyen más fájlalapú tárolóban tárolja.

A Kubernetes lehetővé teszi a konfigurációs opciók elkülönítését egy különálló, ConfigMap nevű objektumba, amely egy kulcs/érték párokat tartalmazó leképezés, amelynek értékei a rövid literáloktól a teljes konfigurációs fájlorig terjednek. Az alkalmazásnak nem kell közvetlenül olvasnia a ConfigMap-et, vagy tudnia kell, hogy az létezik. A ConfigMap tartalma vagy környezeti változóként, vagy egy kötetben lévő fájlként kerül átadásra a konténerek számára. [8]

### 3.2.6. A Secrets bemutatása

A konfiguráció általában érzékeny információkat is tartalmaz, például hitelesítő adatokat és privát titkosítási kulcsokat, amelyeket biztonságban kell tartani. Az ilyen infor-



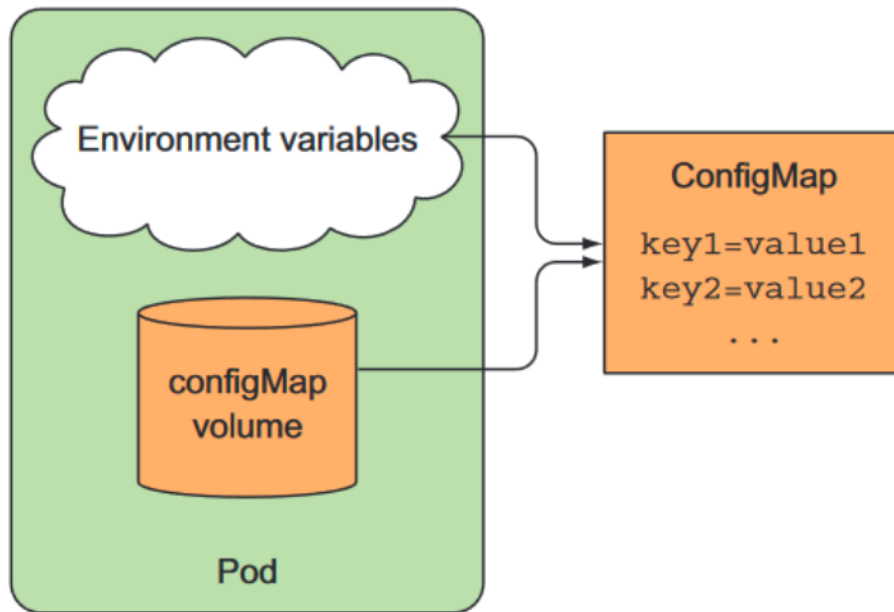
**3.4. ábra.** Szolgáltatás megnyitása külső felhasználók számára [8]

mációk tárolására és terjesztésére a Kubernetes egy különálló objektumot biztosít, amelyet Secretnek hívnak. A titkok hasonlóak a ConfigMaps-hoz, ők is leképezések, amelyek kulcs-érték párokat tartalmaznak. Ugyanúgy használhatók, mint egy ConfigMap. A Kubernetes segít megőrizni a Secrets biztonságát azzal, hogy minden Secret csak azokhoz a node-okhoz jut el, amelyek a Secret hozzáférésre szoruló podokat futtatják. Emellett a node-okon maguk a Secret-ek mindig a memóriában tárolódnak, és soha nem íródnak fizikai tárolókra, ami a lemezek törlését igényelné a Secret törlése után. Magán a master node-on (pontosabban az etcd-ben) a Secret-eket korábban titkosítatlan formában tárolták, ami azt jelentette, hogy a master node-ot védeni kell az tárolt érzékeny adatok biztonsága érdekében. Ez nem csak az etcd tárolás biztonságban tartására terjedt ki, hanem arra is, hogy megakadályozzuk, hogy illetéktelen felhasználók használhassák az API-kiszolgálót, mivel bárki, aki podokat tud létrehozni, fel tudja csatolni a Secretet a podba, és azon keresztül hozzáférhet az érzékeny adatokhoz. A Kubernetes 1.7-es verziójától az etcd titkosított formában tárolja a Secret-eket, ami sokkal biztonságosabbá teszi a rendszert. [8]

### 3.2.7. Kliens könyvtárak az API-kiszolgálóval való kommunikációhoz

A Kubernetes közösségnek számos speciális érdekcsoportja és munkacsoportja van, amelyek a Kubernetes ökoszisztéma egyes részeire összpontosítanak. Jelenleg két Kubernetes API klienskönyvtár létezik, amelyeket az API Machinery speciális érdekcsoport (SIG) támogat: [8]

- Golang client (<https://github.com/kubernetes/client-go>)



**3.5. ábra.** A podok ConfigMaps-okat használnak környezeti változókon és configMap köteteken keresztül [8]

- Python (<https://github.com/kubernetes-incubator/client-python>)

### 3.2.8. Egyéni API-objektumok definiálása

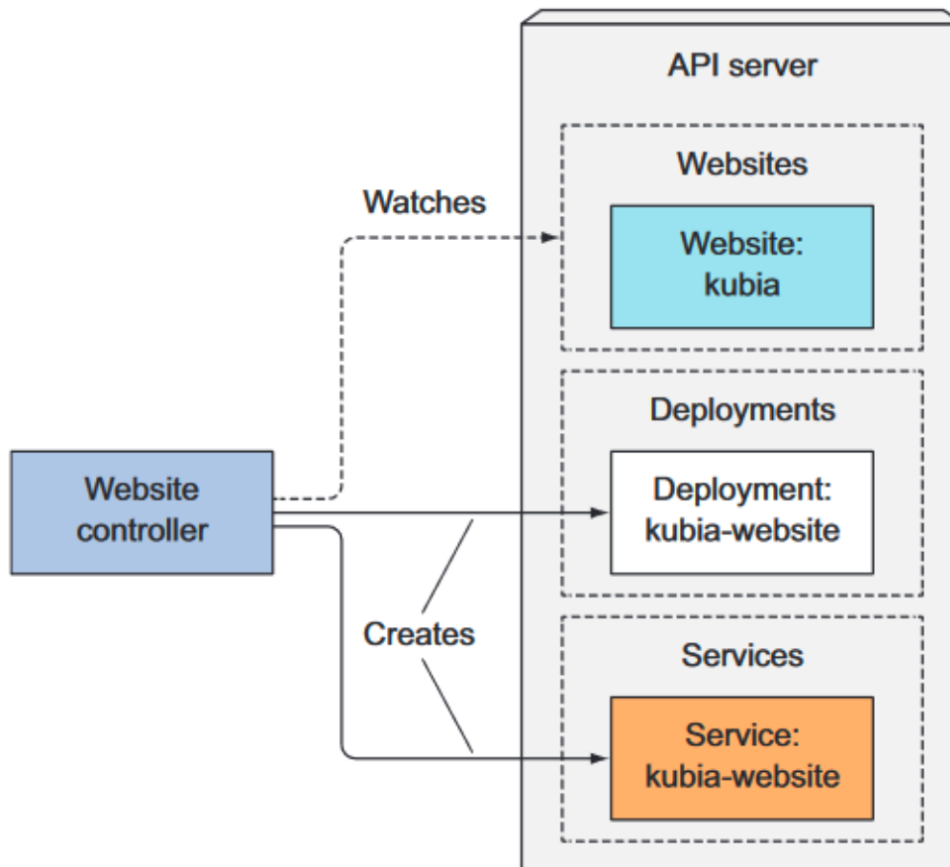
Ahogy a Kubernetes ökoszisztéma fejlődik, egyre több és több magas szintű objektumot fog látni, amelyek sokkal speciálisabbak lesznek, mint a Kubernetes által ma támogatott erőforrások. Ahelyett, hogy Deployments, Services, ConfigMaps és hasonlókkal foglalkozna, egész alkalmazásokat vagy szoftverszolgáltatásokat reprezentáló objektumokat fog létrehozni és kezelni.

Egy egyéni vezérlő fogja megfigyelni ezeket a magas szintű objektumokat, és ezek alapján alacsony szintű objektumokat hoz létre. Például ahhoz, hogy a webhelyobjektumok egy szolgáltatáson keresztül elérhető webkiszolgáló podot futtassanak, létre kell hoznia és telepítenie kell egy webhelyvezérlőt, amely figyeli az API-kiszolgálót a webhelyobjektumok létrehozására, majd létrehozza a szolgáltatást és a webkiszolgáló podot mindegyikhez. Annak érdekében, hogy a Pod kezelt legyen és túlélje a node-ok hibáit, a vezérlő közvetlenül egy Deployment erőforrást hoz létre a nem kezelt Pod helyett. [8]

### 3.2.9. Az operátor bemutatása

Az operátor egy Kubernetes-alkalmazás csomagolására, telepítésére és kezelésére szolgáló módszer. Egy Kubernetes-alkalmazást egyszerre telepítünk a Kubernetesre és kezelünk a Kubernetes API (alkalmazásprogramozási felület) és a kubectl tool (segédeszköz) segítségével.

A Kubernetes-operátor egy alkalmazásspecifikus vezérlő, amely a Kubernetes API funkcionalitását kiterjeszti, hogy komplex alkalmazások példányait hozza létre, konfigurálja és kezelje a Kubernetes-felhasználó nevében. A Kubernetes erőforrás és vezérlő alap-



**3.6. ábra.** A weboldal controllere figyeli a webhely objektumokat és létrehoz egy Deploymentet és egy Service-t [8]

konceptióira épül, de magában foglalja a tartomány- vagy alkalmazásspecifikus tudást az általa kezelt szoftver teljes életciklusának automatizálása érdekében.

A Kubernetesben a control plane vezérője olyan vezérlési feladatokat valósítanak meg, amelyek ismételten összehasonlítják a cluster kívánt állapotát annak tényleges állapotával. Ha a cluster tényleges állapota nem egyezik a kívánt állapottal, akkor a vezérő lépéseket tesz a probléma kijavítására.

Az operátor egy olyan egyéni Kubernetes vezérő, amely egyéni erőforrást (későbbiekben custom resource vagy CR) használ az alkalmazások és komponenseik kezelésére. A magas szintű konfigurációt és beállításokat a felhasználó adja meg egy custom resource-on belül. A Kubernetes operátor a magas szintű utasításokat az operátor logikájába ágyazott legjobb gyakorlatok alapján fordítja le az alacsony szintű műveletekre.

A custom resource Kubernetes API-bővítési mechanizmusa. Egy egyéni erőforrás-definíció (későbbiekben custom resource definition vagy CRD) definiál egy CR-t, és felsorolja az operátor felhasználói számára elérhető összes konfigurációt. A Kubernetes-operátor figyeli a CR típusát, és alkalmazásspecifikus műveleteket hajt végre annak érdekében, hogy az aktuális állapot megfeleljen az erőforrás által kívánt állapotának.

A Kubernetes-operátorok új objektumtípusokat vezetnek be az CRD-k révén. A CRD-t a Kubernetes API ugyanúgy kezelheti, mint a beépített objektumokat, beleértve a kubectl-en keresztüli interakciót és a szerepkör-alapú hozzáférés-szabályozásba (RBAC) való felvételt. Az operátor továbbra is figyelemmel kíséri az alkalmazását, miközben az

fut, és automatikusan biztonsági mentést készíthet az adatokról, helyreállíthatja a hibákat, és frissítheti az alkalmazást.

A Kubernetes-operátor által végzett műveletek szinte bármi lehet: egy összetett alkalmazás skálázása, az alkalmazás verziófrissítése, vagy akár egy speciális hardverrel rendelkező számítási cluster node-jainak kernelmoduljainak kezelése. [9]

Az Istio operátor felépítése látható a(z) 20 oldalon lévő 5.2 ábrán.

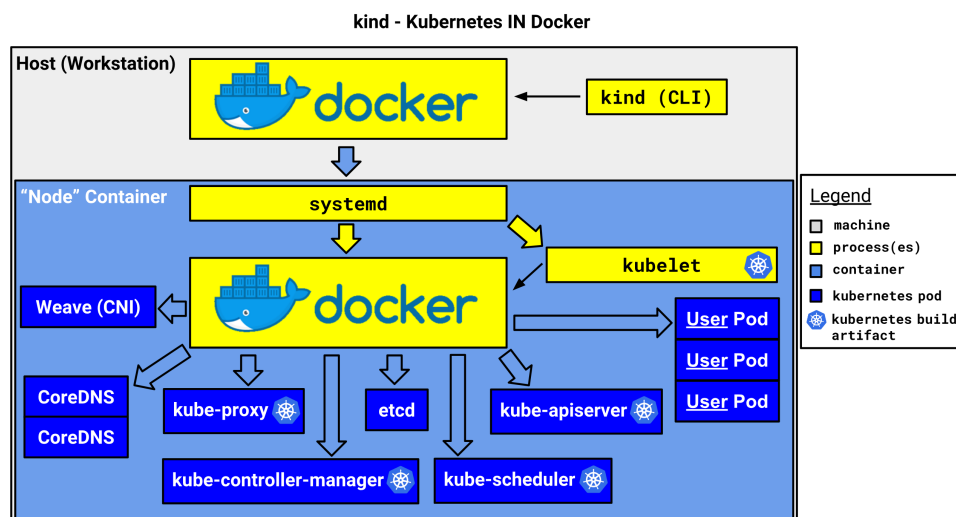
### 3.3. Kubernetes in Docker

A Kubernetes in Docker (későbbiekben kind) egy eszközcsoomag a helyi Kubernetes cluster-ekhez, ahol minden node egy Docker konténer. A kind a Kubernetes tesztelését célozza meg.

A kind funkcionalitásának nagy részét megvalósító go csomagokra, a felhasználóknak szánt parancssorra és egy node base image-re (alap lemezképre) oszlik. A szándék az, hogy a kind csomagjai importálható és újrafelhasználható legyen más eszközök (pl. kubetest) által, míg a CLI gyors módot biztosít e csomagok használatára és hibakeresésére.

Bár nem minden tesztelés végezhető el valódi cluster-ek nélkül a felhőben, de elég lehet ahhoz, hogy ha valami ilyesmit akarunk: [10]

- nagyon olcsó cluster-eket futtatni, amelyeket bármely fejlesztői környezetben replikálhatóak.
- integrálható más eszközökkel
- alaposan dokumentált és karbantartható
- nagyon stabil, kiterjedt hibakezeléssel rendelkezik.



3.7. ábra. KinD felépítése a gyakorlatban [10]

## 4. fejezet

# Helm csomagkezelő

A Helm egy olyan eszköz, amely automatizálja a Kubernetes-alkalmazások létrehozását, csomagolását, konfigurálását és telepítését azáltal, hogy a konfigurációs fájlokat egyetlen újrafelhasználható csomagban egyesíti.

Egy mikroszolgáltatás-architektúrában az alkalmazás növekedésével egyre több mikroszolgáltatást hoz létre, ami egyre nehezebben kezelhetővé teszi azt. A Kubernetes, egy nyílt forráskódú konténer-orchestrációs technológia amely egyszerűsíti a folyamatot azáltal, hogy több mikroszolgáltatást egyetlen telepítésbe csoportosít. A Kubernetes-alkalmazások kezelése azonban a fejlesztési életciklus során saját kihívások sorát hozza magával, beleértve a verziókezelést, az erőforrás-elosztást, a frissítést és a visszaállításokat. A Helm az egyik legkönnyebben elérhető megoldást kínálja erre a problémára, következetesebbé, megismételhetőbbé és megbízhatóbbá téve a telepítéseket.

A konténer egy apró szoftverkomponens, amely egy alkalmazást és függőségeit egyetlen képfájlba csomagolja. A konténerek hordozhatóak a különböző platformok között, ami gyorsabb alkalmazásindítást és egyszerű skálázást eredményez.

A Kubernetes YAML konfigurációs fájlok segítségével telepít. A gyakori frissítéseket tartalmazó összetett telepítések esetében nehéz lehet nyomon követni ezeknek a fájloknak az összes különböző verzióját. A Helm egy praktikus eszköz, amely egyetlen telepítési YAML-fájlt tart fenn a verzióinformációkkal. Ez a fájl lehetővé teszi egy összetett Kubernetes cluster beállítását és kezelését néhány paranccsal. [5]

### 4.1. Helm chart-ok

A Helm chart egy olyan csomag, amely tartalmazza az összes szükséges erőforrást egy alkalmazás Kubernetes cluster-re való telepítéséhez. Ez magában foglalja a telepítésekhez szükséges YAML konfigurációs fájlokat, service-eket, secret-eket és configmap-okat, amelyek meghatározzák az alkalmazás kívánt állapotát.

A Helm chart olyan YAML fájlokat és sablonokat csomagol össze, amelyek segítségével további konfigurációs fájlokat lehet generálni a paraméterezett értékek alapján. Ez lehetővé teszi a konfigurációs fájlok testreszabását a különböző környezetekhez, valamint újrafelhasználható konfigurációk létrehozását. Ezenkívül minden egyes chart-ot egymástól függetlenül lehet verziószámozni és kezelni, ami megkönnyíti az alkalmazás több, különböző konfigurációjú verziójának karbantartását. [5]

## 4.2. Helm működése

A Helm alkalmazáskönyvtár chart-okat használ a Kubernetes-alkalmazások definiálására, létrehozására, telepítésére és frissítésére. Ezek a chart-ok lehetővé teszik a Kubernetes manifesztok kezelését anélkül, hogy a Kubernetes parancssori felületéhez (későbbiekben CLI) hozzá nyúltunk volna vagy bonyolult Kubernetes parancsokat kellene megjegyezni a cluster vezérléséhez.

Tekintsünk egy gyakorlati forgatókönyvet, ahol a Helm hasznos lehet. Tegyük fel, hogy az alkalmazást tíz replikával rendelkező éles (későbbiekben production) környezetben szeretné telepíteni. Ezt megadja az alkalmazás telepítési YAML fájljában és a kubectl paranccsal futtatja a telepítést.

Most futtassa ugyanazt az alkalmazást egy tesztelői (későbbiekben staging) környezetben. Tegyük fel, hogy három replikára van szüksége a staging környezetben, és hogy egy belső alkalmazás buildet fog futtatni a staging környezetben. Ehhez frissítse a replikák számát és a Docker image taget a telepítési YAML fájlban, majd használja azt a staging Kubernetes cluster-ben.

Ahogy az alkalmazás egyre összetettebbé válik, úgy nő a YAML-fájlok száma is. Végül a YAML-fájl konfigurálható mezői is megnőnek. Hamarosan a sok YAML fájl frissítése ugyanazon alkalmazás különböző környezetekben történő telepítéséhez nehezen kezelhetővé fog válni.

A Helm segítségével a mezők a környezettől függően paraméterezhetők. Az előző példában a replikák és Docker-képek statikus értéke helyett egy másik fájlból veheti át ezeket a mezőknek az értékét. Ennek a fájlnak a neve values.yaml. [5]

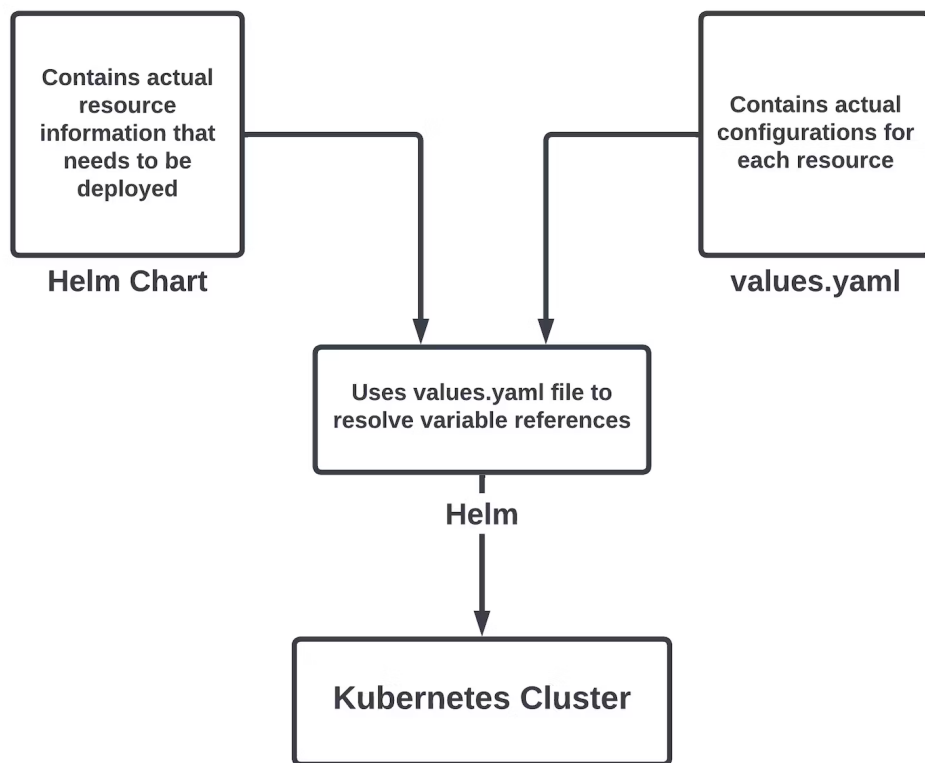
## 4.3. Helm repository

A Helm repository az a hely, ahová Helm chart-okat lehet feltölteni. Létrehozható privát adattár is, ha ez a preferencia. Az Artifact Hub egy globális Helm adattár, amely kereshető chart-okat tartalmaz, amelyeket számos célra telepíthet. Röviden, az Artifact Hub azt teszi a chart-ok számára, amit a Docker Hub a Docker-képek számára. [5]

## 4.4. Mikor érdemes használni

A Helm akkor hasznos, ha a projektet a Kubernetes segítségével komplex, sok mikroszolgáltatást tartalmazó alkalmazásokat futtat. A Helm használatával könnyen automatizálható az alkalmazás telepítése és kezelése, csökkentve a kézi munka mennyiségét, és javítva a rendszer megbízhatóságát és stabilitását. A Helm emellett hozzáférést biztosít az előre konfigurált csomagok kiterjedt tárházához, így könnyen hozzáadható új funkciók az alkalmazáshoz.

Az alkalmazás összetevőinek moduláris táblázatokba szervezésével, amelyeket könnyen telepíthet és frissíthet, a Helm leegyszerűsíti az alkalmazáskomponensek kezelésének folyamatát. Csökkentheti az alkalmazás karbantartásához szükséges kézi munka mennyiségét, és segít elkerülni a hibákat, amelyek az összetett rendszerek kézi kezelése során keletkezhetnek.



**4.1. ábra.** Helm működési folyamatábrája [5]

A Helm támogatja a konténerek telepítését több környezetben is, így a konténerek életciklusa a fejlesztési folyamat során könnyen kezelhető. [5]



## 5. fejezet

# Service Mesh

A modern alkalmazásokat jellemzően mikroszolgáltatások elosztott gyűjteményeként architektúrázzák, amelyek mindegyike valamilyen különálló üzleti funkciót lát el. A szolgáltatásháló egy dedikált infrastrukturális réteg, amelyet hozzáadhat az alkalmazásaihoz. Lehetővé teszi, hogy átlátható módon adjon hozzá olyan képességeket, mint a megfigyelhetőség, a forgalomirányítás és a biztonság, anélkül, hogy azokat a saját kódjához kellene hozzáadnia. A "szolgáltatásháló" kifejezés leírja mind a minta megvalósításához használt szoftver típusát, mind pedig a szoftver használata során létrehozott biztonsági vagy hálózati tartományt.

Ahogy az elosztott szolgáltatások telepítése, például egy Kubernetes-alapú rendszerben, egyre nagyobb és összetettebb lesz, egyre nehezebbé válhat a megértése és kezelése. Követelményei közé tartozhat a felderítés, a terheléelosztás, a hibaelhárítás, a metrikák és a felügyelet. A szolgáltatásháló gyakran olyan összetettebb üzemeltetési követelményekkel is foglalkozik, mint az A/B tesztelés, a kanáris telepítések, a forgalomkorlátozás, a hozzáférés-szabályozás, a titkosítás és a végponttól végpontig tartó hitelesítés.

Az elosztott alkalmazást a szolgáltatások közötti kommunikáció teszi lehetővé. Ennek a kommunikációnak az útválasztása mind az alkalmazás cluster-én belül, mind pedig azok között egyre összetettebbé válik a szolgáltatások számának növekedésével. Az Istio segít csökkenteni ezt a komplexitást, miközben megkönnyíti a fejlesztőcsapatok dolgát.

### 5.1. Istio

Az Istio a fejlesztők és az üzemeltetők előtt álló kihívásokat kezeli egy elosztott vagy mikroszolgáltatási architektúrával. Akár a jelenlegi rendszerből építkeznek, akár a nulláról vagy a meglévő alkalmazások felhőalapúvá történő migrálásáról, az Istio tud segíteni.

Az Istio egy nyílt forráskódú szolgáltatásháló, amely átláthatóan rétegződik a meglévő elosztott alkalmazásokra. Az Istio fejlett funkciói egységes és hatékonyabb módot biztosítanak a szolgáltatások biztosítására, összekapcsolására és felügyeletére. Az Istio az út a terheléelosztáshoz, a szolgáltatások közötti hitelesítéshez és a felügyelethez - kevés vagy semmilyen szolgáltatási kódváltoztatással. Kiforrott control plane-je létfontosságú funkciókat biztosít, többek között:

- Biztonságos szolgáltatások közötti kommunikáció egy cluster-ben TLS-titkosítással, erős identitásalapú hitelesítés és engedélyezés.

- Automatikus terheléelosztás a HTTP, gRPC, WebSocket és TCP forgalomhoz.
- A forgalom viselkedésének finomra szabott vezérlése gazdag útválasztási szabályokkal, újbóli próbálkozásokkal, meghibásodásokkal és hibainjektálással.
- Beilleszthető házirend-réteg és konfigurációs API, amely támogatja a hozzáférés-szabályozást, a sebességkorlátozást és a kvótákat.
- Automatikus mérőszámok, naplók és nyomkövetés a cluster-en belüli összes forgalomra vonatkozóan, beleértve a cluster be- és kilépési pontjait is.

Az Istio-t bővíthetőségre tervezték, és a telepítési igények széles skáláját képes kezelni. Az Istio control plane a Kubernetes-en fut, és az adott cluster-ben telepített alkalmazásokat hozzáadhatja a hálójához (későbbiekben mesh), a mesh-t kiterjesztheti más cluster-ekre, vagy akár VM-eket és más a Kubernetes-en kívül futó végpontokat is csatlakoztathat. A közreműködők, partnerek, integrációk és disztribútorok nagy ökoszisztémája az Istio-t a legkülönbébb helyzetekre bővíti és hasznosítja. Az Istio-t telepítheti saját maga, vagy számos gyártó rendelkezik olyan termékekkel, amelyek integrálják az Istio-t és kezelik azt az Ön számára.

Az Istio két komponensből áll: az adatsíkból (későbbiekben data plane) és a vezérlési síkból (későbbiekben control plane). Az data plane a szolgáltatások közötti kommunikációt jelenti. Service mesh nélkül a hálózat nem érti az átküldött forgalmat, és nem tud döntéseket hozni az alapján, hogy milyen típusú forgalomról van szó, vagy hogy kitől vagy kihez érkezik. A service mesh egy proxy segítségével elfogja az összes hálózati forgalmat, és az Ön által beállított konfiguráció alapján lehetővé teszi az alkalmazásfüggő funkciók széles körét. Az Envoy proxy minden olyan szolgáltatással együtt kerül telepítésre, amelyet a cluster-ben elindít, vagy a VM-eken futó szolgáltatások mellett fut. A control plane átveszi a kívánt konfigurációt és a szolgáltatásokról alkotott nézetét, és dinamikusan programozza a proxy-kiszolgálókat, frissítve azokat, amint a szabályok vagy a környezet változik. [4]

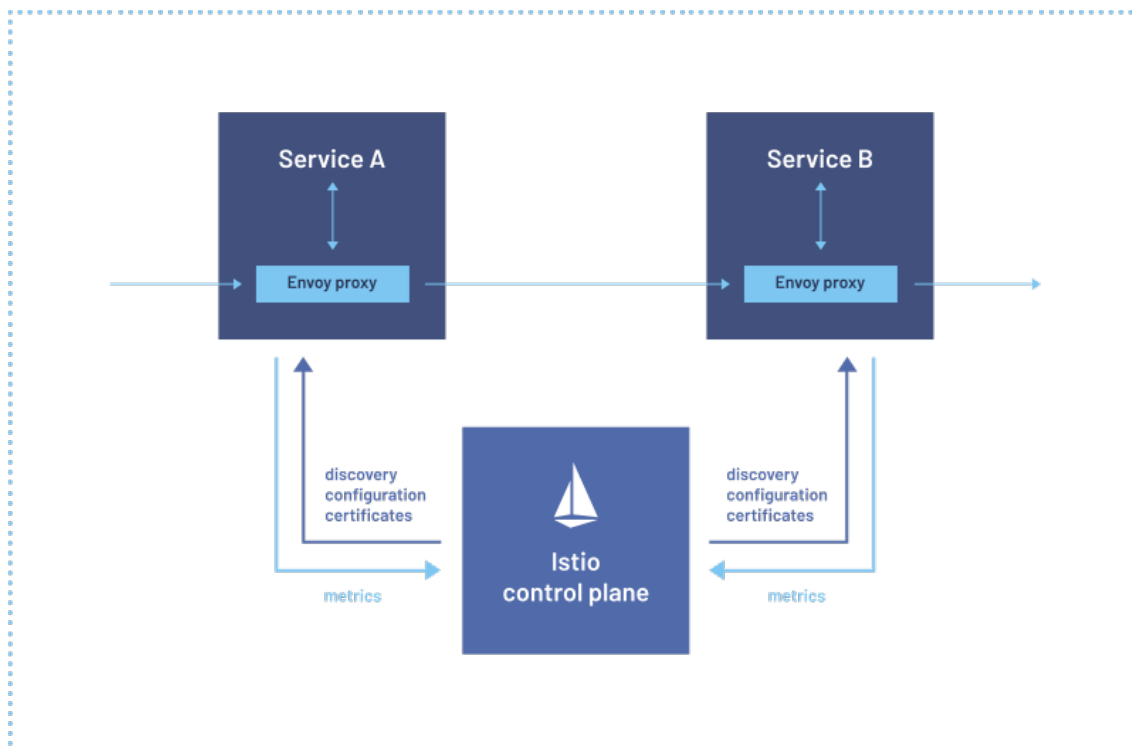
## 5.2. Istio operátor

Az Istio-operator egy olyan egyéni erőforrást definiál, amely leírja az Istio telepítés kívánt állapotát. Tartalmazza az összes szükséges konfigurációs értéket, és ha ezek közül egy vagy több megváltozik, az operátor automatikusan összehangolja a komponensek állapotát az új kívánt állapotnak megfelelően.

Az Istio-operátor lehetővé teszi a távoli konfigurációt futtató Kubernetes control plane-ek számára, hogy egyetlen Istio control plane-hez csatlakozzanak. Az operátor kezeli az Istio komponensek távoli cluster-ekre történő telepítését, és olyan szinkronizációs mechanizmust ad, amely hozzáférést biztosít az Istio központi komponenseihez a távoli cluster-ekről. Automatikusan nyomon követi azokat a névtereket, ahol az automatikus sidecar-injection engedélyezve van, így központilag kezelheti a funkciókat ahelyett, hogy egyesével kellene a névtereket felcímkézni. [1]

## 5.3. Cisco Istio operátor

Az istio csapat által fejlesztett istio operátor nem bizonyult elegendőnek a Calisti Service Mesh Manager (SMM) termékének fejlesztése közben, így fork-olták és kibővítették



**5.1. ábra.** Az Istio használata utáni architektúra [4]

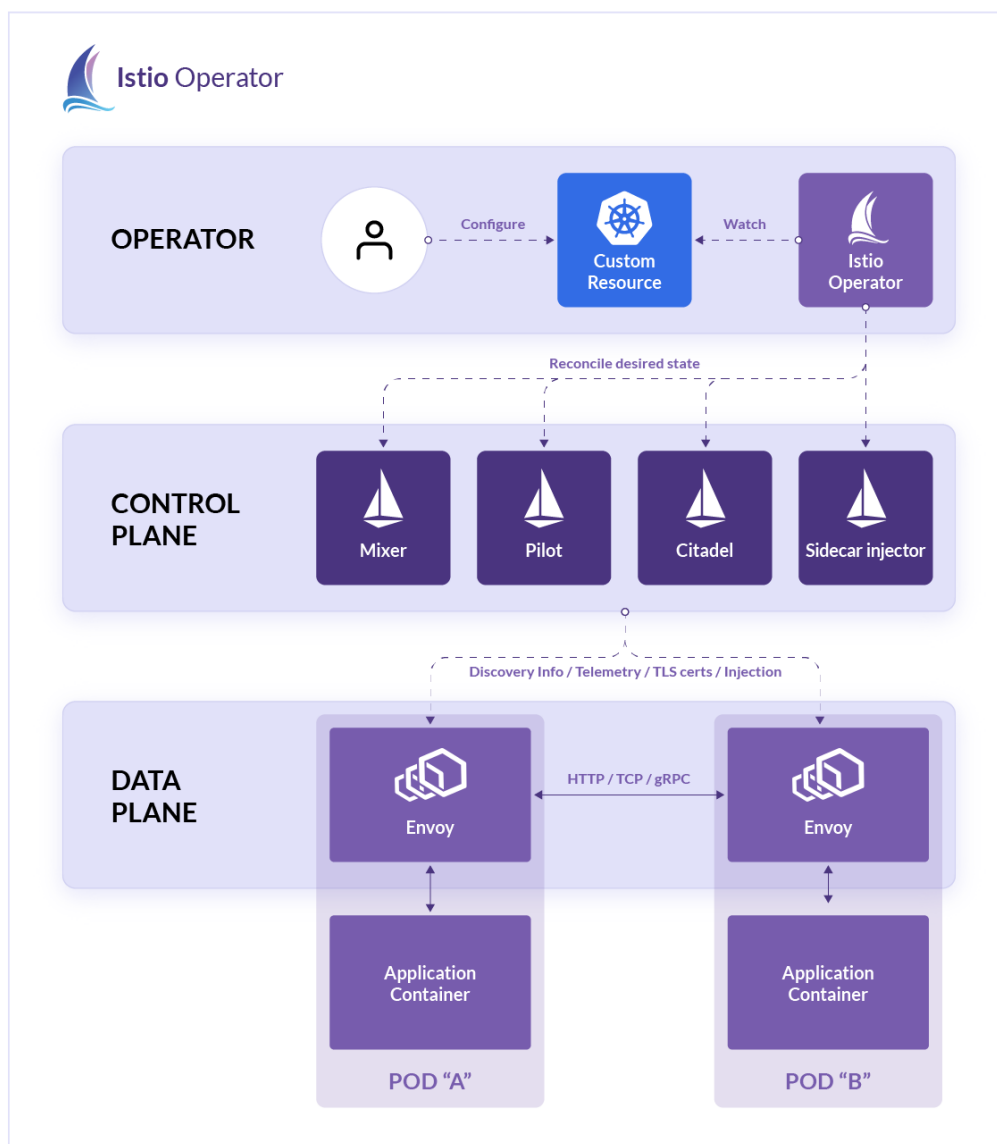
az istio operátort. Az SMM segít telepíteni, frissíteni, és menedzselni az service mesh-t és további egyéb eszközöket is nyújt:

- Dashboard
- Automatikus multi cluster támogatás
- Topológiai gráf
- mTLS menedzsment
- Virtuálisgép integráció
- stb...

## 5.4. Multi cluster

A több cluster-es (későbbiekben multi-cluster) Kubernetes egy olyan telepítési módszer, amely két vagy több cluster-ből áll. Ez a telepítési módszer rendkívül rugalmas. Lehetnek cluster-ek ugyanazon a fizikai gépen vagy ugyanazon adatközpontban lévő különböző gépeken. Többfelhős környezetet is létrehozhat különböző felhőkben akár különböző országokban élő cluster-ekkel is.

A multi-cluster-es architektúra legegyszerűbb formájában nem tűnik olyan bonyolultnak és ijesztőnek. A valóságban azonban többféleképpen lehet kialakítani egy multi-cluster-es Kubernetes-architektúrát. Az architekturális választás a két fő architekturális megközelítésen alapul - szegmentáció vagy replikáció. [11]



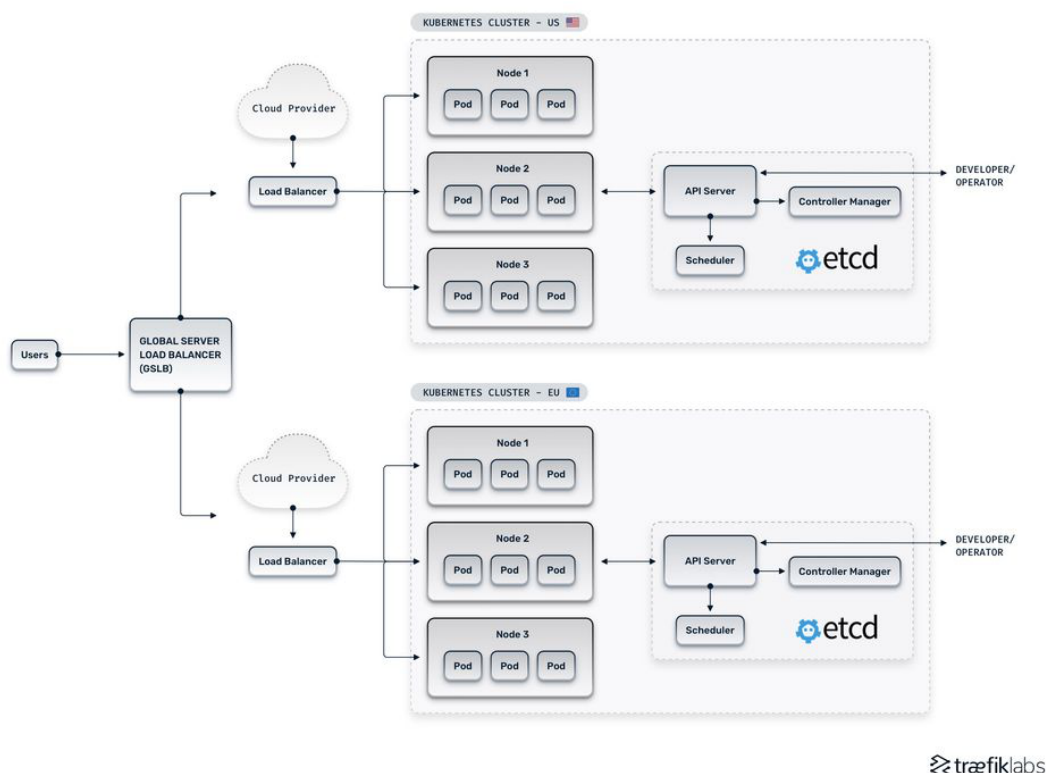
5.2. ábra. Az Istio operátor vizualizálása [1]

#### 5.4.1. Több klaszteres architektúra tervek

A szegmentációs megközelítéssel az alkalmazást független komponensekre tagolják, amelyeket általában Kubernetes-szolgáltatásokként ábrázolnak, és üzemeltetési követelményeknek megfelelően különböző Kubernetes-klaszterekhez rendelnek hozzá. A replikációs megközelítéssel egy Kubernetes cluster pontos másolatai több adatközpontban, különböző helyszíneken vannak elhelyezve. Ez redundanciát biztosít, így az alkalmazás elérhető marad abban az esetben is, ha az egyik cluster elérhetatlenné válik. [11]

#### 5.4.2. Kubernetes-központú vs. hálózat-központú konfiguráció

A multi-cluster-es Kubernetes környezetek kezelésének két alapvető módszere van - Kubernetes-központú és a hálózat-központú. Ha a Kubernetes-központú módszert hasz-



5.3. ábra. Az Istio használata utáni architektúra [11]

náljuk a multi-cluster-es környezet kezelésére, akkor különböző alkalmazások futnak különböző Kubernetes cluster-eken, mégis mindegyiket egy központi helyről kezelik.

A hálózat-központú módszerrel az alkalmazás pontos másolatát több, különböző helyen lévő cluster-eken telepíti. Bár ezek a cluster-ek különálló egységként viselkednek, az összes Kubernetes-cluster között egy hálózaton keresztül fenntartja a kommunikációt. [11]

#### 5.4.3. A több klaszteres architektúra előnyei

Minden architektúra központi eleme az egyes cluster-ek függetlensége, és ebben rejlik a multi-cluster-es Kubernetes-architektúra nagy előnye. A multi-cluster-es Kubernetes valós előnyeinek köre azonban ezen túlmutat. [11]

#### Rugalmasság

A multi-cluster jelentős rugalmasságot és irányítást biztosít a tervezés és a konfiguráció megválasztása terén. Például a Kubernetes különböző verzióit vagy disztribúcióit használhatja a különböző cluster-ekben, hogy megfeleljen az alkalmazások különböző részeire vonatkozó különleges igényeknek vagy követelményeknek. A több multi-cluster-es architektúrával a Kubernetes különböző vagy újabb verzióit is tesztelheti elszigetelt cluster-ekben, mielőtt frissítené a produktív környezetét, így minimálisra csökkentheti a hibás változtatások és a leállások kockázatát. [11]

## Elérhetőség, skálázhatóság és erőforrás-kihasználás

A multi-cluster-es telepítések egyik legnagyobb előnye a Kubernetes teljesítményének javítása, különösen a rendelkezésre állás és a skálázhatóság tekintetében. A multi-cluster-es architektúrával a munkaterheléseket a cluster-ek között mozgathatja. Egy cluster-t használhat biztonsági mentésként, és a cluster-ek különböző adatközpontokban, helyszíneken és felhőkben is szétszórhatja. Ezzel pedig minimálisra csökkentheti annak kockázatát, hogy egyetlen cluster meghibásodása miatt az egész környezete leálljon. A cluster-ek különböző helyszíneken történő telepítésével csökkenti az alkalmazást tároló cluster és a végfelhasználó közötti fizikai távolságot, növelve ezzel a teljesítményt és minimalizálva a késleltetést. A különböző cluster-ek között elosztott munkaterhelések nagyobb skálázhatóságot is eredményeznek. A multi-cluster-es Kubernetes lehetővé teszi, hogy a különböző cluster-eket az adott terhelésüknek megfelelően fel- és lefelé skálázza - ez optimalizálja az erőforrás kihasználtságot az erőforrások különböző teljesítménykövetelményekhez való igazításával. [11]

## Feladat izoláció

Az elszigeteltség bizonyos szintje még egy cluster-es architektúrában is elérhető a névterek használatával. Az egy cluster-es forgatókönyvben azonban a Kubernetes biztonsági rendszerének sajátosságai miatt a környezetek nem tökéletesen elszigeteltek egymástól. A cluster-en belüli elszigetelés ezen módszerének használata esetén az azonos hardveren osztozó alkalmazások hatással lehetnek egymásra - ez a "zajos szomszéd" problémaként is ismert. A munkafeladatok több Kubernetes cluster-en keresztüli futtatásával magas szintű elszigeteltséget eredményezhet. Az esetleges cluster hibák vagy konfigurációs változások csak az adott cluster-t érintik. A munkaterhelés-elszigetelés kulcsfontosságú a fejlesztőcsapatok számára, mivel segít nekik a problémák egyszerű elkülönítésében és diagnosztizálásában, az új funkciók és konfigurációk tesztelésében, az alkalmazások fejlesztésében és frissítésében, és mindezt úgy, hogy közben a produktív környezet biztonságos és elérhető marad. [11]

## Biztonság és megfelelés

A multi-cluster-es architektúra által biztosított magas szintű terheléskülönítéssel minimálisra csökkentheti annak kockázatát, hogy az egymástól független alkalmazások nem kívánt módon lépjenek kölcsönhatásba egymással. A szigorú elszigeteltség csökkenti annak kockázatát is, hogy az egyik cluster-ben felmerülő biztonsági probléma az egész környezetet befolyásolja. Ez bizonyos mértékig egyetlen Kubernetes cluster-en belül is megvalósítható, például pod biztonsági házirendek használatával. De semmi sem hasonlítható a multi-cluster-es Kubernetes által biztosított magas szintű teherelkülönítéshez. A multi-cluster-es telepítések megkönnyítik a különböző országok eltérő szabályainak és irányelveinek való megfelelést is. Erre remek példa az általános adatvédelmi rendelet (későbbiekben GDPR) az EU-ban. A GDPR előírja, hogy a felhasználói adatok ne hagyják el az EU-t. Ha világszerte vannak felhasználói és egyetlen cluster-ben futtatja az alkalmazást, akkor szinte lehetetlen lenne megfelelni ezeknek a követelményeknek. A multi-cluster-es telepítésekkel egy cluster-t telepíthet az EU-ban, hogy megfeleljen ezeknek az igényeknek és a helyi előírásoknak, miközben máshol más cluster-eket futtathat. [11]

#### 5.4.4. A több klaszteres Kubernetes kihívásai

Nagyon kevés jó dolog van az életben, amihez nem kell akadályokat leküzdeni, és a Kubernetes biztosan nem tartozik ezek közé. A végtelen lehetőségek és a multi-cluster-es telepítésekkel járó fontos előnyök ellenére van néhány kihívás. A Kubernetes nagyon összetett, a multi-cluster-es Kubernetes pedig még összetettebb. A hozzáadott komplexitás számos formában jelentkezhet. [11]

##### Konfiguráció

Az architektúra kialakításától függően a környezet különböző cluster-i különböző konfigurációkat igényelnek az egyes cluster-ek sajátos igényeinek kielégítésére. Ez aprólékos munka, és ezt nem lehet megkerülni. Ugyanennek a problémának egy másik szintje a tűzfal konfigurációja. Az egy cluster-es architektúrában egyetlen API-kiszolgálót lehet elérni egy adott címen.

Multi-cluster-es telepítés esetén több API-kiszolgálót ér el de gondoskodnia kell arról, hogy a tűzfal lehetővé tegye a kapcsolódó cluster-ekben lévő API-kiszolgálók elérését. Ennek enyhítésére automatizálási szkripteket használhat, amelyek sajnos szintén növelik a bonyolultsági szintet. [11]

##### Biztonság

multi-cluster-es Kubernetesben a munkaterhelés elkülönítése segíti a jobb biztonsági helyzetet, de ennek eléréséhez figyelembe kell vennie az egyes cluster-ekre vonatkozó eltérő szabályokat és biztonsági tanúsítványokat. Ahhoz, hogy megfeleljen a különböző helyi szabályozásoknak, ezeket a biztonsági tanúsítványokat különböző adatközpontokban és cluster-eken kell kezelnie. [11]

##### Telepítés

Természetesen a multi-cluster-es Kubernetes architektúrában a telepítések egyre összetettebbé válnak. Ez különösen igaz a GitOps megközelítés használata esetén. A GitOps-ban a forráskódkezelő szolgáltatása az összes telepítési tevékenység egyetlen forrása. Ez azt jelenti, hogy a GitOps folyamatainak jól szervezettnek, biztonságosnak és intelligensnek kell lenniük. A GitOps segíthet automatizálni a cluster-ek konfigurálását, képessé tesz a Kubernetes- cluster-ek életciklusának irányítására és a skálázott telepítésre. [11]

##### Költség

Egy másik kihívás, amelyet a multi-cluster-es Kubernetes architektúra bevezetésekor figyelembe kell venni, a költségek potenciálisan jelentős növekedése. A multi-cluster több node-ot jelent, ami viszont az erőforrás-felhasználás és végső soron a költségek növekedését eredményezi. Egy másik fontos szempont, amit szem előtt kell tartani, a megnövekedett számú terheléelosztók, ingresszek, valamint naplózási és felügyeleti erőforrások szükségessége. [11]

## 6. fejezet

# Felhőszolgáltatások

A felhőalapú számítástechnika (későbbiekben cloud-computing) a számítástechnikai erőforrások - beleértve a tárolást, a feldolgozási teljesítményt, az adatbázisokat, a hálózat-építést, az analitikát, a mesterséges intelligenciát és a szoftveralkalmazásokat - interneten (a felhőben) történő rendelkezésre bocsátása. Ezen erőforrások kiszervezésével a vállalatok hozzáférhetnek a számítási eszközökhöz, amelyekre szükségük van, amikor szükségük van rájuk, anélkül, hogy fizikai, helyben lévő IT-infrastruktúrát kellene vásárolniuk és fenntartaniuk. Ez rugalmas erőforrásokat, gyorsabb innovációt és méretgazdaságosságot biztosít. Sok vállalat számára a felhőre való áttérés közvetlenül kapcsolódik az adatok és az informatika modernizálásához. [7]

### 6.1. A felhőalapú számítástechnika jellemzői

A cloud-computing előtt a szervezetek helyben vásároltak és tartottak fenn informatikai infrastruktúrát. Bár a felhőre való kezdeti áttérés nagy részét a költségmegtakarítás vezérelte, sok szervezet úgy találja, hogy a nyilvános, a privát vagy a hibrid felhőinfrastruktúra számos előnyt kínál.

Az agilis és DevOps-csapatok számára a felhőalapú számítástechnika lehetőséget nyújt a fejlesztési folyamat egyszerűsítésére és felgyorsítására.

Az alábbiakban felsoroljuk a felhőalapú számítástechnikát meghatározó jellemzőket. [7]

- Igény szerinti önkiszolgálás
- Széles körű hálózati hozzáférés
- Erőforrás-összevonás
- Gyors rugalmasság
- Mérhető szolgáltatás



## 6.2. A felhőalapú telepítések típusai

A felhőalapú telepítéseknek három különböző típusa létezik. Mindegyiknek megvannak az egyedi előnyei, és a szervezetek gyakran többféle megoldás használatából is profitálnak. [7]

### 6.2.1. Publikus felhő

A publikus felhők számítástechnikai erőforrásokat - szervereket, tárolókat, alkalmazásokat stb. – nyújtanak az interneten keresztül egy felhőszolgáltatótól, például az AWS-től és a Microsoft Azure-től. A felhőszolgáltatók birtokolják és üzemeltetik az összes hardvert, szoftvert és egyéb támogató infrastruktúrát. [7]

### 6.2.2. Privát felhő

A privát felhő olyan számítási erőforrások, amelyek kizárólag egy szervezet számára vannak fenntartva. Fizikailag elhelyezkedhet a szervezet helyszíni adatközpontjában, vagy egy felhőszolgáltató által üzemeltetett tárhelyen. A privát felhő a nyilvános felhőknél magasabb szintű biztonságot és adatvédelmet nyújt azáltal, hogy dedikált erőforrásokat kínál a vállalatok számára.

A privát felhő ügyfelei megkapják a nyilvános felhő elsődleges előnyeit, beleértve az önkiszolgálást, a skálázhatóságot és a rugalmasságot, de további előnyökkel, a további ellenőrzéssel és testreszabhatósággal. Ráadásul a privát felhők magasabb szintű biztonsággal és adatvédelemmel rendelkezhetnek, mivel a nyilvános forgalom számára nem hozzáférhető magánhálózatokon vannak elhelyezve. [7]

### 6.2.3. Hibrid felhő

A hibrid felhők a privát és a nyilvános felhők (például a Red Hat által támogatott IBM Hybrid Cloud) kombinációja, amelyeket olyan technológiával kapcsolnak össze, amely lehetővé teszi az adatok és az alkalmazások együttes működését. Az érzékeny szolgáltatásokat és alkalmazásokat a biztonságos privát felhőben lehet tartani, míg a nyilvánosan elérhető webkiszolgálók és az ügyfelekkel szembenéző végpontok a nyilvános felhőben élhetnek. A legtöbb népszerű harmadik féltől származó felhőszolgáltató hibrid felhőmodellt kínál, amely lehetővé teszi a felhasználók számára, hogy igényeiknek megfelelően kombinálják a privát és a nyilvános felhőket. Ez nagyobb rugalmasságot biztosít a vállalkozások számára az alkalmazásuk egyedi infrastrukturális követelményeinek telepítésében. [7]

## 6.3. Felhőalapú számítástechnikai szolgáltatások

A felhőalapú számítástechnika dinamikus tulajdonságai új, magasabb szintű szolgáltatások alapját képezik. Ezek a szolgáltatások nemcsak kiegészíthetik, hanem gyakran szükséges szolgáltatásokat is nyújthatnak az agilis és DevOps-csapatok számára. [7]

### 6.3.1. Infrastruktúra mint szolgáltatás

Az infrastruktúra mint szolgáltatás (későbbiekben Infrastructure as a Service vagy IaaS) egy olyan alapvető felhőszolgáltatási réteg, amely lehetővé teszi a szervezetek számára, hogy informatikai infrastruktúrát - szervereket, tárolókat, hálózatokat, operációs rendszereket - béreljenek egy felhőszolgáltatótól. Az IaaS lehetővé teszi a felhasználók számára, hogy a nyers fizikai szerverraktárakból lefoglalják és rendelkezésre bocsássák a szükséges erőforrásokat. Ezenkívül az IaaS lehetővé teszi a felhasználók számára, hogy előre konfigurált gépeket foglaljanak speciális feladatokra, például terheléelosztókra, adatbázisokra, e-mail szerverekre, elosztott várólistákra.

A DevOps-csapatok az IaaS-t olyan mögöttes platformként használhatják, amelyre egy DevOps-eszközláncot építhetnek, amely magában foglalhatja különböző harmadik féltől származó eszközök használatát. [7]

### 6.3.2. Platform mint szolgáltatás

A platform mint szolgáltatás (későbbiekben Platform as a Service vagy PaaS) egy IaaS-re épülő felhőinfrastruktúra, amely erőforrásokat biztosít felhasználói szintű eszközök és alkalmazások létrehozásához. Az alapul szolgáló infrastruktúrát, beleértve a számítási, hálózati és tárolási erőforrásokat, valamint a fejlesztőeszközöket, az adatbázis-kezelő rendszereket és a köztes szoftvereket.

A PaaS kihasználja az IaaS-t, hogy automatikusan kiossza a nyelvi alapú tech stack működtetéséhez szükséges erőforrásokat. Népszerű nyelvi tech stackek a Ruby On Rails, a Java Spring MVC, a MEAN és a JAM stackek. A PaaS-ügyfelek ezután egyszerűen feltölthetik az alkalmazáskódjukat, amely automatikusan települ a PaaS infrastruktúrájára. Ez egy újszerű és hatékony munkafolyamat, amely lehetővé teszi a csapatok számára, hogy teljes mértékben az adott üzleti alkalmazás kódjára koncentráljanak, és ne aggódjanak a tárhely és az infrastruktúra problémái miatt. A PaaS automatikusan kezeli az infrastruktúra skálázását és felügyeletét, hogy a megfigyelt forgalmi terhelésnek megfelelően növelje vagy csökkentse az erőforrásokat. [7]

### 6.3.3. Szoftver mint szolgáltatás

A szoftver mint szolgáltatás (későbbiekben Software as a Service vagy SaaS) az interneten keresztül, igény szerint és jellemzően előfizetés alapján nyújt szoftveralkalmazásokat. A felhőszolgáltatók üzemeltetik és kezelik az alkalmazást, és szükség szerint gondoskodnak a szoftverfrissítésekről és a biztonsági javításokról. A SaaS-re példák a CRM-rendszerek, webmail-alkalmazások, termelékenységi eszközök, mint a Jira és a Confluence, elemzőeszközök, felügyeleti eszközök, csevegőalkalmazások és még sok más. [7]

### 6.3.4. Funkció mint szolgáltatás

A szolgáltatásként nyújtott funkció (Function as a Service vagy FaaS) olyan felhőalapú számítástechnikai szolgáltatás, amely olyan platformot kínál, amelyen az ügyfelek alkalmazásokat fejleszthetnek, futtathatnak és kezelhetnek. Ezáltal a fejlesztőknek nem kell kiépíteniük és fenntartaniuk az alkalmazás fejlesztéséhez és elindításához szükséges infrastruktúrát. A felhőszolgáltatók felhőalapú erőforrásokat kínálnak, végrehajtanak egy

kódblokkot, visszaadják az eredményt, majd megsemmisítik a felhasznált erőforrásokat. [7]

## 6.4. A felhőalapú számítástechnika előnyei

### Csökkentett költség

A felhőalapú erőforrásokat használó csapatoknak nem kell saját hardvereszközöket vásárolniuk. A hardverköltségeken túl a felhőszolgáltatók mindent megtesznek a hardverhasználat maximalizálása és optimalizálása érdekében. Ezáltal a hardver és a számítási erőforrások árucikké válnak, és a felhőszolgáltatók a legalacsonyabb végeredményért versenyeznek. [7]

### Nagyobb skálázhatóság

Mivel a felhőalapú számítástechnika alapértelmezés szerint rugalmas, a szervezetek igény szerint méretezhetik az erőforrásokat. A felhőalapú számítástechnika automatikus skálázási funkciókat tesz lehetővé a csapatok számára. A felhőalkalmazások a forgalom kiugrásaira reagálva automatikusan képesek csökkenteni és növelni infrastrukturális erőforrásaikat. [7]

### Nagyobb teljesítmény

A felhőalapú számítástechnika a legújabb és legnagyobb számítási erőforrásokat kínálja. A felhasználók hozzáférhetnek a legújabb, extrém, többmagos CPU-kkal felszerelt, nehéz párhuzamos feldolgozási feladatokra tervezett gépekhez. Emellett a nagy felhőszolgáltatók élvonalbeli GPU és TPU hardveres gépeket kínálnak az intenzív grafikai, mátrix- és mesterséges intelligencia feldolgozási feladatokhoz. Ezek a felhőszolgáltatók folyamatosan frissítik a legújabb processzortechnológiát.

A nagyobb felhőszolgáltatók globálisan elosztott hardverhelyekkel rendelkeznek, amelyek a fizikai kapcsolódási helytől függően nagy teljesítményű kapcsolatokat biztosítanak. Emellett a felhőszolgáltatók globális tartalomszolgáltató hálózatokat kínálnak, amelyek a felhasználói kéréseket és tartalmakat hely szerint gyorsítótárba helyezik. [7]

### Nagyobb végrehajtási sebesség

A felhőinfrastruktúrákat használó csapatok gyorsabban tudnak végrehajtani és értéket nyújtani ügyfeleiknek. Az agilis szoftvercsapatok kihasználhatják a felhőinfrastruktúrát, hogy gyorsan új virtuális gépeket indíthassanak az egyedi ötletek kikísérletezéséhez és validálásához, valamint automatizálhatják a pipeline tesztelési és telepítési fázisait. [7]

### Nagyobb biztonság

A privát felhő tárhely elszigetelt tűzfalas infrastruktúrát kínál, amely javítja a biztonságot. Emellett a felhőszolgáltatók számos biztonsági mechanizmust és technológiát kínálnak a biztonságos alkalmazások kialakításához. A felhasználók hozzáféréseinek ellenőrzése

fontos biztonsági szempont, és a legtöbb felhőszolgáltató kínál eszközöket a felhasználók hozzáféréseinek részletes korlátozására. [7]

## **Folyamatos integráció és szállítás**

A folyamatos integráció és folyamatos szállítás (későbbiekben CI/CD) a DevOps-gyakorlók egyik legfontosabb gyakorlata, amely segít növelni a csapat sebességét és csökkenteni a piacra jutási időt. A felhőalapú CI/CD, például a Bitbucket Pipelines lehetővé teszi a csapatok számára, hogy automatikusan építsenek, teszteljenek és telepítsenek kódot anélkül, hogy a CI-infrastruktúra kezelésével vagy karbantartásával kellene foglalkozniuk. A Bitbucket Pipelines a Docker konténerekre támaszkodik a kiadási csővezeték elszigeteltségének és reprodukálhatóságának biztosítása érdekében. A csapatok hasonló parancsokat futtathatnak, mint egy helyi gépen, de minden egyes buildhez friss és reprodukálható beállítás minden előnyével. [7]

## **Átfogó felügyelet és incidenskezelés**

A felhőalapú telepítések lehetővé teszik a csapatok számára, hogy eszközeiket a végponttól a végpontig összekapcsolják, megkönnyítve a csővezeték minden részének nyomon követését. Az átfogó felügyelet egy másik kulcsfontosságú képesség a DevOps-ot gyakorló szervezetek számára, mivel lehetővé teszi a problémák és incidensek gyorsabb kezelését. A felhőszolgáltatók megosztják a rendszer állapotára vonatkozó mérőszámokat, beleértve az alkalmazás és a szerver CPU-ját, memóriáját, a kérések számát, a hibaarányt, az átlagos válaszidőt stb. Például a terhelés figyelése sok virtuális gépen keresztül azt jelenti, hogy a csapatok több kapacitást tudnak hozzáadni, ha megnövekszik az igény, vagy a csapatok automatizálhatják a skálázást (fel/le) ezen mérőszámok alapján, hogy csökkentsék az emberi beavatkozást és a költségeket. [7]

## 7. fejezet

# Go programozási nyelv

### 7.1. A programozási nyelv

Minden programozási nyelv tükrözi alkotóinak programozási filozófiáját, amely gyakran jelentős mértékben reagál a korábbi nyelvek vélt hiányosságaira. A Go projekt a Google számos szoftverrendszerével kapcsolatos csalódottságból született, amelyek a komplexitás robbanásszerű növekedésétől szenvedtek. (Ez a probléma korántsem csak a Google-nál jelentkezik.) Ahogy Rob Pike fogalmazott, a "komplexitás multiplikatív": egy probléma megoldása a rendszer egy részének bonyolultabbá tételével lassan, de biztosan növeli a többi rész komplexitását. A funkciók, opciók és konfigurációk hozzáadására, valamint a kód gyors szállítására irányuló állandó nyomás miatt könnyű elhanyagolni az egyszerűséget, pedig hosszú távon az egyszerűség a jó szoftver kulcsa. Az egyszerűség több munkát igényel a projekt elején, hogy egy ötletet a lényegére redukáljunk, és több fegyelmet a projekt teljes élettartama alatt, hogy megkülönböztessük a jó változtatásokat a rossz vagy káros változtatásoktól. Elegendő erőfeszítéssel egy jó változtatás a Fred Brooks által "konceptcionális integritásnak" nevezett terv veszélyeztetése nélkül is megvalósítható, de egy rossz változtatás nem, és egy ártalmas változtatás az egyszerűséget felszínes rokonára, a kényelemre cseréli. Csak a tervezés egyszerűsége révén maradhat egy rendszer stabil, biztonságos és koherens, miközben növekszik. A Go projekt magában foglalja magát a nyelvet, az eszközöket és a szabványos könyvtárakat, és nem utolsósorban a radikális egyszerűség kulturális programját.

Mivel a Go egy nemrégiben kifejlesztett magas szintű nyelv, a visszatekintés előnyeit élvezheti, és az alapok jól sikerültek: van szemétgyűjtés, csomagrendszer, első osztályú függvények, lexikális hatókör, rendszerhívó interfész és megváltoztathatatlan karakterláncok, amelyekben a szöveg általában UTF-8 kódolású. Viszonylag kevés funkcióval rendelkezik, és nem valószínű, hogy továbbiakkal bővül. Például nincsenek implicit numerikus konverziók, nincsenek konstruktorok vagy destruktorok, nincs operátor-túlterhelés, nincsenek alapértelmezett paraméterértékek, nincs öröklés, nincs generika, nincsenek kivételek, nincsenek makrók, nincsenek függvényjegyzetek és nincs szál-lokális tárolás. A nyelv kiforrott és stabil, és garantálja a visszafelé kompatibilitást: a régebbi Go programok lefordíthatók és futtathatók a fordítóprogramok és szabványos könyvtárak újabb verzióival.

A Go elégséges típusrendszerrel rendelkezik ahhoz, hogy elkerülje a legtöbb gondatlan hibát, amely a dinamikus nyelvek programozóit sújtja, de egyszerűbb típusrendszerrel rendelkezik, mint a hasonló tipizált nyelvek. Ez a megközelítés néha a "tipizálatlan" programozás elszigetelt zugait eredményezheti a típusok tágabb keretén belül, és a Go progra-

mozók nem mennek el olyan messzire, mint a C++ vagy a Haskell programozók, hogy a biztonsági tulajdonságokat típusalapú bizonyításként fejezzék ki. A gyakorlatban azonban a Go a programozóknak a viszonylag erős típusrendszerek biztonsági és futásidőbeli teljesítménybeli előnyeinek nagy részét biztosítja, anélkül, hogy egy összetett típusrendszer terhei terhelnék őket. A Go ösztönzi a kortárs számítógépes rendszerek tervezésének tudatosítását, különösen a lokalitás fontosságát. A beépített adattípusai és a legtöbb könyvtári adatszerkezet úgy van kialakítva, hogy természetesen explicit inicializálás vagy implicit konstruktorok nélkül működjön, így viszonylag kevés memóriafoglalást és memóriairást rejt a kód.

A Go összesített típusai (structok és tömbök) közvetlenül tartják elemeiket, így kevesebb tárolást és kevesebb allokációt és mutató indirekciót igényelnek, mint az indirekt mezőket használó nyelvek. És mivel a modern számítógép egy párhuzamos gép, a Go rendelkezik a CSP-n alapuló párhuzamossági funkciókkal, ahogy azt már korábban említettük. A Go könnyű szálainak vagy goroutine-ainak változó méretű halmai kezdetben elég kicsik ahhoz, hogy egy goroutine létrehozása olcsó, és egymillió létrehozása praktikus. A Go szabványos könyvtára, amelyet gyakran úgy jellemeznek, hogy "elemekkel együtt" érkezik, tiszta építőelemeket és API-kat biztosít az I/O, a szövegfeldolgozás, a grafika, a kriptográfia, a hálózatépítés és az elosztott alkalmazások számára, számos szabványos fájlformátum és protokoll támogatásával. A könyvtárak és az eszközök széleskörűen használják a konvenciókat, hogy csökkentsék a konfiguráció és a magyarázat szükségességét, ezáltal egyszerűsítve a programlogikát, és a különböző Go programokat egymáshoz hasonlóbba és ezáltal könnyebben tanulhatóvá téve. A go eszközzel épített projektek csak fájl- és azonosítóneveket, valamint egy alkalmi speciális megjegyzést használnak a projekt összes könyvtárának, futtatható fájljának, tesztjének, tesztjének, benchmarkjának, példájának, platform-specifikus változatának és dokumentációjának meghatározásához; a Go forrás maga tartalmazza az építési specifikációt. [6]

## 7.2. Csomagkezelő

Egy szerény méretű program ma akár 10.000 függvényt is tartalmazhat. Szerzőjének azonban csak néhányra kell gondolnia, és még kevesebbet kell megterveznie, mert a túlnyomó többséget mások írták, és csomagok révén újrafelhasználásra bocsátották. A Go több mint 100 szabványos csomagot tartalmaz, amelyek a legtöbb alkalmazás alapját képezik. A Go közösség, a csomagtervezés, -megosztás, -újrafelhasználás és -fejlesztés virágzó ökoszisztémája még sokkal többet publikált, és ezek kereshető indexe megtalálható a <http://godoc.org> oldalon. A Go a go tool-t is tartalmazza, egy kifinomult, de egyszerűen használható parancsot a Go csomagok munkaterületeinek kezelésére.

Minden csomagrendszer célja, hogy a nagy programok tervezését és karbantartását praktikussá tegye azáltal, hogy a kapcsolódó funkciókat könnyen értelmezhető és módosítható egységekbe csoportosítja, függetlenül a program többi csomagjától. Ez a modularitás lehetővé teszi, hogy a csomagokat különböző projektek megosszák és újra felhasználják, egy szervezeten belül terjesszék, vagy a nagyvilág számára elérhetővé tegyék. Minden csomag egy külön névteret határoz meg, amely az azonosítóit foglalja magába. Minden név egy adott csomaghoz kapcsolódik, így rövid, egyértelmű neveket választhatunk a leggyakrabban használt típusokhoz, függvényekhez stb. anélkül, hogy konfliktusokat okoznánk a program más részeivel. A csomagok kapszulázást is biztosítanak azáltal, hogy szabályozzák, mely nevek láthatók vagy exportálhatók a csomagon kívülre. A csomagtagok láthatóságának korlátozása elrejtja a segédfüggvényeket és típusokat a csomag API-ja mö-

g , lehet v  t ve a csomag karbantart j nak, hogy az implement ci  megv ltoztat s val biztos lehessen abban, hogy a csomagon k v li k d t ez nem  rinti. A l that s g korl toz sa a v ltoz kat is elrejt ,  gy az  gyfelek csak olyan export lt f ggv nyeken kereszt l  rhetik el  s friss thetik  ket, amelyek meg rzik a bels  invari nsokat, vagy k lcs n s k z r st k nyszer tenek ki egy p rhuzamos programban. Amikor megv ltoztatunk egy f jlt,  jra kell ford tanunk a f jl csomagj t  s potenci lisan az  sszes olyan csomagot, amely f gg t le. A Go ford t sa jelent sen gyorsabb, mint a legtöbb m s ford tott nyelv , m g akkor is, ha a null r l  p tkez nk. A ford t  sebess g nek h rom f  oka van. El sz r is, minden import lt csomagot explicit m don fel kell sorolni minden forr sf jl elej n,  gy a ford t nak nem kell egy e sz f jl elolvasnia  s feldolgoznia, hogy meghat rozza a f gg s geket. M sodszor, egy csomag f gg s gei egy  r ny tott aciklikus gr fot alkotnak,  s mivel nincsenek ciklusok, a csomagok k l n-k l n  s esetleg p rhuzamosan ford that k. V g l, egy leford tott Go csomag objektumf jlja nemcsak mag ra a csomagra, hanem a f gg s geire vonatkoz  exportinform ci kat is r gz ti. Egy csomag ford t sakor a ford t nak minden import l shoz be kell olvasnia egy objektumf jlt, de ezeken a f jlokon t l nem kell n znie. [6]

### 7.3. Tesztel s

A mai programok természetesen sokkal nagyobbak  s  sszetettebbek, emiatt rengeteg er fesz t st tettek olyan technik k kifejleszt s re, amelyek ezt a komplexit st kezelhet v  teszik. K l n sen k t technika emelkedik ki hat konys g val. Az el s  a programok rutin-szer  szak rt i fel lvizsg lata, mi tt bevezet sre ker ln nek. A m sodik, amely ennek a fejezetnek a t rgya, a tesztel s. A tesztel s, amely alatt implicit m don az automatiz lt tesztel st  rtj k, olyan kis programok  r s nak gyakorlata, amelyek ellen rzik, hogy a tesztelt k d (a production k d) a v rt m don viselkedik-e bizonyos bemenetek esetén, amelyek  ltal ban vagy gondosan megv lasztottak, hogy bizonyos funkci kat gyakoroljanak, vagy v letlenszer ek, hogy sz lesk r  lefedetts get biztos tsanak. A szoftvertesztel s ter lete hatalmas. A tesztel s feladata minden programoz t id nk nt, n h ny programoz t pedig  lland an foglalkoztat. A tesztel ssel kapcsolatos szakirodalom t bb ezer nyomtatott k nyvet  s t bb milli  szavas blogbejegyz eket tartalmaz. Minden elterjedt programoz si nyelvben t bb tucat tesztk sz t sre sz nt szoftvercsomag l tezik, n melyikben rengeteg el m letet,  s  gy t nik, a ter let nem kevés pr f t t vonz, akiknek kultikus k vet i vannak. Ez szinte e g ahhoz, hogy meggy zz k a programoz kat arról, hogy a hat kony tesztek  r s hoz egy e sz sor  j k zs get kell elsaj t taniuk. A Go tesztel si megk zel t se ehhez k pest meglehet sen alacsony technol giai sz nvonal nak t nhet. Egyetlen parancsra t maszkodik, a `go test`,  s egy sor konvenc ra a tesztf ggv nyek  r s hoz, amelyeket a `go test` k pes lefuttatni. A viszonylag k nny  mechanizmus hat kony a tiszta tesztel shez,  s természetesen kiterjeszthet  a benchmarkokra  s a dokument ci hoz sz ks ges szisztematikus p ld kra.

A gyakorlatban a tesztk d  r sa nem sokban k l nb zik az eredeti program  r s t l. R vid f ggv nyeket  r nk, amelyek a feladat egy r sz re  sszpontos tanak. Vigy znunk kell a peremfelt telekre, gondolkodnunk kell az adatszerkezeteken,  s  rveln nk kell, hogy a megfelel  bemenetekb l milyen eredm nyeket kell produk lnia a sz m t snek. De ez ugyanaz a folyamat, mint a k z ns ges Go k d  r sa; nem kell  j jel lesek, konvenc kat  s eszk z ket haszn lni hozz . A `go test` alparancs a Go csomagok tesztel je, amelyek bizonyos konvenc k szerint vannak megszervezve. Egy csomagk nyvt rban a `__test.go` v gz d s  f jlok nem r szei a `go build`  ltal  p tett csomagnak, de a `go test`  ltal  p tett csomagnak igen. A `*__test.go` f jlokon bel l h romf le funkci t kezel nk speci lisan : teszt-

tek, benchmarkok és példák. A tesztfüggvény, amely egy olyan függvény, amelynek neve `Test` -el kezdődik, a program logikájának helyes viselkedését vizsgálja; a `go test` meghívja a tesztfüggvényt és az eredményt jelenti, amely vagy `"PASS"` vagy `"FAIL"`. A benchmark függvény neve `Benchmark` -al kezdődik, és valamilyen művelet teljesítményét méri; a `go test` a művelet átlagos végrehajtási idejét jelenti. Egy példafüggvény pedig, amelynek neve `Example` -vel kezdődik, gépileg ellenőrzött dokumentációt biztosít. A `go test` eszköz átvizsgálja a `*_test.go` fájlokat ezen speciális függvények után, létrehoz egy ideiglenes főcsomagot, amely mindegyiket a megfelelő módon hívja meg, felépíti és lefuttatja, jelenti az eredményeket, majd takarít. [6]



## 8. fejezet

# Elosztott verziókezelő rendszerek

### 8.1. Git

A verziókezelő rendszer (későbbiekben Version Control System vagy VCS) nyomon követi a változtatások történetét, ahogy az emberek és a csapatok együtt dolgoznak a projekteken. Ahogy a fejlesztők változtatásokat végeznek a projekten, a projekt bármely korábbi verziója bármikor visszaállítható.

A fejlesztők áttekinthetik a projekttörténetet, hogy megtudják:

- Milyen változtatásokat végeztek?
- Ki végezte a változtatásokat?
- Mikor történtek a változtatások?
- Miért volt szükség a változtatásokra?

A VCS-ek minden egyes közreműködőnek egységes és konzisztens képet adnak a projektről, felszínre hozva a már folyamatban lévő munkát. A változtatások átlátható előzményeinek, a változtatásokat végző személyeknek és a projekt fejlődéséhez való hozzájárulásuknak a megtekintése segít a csapattagoknak abban, hogy a független munka során is összhangban maradjanak.

Egy elosztott verziókezelő rendszerben (későbbiekben Distributed Version Control System vagy DVCS) minden fejlesztő rendelkezik a projekt és a projekttörténet teljes másolatával. Az egykor népszerű központosított verziókezelő rendszerekkel ellentétben a DVCS-eknek nincs szükségük állandó kapcsolatra egy központi adattárral. A Git a legnépszerűbb elosztott verziókezelő rendszer. A Git-et gyakran használják nyílt forráskódú és kereskedelmi szoftverfejlesztésre egyaránt, és jelentős előnyökkel jár az egyének, a csapatok és a vállalkozások számára.

- A Git segítségével a fejlesztők egy helyen láthatják a változtatások, döntések és a projekt előrehaladásának teljes idővonalát. Attól a pillanattól kezdve, hogy hozzáférnek egy projekt előzményeihez, a fejlesztő minden szükséges kontextussal rendelkezik ahhoz, hogy megértse azt, és elkezdjen hozzájárulni.
- A fejlesztők minden időzónában dolgoznak. Egy olyan DVCS-vel, mint a Git, az együttműködés bármikor megtörténhet, miközben a forráskód integritása megmarad.

Az ágak használatával a fejlesztők biztonságosan javasolhatnak változtatásokat a termelési kódhoz.

- A Git-et használó vállalkozások lebonthatják a csapatok közötti kommunikációs akadályokat, és a csapatok a legjobb munkájukra koncentrálhatnak. Ráadásul a Git lehetővé teszi, hogy az egész vállalat szakértői együttműködjenek a nagyobb projektekben.

A repository vagy Git-projekt a projekthez tartozó fájlok és mappák teljes gyűjteményét foglalja magában, az egyes fájlok revíziós előzményeivel együtt. A fájlok előzményei időbeli pillanatképek formájában jelennek meg, amelyeket commitoknak nevezünk. A commitok több fejlesztési sorba, úgynevezett ágakba szervezhetők. Mivel a Git egy DVCS, a tárolók önálló egységek, és bárki, aki rendelkezik a tároló másolatával, hozzáférhet a teljes kódbázishoz és annak történetéhez. A parancssor vagy más egyszerű kezelőfelületek használatával a Git-tár lehetővé teszi a következőket is: interakció az előzményekkel, a tár klónozása, ágak létrehozása, átadás, beolvasztás, összevonás, a kódverziók közötti változások összehasonlítása stb.

Az olyan platformokon keresztül, mint a GitHub, a Git több lehetőséget biztosít a projektek átláthatóságára és együttműködésére is. A nyilvános adattárak segítik a csapatok együttműködését a lehető legjobb végtermék létrehozásában. [2]

## 8.2. GitHub

A GitHub egy olyan fejlesztési platform, amely lehetővé teszi a kódok tárolását és felülvizsgálatát, a projektek kezelését és a szoftverek készítését 50 millió fejlesztővel együttműködve. Miért épít mindenki a GitHubra? Mert biztosítja azokat a fontos DevOps funkciókat, amelyekre a vállalatoknak és a különböző méretű szervezeteknek szükségük van a nyilvános és magánprojektjeikhez. Legyen szó funkciók tervezéséről, hibák javításáról vagy a változtatásokon való együttműködésről, a GitHub az a hely, ahol a világ szoftverfejlesztői összegyűlnek, hogy dolgokat hozzanak létre és aztán jobbra teszik őket. A GitHub egy felhőplatform, amely a Git-et használja alapvető technológiaként. Leegyszerűsíti a projekteken való együttműködés folyamatát, és olyan weboldalt, parancssori eszközöket és általános folyamatot biztosít, amely lehetővé teszi a fejlesztők és a felhasználók közös munkáját.

### 8.2.1. GitHub Actions ismertető

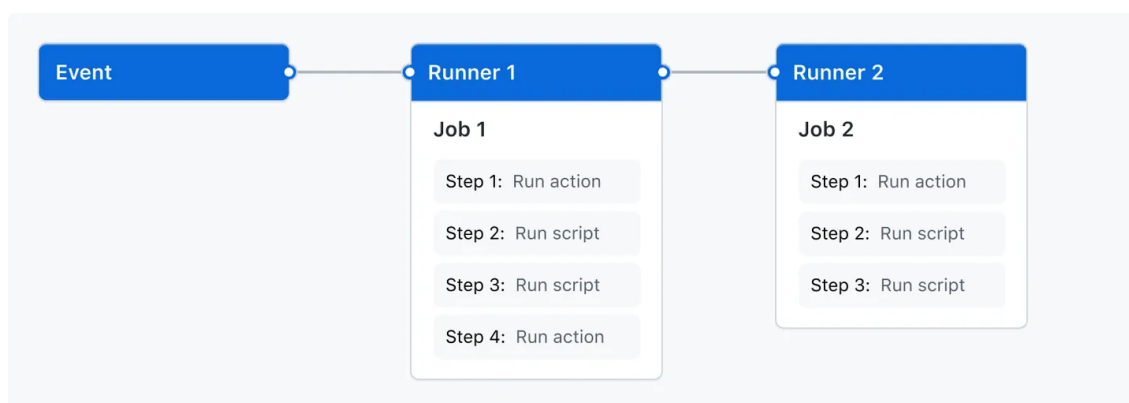
A GitHub Actions egy folyamatos integrációs és folyamatos szállítási (későbbiekben CI/CD) platform, amely lehetővé teszi, hogy automatizálja a build, a tesztelés és a telepítés folyamatot. Létrehozhat olyan munkafolyamatokat (későbbiekben workflow), amelyek minden egyes pull-kérelem után build-el és tesztel a tárolóban (későbbiekben repository).

A GitHub Actions túlmutat a DevOps-on, és lehetővé teszi, hogy workflow-kat futtasson, amikor más események is történnek a tárolóban. Például futtathat olyan workflow-t, amely automatikusan hozzáadja a megfelelő címkéket, amikor valaki új problémát (későbbiekben issue) hoz létre az repository-ban.

A GitHub Linux, Windows és macOS virtuális gépeket biztosít a workflow-ok futtatásához, de saját adatközpontban vagy felhő-infrastruktúrában is hosztolhatsz saját, saját üzemeltetésű futásokat (későbbiekben runners). [3]

### 8.2.2. A GitHub Actions elemei

Konfigurálhat egy GitHub Actions workflow-t, amely akkor indul el, amikor egy esemény történik a repository-ban, például egy pull request megnyitásakor vagy egy issue létrehozásakor. A workflow egy vagy több feladatot tartalmaz, amelyek futhatnak egymás utáni sorrendben vagy párhuzamosan. Minden egyes feladat a saját virtuális gépi runner-en vagy egy konténeren belül fut, és egy vagy több lépésből áll, amelyek vagy egy általad definiált szkriptet futtatnak, vagy egy műveletet futtatnak, ami egy újrafelhasználható bővítmény, amely egyszerűsítheti a workflow-t. [3]



**8.1. ábra.** Egy esemény diagramja, amely az 1. runner-t az 1. feladat futtatására indítja, ami a 2. runner-t a 2. feladat futtatására indítja. Mindegyik feladat több lépésre van bontva. [3]

### 8.2.3. Workflows

A workflow egy konfigurálható automatizált folyamat, amely egy vagy több feladatot futtat. A workflow-kat az repository-ba bevitt YAML fájl határozza meg, és akkor futnak, amikor az repository-ban lévő esemény kiváltja őket, vagy manuálisan, illetve meghatározott ütemezés szerint is elindíthatók.

A workflow-k a `.github/workflows` könyvtárban vannak definiálva egy repository-ban, és egy repository-hoz több workflow is tartozhat, amelyek mindegyike más-más feladatokat végezhet. Például lehet egy workflow a pull-kérések létrehozására és tesztelésére, egy másik workflow az alkalmazás telepítésére minden egyes kiadás létrehozásakor, és még egy másik workflow, amely minden egyes alkalommal hozzáad egy címkét, amikor valaki új issue-t nyit. Hivatkozhat egy workflow-ra egy másik workflow-on belül. [3]

### 8.2.4. Events

Az esemény (event) egy konkrét tevékenység egy repository-ban, amely egy workflow végrehajtását indítja el. Az aktivitás például a GitHubról származhat, amikor valaki lét-

rehoz egy pull requestet, megnyit egy issue-et, vagy egy commit-ot küld a repositoryba. A workflow-t időzítéssel, REST API-ba való beküldéssel vagy manuálisan is elindíthatja. [3]

### 8.2.5. Jobs

A feladat (job) egy workflow lépéseinek olyan csoportja, amelyet ugyanazon a runner-en hajtanak végre. Minden lépés vagy egy végrehajtandó shell szkript, vagy egy végrehajtandó művelet. A lépések sorrendben kerülnek végrehajtásra, és egymástól függnnek. Mivel minden lépés ugyanazon a futón kerül végrehajtásra, az adatokat megoszthatja egyik lépérről a másikra. Például lehet egy lépés, amely elkészíti az alkalmazást, majd egy lépés, amely teszteli a felépített alkalmazást.

Beállíthatja egy feladat függőségeit más feladatokkal; alapértelmezés szerint a feladatoknak nincsenek függőségeik, és párhuzamosan futnak egymással. Ha egy feladat függőséget vesz fel egy másik feladattól, akkor megvárja a függő feladat befejezését, mielőtt lefuthatna. Lehet például több, különböző architektúrákhoz tartozó építési feladat, amelyek nem függnnek egymástól, és egy csomagolási feladat, amely függ ezektől a feladatoktól. A build feladatok párhuzamosan futnak, és ha mindegyik sikeresen befejeződött, akkor a csomagolási feladat is lefut. [3]

### 8.2.6. Actions

A művelet (action) a GitHub Actions platform egyéni alkalmazása, amely egy összetett, de gyakran ismétlődő feladatot hajt végre. Egy művelet használatával csökkentheti a workflow fájlokban írt ismétlődő kód mennyiségét. Egy művelet lehívhatja a git-tárat a GitHubról, beállíthatja a megfelelő eszköztárat az build környezetéhez, vagy beállíthatja a hitelesítést a felhőszolgáltatónál.

Írhat saját műveleteket, vagy találhat workflow-okhoz felhasználható műveleteket a GitHub Marketplace-en. [3]

### 8.2.7. Runners

A futtató (későbbiekben runner) egy olyan kiszolgáló, amely a workflow-okat futtatja, amikor azoknak el kell indulniuk. Minden runner egyszerre egyetlen feladatot futtathat. A GitHub Ubuntu Linux, Microsoft Windows és macOS runnereket biztosít a workflow-ok futtatásához; minden workflow futtatása egy friss, újonnan rendelkezésre bocsátott virtuális gépen történik. A GitHub nagyobb futókat is kínál, amelyek nagyobb konfigurációkban állnak rendelkezésre. [3]

### 8.2.8. Egy példa workflow

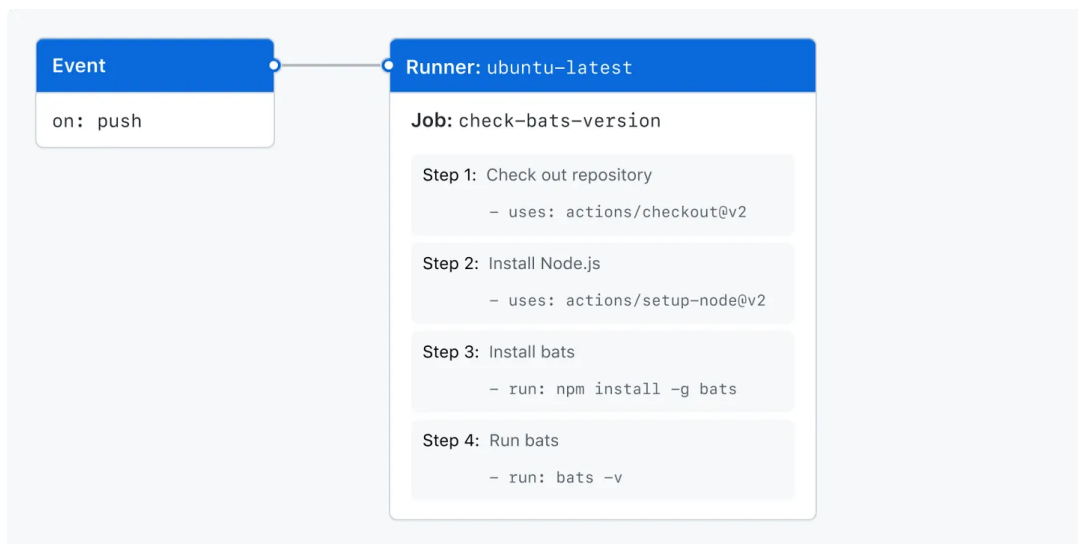
A GitHub Actions YAML szintaxist használ a workflow-ok meghatározásához. Minden workflow különálló YAML-fájlként tárolódik a kódtárában, a `.github/workflows` nevű könyvtárban.

Létrehozhat egy példát a repository-jában, amely automatikusan elindít egy sor parancsot, amikor kódot küldünk be. Ebben a workflow-ban a GitHub Actions ellenőrzi a

beküldött kódot, telepíti a bats tesztelési keretrendszert, és lefuttat egy alapvető parancsot a bats verziójának kiadására: `bats -v`. [3]

```
name: learn-github-actions
run-name: ${{ github.actor }} is learning GitHub Actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

Ezen az ábrán látható az imént létrehozott példa workflow fájl, valamint a GitHub Actions összetevőinek hierarchikus elrendezése. Minden lépés egyetlen műveletet vagy szkriptet hajt végre. Az 1. és 2. lépés műveleteket, míg a 3. és 4. lépés szkripteket futtat.



**8.2. ábra.** Egy workflow triggerét, runner-ét és feladatát bemutató ábra. A feladat 4 lépésre van bontva. [3]

## 9. fejezet

# KLI CLI

### 9.1. Tervezés

A projekt tervezésekor kettő logikai részre osztottam a CLI-t. Front-end és back-end csomagokat külön fejlesztettem, melyeket végül egy projektbe integráltam be. A szándék az volt, hogy a back-end részt más által fejlesztett cli (avagy front-end) is tudja használni.

### 9.2. Model-View-Controller ismertetése

A MVC (Model-View-Controller) programtervezési mintát használtam a fejlesztéskor. Ez a modell segített abban, hogy külön válasszam a nézet réteget az adatrétegtől. A két réteg közötti kommunikációt egy vezérlői réteg oldja meg. Itt történik az adatok átadása, validálása az üzleti logika segítségével. Mivel a modulok a projektben függetlenek egymástól köszönhetően az MVC alkalmazásának, így könnyű volt a későbbiekben változtatni és bővíteni az egyes modulok viselkedését. A tesztelhetőséget is nagyban megkönnyítette, mivel lehetőség adódott szimulálni (mock-olni) egy adott modult a másik modul teszteléskor.

A hátrányait figyelembe vettem, amikor döntöttem a projektem felépítéséről. A legtöbb feladatot a back-end végzi el, a vezérlői réteg kódja pedig MVC boilerplate kód.

### 9.3. Megvalósítás

#### 9.3.1. Front-end

Egy nyílt forráskódú könyvtárcsomag használatával felépítettem egy konzolos applikációt Go nyelv használatával. Az alkalmazás képes automatizáltan telepíteni és eltávolítani helm chart csomagokat a Kubernetes cluster-ból. Használata a következő flag-eken keresztül történik:

- install, ahol az install után vár egy resource-t, amit telepít
- uninstall, ahol az uninstall után vár egy resource-t, amit eltávolít

Használati példák:

- `kli install --resource istio-operator`
- `kli uninstall --resource cluster-registry`

Amikor meghívjuk az `install` flag-ek akkor a telepítés előtt ellenőrzés zajlik le, majd egy repository hozzáadás és frissítés. Ezután települ fel a `helm` csomag. A célja ennek a front-end csomagnak, hogy minimalizálja a függőségeit a külső moduloktól és csak a `kubereflex` back-end csomag funkcióira támaszkodjon.

### 9.3.2. Back-end

A `kubereflex` csomag felépítése modulárisra lett tervezve. A csomag fő modulja arra szolgál, hogy publikusan elérhetővé tegye az amúgy privát funkciókat az almodulokból. Emiatt a fő modul egyfajta kontroller szerepet tölt be. A külső modulok használata itt is minimalizálva van. Ideális esetben csak az almodulokat importálja be.

#### **kubereflex modul**

Ez a back-end modul, melynek több almodulja van. Ebben a modulban található a üzleti logikai réteg, melynek segítségével történik meg az adatcsere.

#### **kubectl almodul**

Ez az almodul felel az összes olyan funkcióért, amit a Kubernetes-ben kell végrehajtani. Ebben a csomagban van a névtér létrehozásához és már létrehozott névtér ellenőrzéséhez szükséges kódrészlet és majd itt lesz más csak kubernetes funkció is a jövőben, mely segít a cluster ellenőrzésében és a telepítési feltételek kialakításában, meglévő resource-ok konfigurálásában.

#### **helm almodul**

Ez az almodul felel az összes olyan funkcióért, amit a Helm csomagkezelővel szeretnénk végrehajtani. Ilyen például az előre kész helm chart applikáció telepítése, törlése, repository hozzáadás, frissítés. Erre az almodulra fog a front-end nagyon sokat támaszkodni, mert az applikációk telepítése gyakran ezen az úton zajlik.

#### **io almodul**

Ez az almodul felel minden fájl művelettel kapcsolatos funkciókért. Elsősorban a `config` fájlok betöltését teszi lehetővé a memóriába, de CRD leíró yml fájl letöltésére is képes. Ennek az almodulnak a segítségével lehetséges a kontextus választó menü, mivel a kubernetes elérhető kontextusokat a konfigurációs fájlban vannak eltárolva.

## 9.4. Tesztelés

A csomagokhoz írtam teszt fájlokat, melyek minden egyes függvényt letesztelnek minta adatok segítségével. Ezek a tesztek sokat segítenek olyan hibák felderítésében, ami normális futtatáskor nem érzékelhető köszönhetően a teszt adatoknak.

A teszteléshez szükséges volt létrehoznom pár teszt specifikus függvényt, mely segít a teszt adatokat és a teszt klienseket beállítani, mielőtt a tesztelni kívánt függvény meghívásra kerülne.

## 9.5. Kritikai elemzése

A cli telepítési folyamatának reprezentálása lehetne felhasználóbarátabb azzal, hogy látszódná az elvégzett feladatok és az összes feladatok aránya, a várható idő amíg tart a telepítés és hiba esetén dokumentációt az adott hiba okáról. Flag-ek kiosztása lehetne intuitívabb. Tesztelés lehetne jobb.

## 9.6. Továbbfejlesztési lehetőségek

Ennek a CLI programnak a továbbfejlesztését úgy képzelem el, hogy olyan funkciók kerülnek implementálásra, melyek segítenek a felhasználónak az Istio operátor telepítésében.

### 9.6.1. Operátor frissítése

Meglévő klaszter telepítésnél lehetőség adódna frissíteni a komponenseket egy adott verziószámra. Ez a verziószám lehetne a legfrissebb kiadás, de régebbi verziók is rendelkezésre állnának.

### 9.6.2. Log rendszer

Lehetőség lenne megadni flag-eken keresztül, hogy milyen fájlba írja ki a kimenetet. Előre meg lehetne határozni több szintű log-olási részletességet és lehetőség adódna a standard output-ra való kiírás minimalizálására. A log fájl jól struktúrált lenne, ami segítené visszanezhetővé tenni a telepítési folyamat sikerességét. A log fájlban sokkal több részletet lehetne leírni az adott folyamatról vagy annak hibájáról, míg ez a részletesség a standard output-on zavaró lehetne.

### 9.6.3. Virtuálisgép integráció megvalósítása

Lehetőség lenne a klaszterhez hozzáadni létező virtuálisgépeket, ezzel segítve a át-migrálást a klaszterre. Vannak olyan workload-ok, melyek jobban teljesítenek monolitikus applikációként és negatív hatással lenne a mikroszolgáltatási struktúra.



#### 9.6.4. Klaszterek létrehozása telepítés előtt

A felhasználónak lehetősége lenne létrehozni előre definiált vagy a program alapértelmezett értékeit használva egy vagy több klasztert. Ennek megvalósítása minden cloud provider-nél más és más, mert az authetntikálás, autorizálás és API-n keresztüli paran-csok egyediek.

#### 9.6.5. Automatizált post-install

Miután megtörtént a telepítés és attach, lehetőség lenne kiválasztani a menüből előre összeállított vagy egyedileg definiált deploymenteket és helm chart-ok telepítésére. A post-install konfigurációs leíró fájlt meg lehetne adni egy flag-el, így teljesen automatikusan futna le a teljes folyamat. A felkínált post-install lista olyan elemeket tartalmazna, melyek nagy segítséget nyújthatnak a klaszter további üzemeltetésében. Előre összeállított deploymentekre példák:

- Kubernetes dashboard
- Prometheus monitoring system
- Grafana dashboard
- cert-manager TLS cert manager

## 10. fejezet

# Összefoglaló

Egyetlen Kubernetes klaszter számos opciót és lehetőséget kínál. Ha azonban nagyobb rugalmasságot, skálázhatóságot, ellenállóképességet és biztonságot keres, akkor a több klaszteres architektúra a jobb választás az alkalmazása számára. Ez nem jelenti azt, hogy a több klaszteres Kubernetes sétagaloppnak tekinthető - a vele járó fontos előnyök ellenére van néhány komoly kihívás, amit figyelembe kell vennie. Mindenesetre az egy- vagy többklaszteres architektúra közötti döntés kulcsa az, hogy az előnyök meghaladják a kihívásokat és a költségeket.

# Köszönetnyilvánítás

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

# Irodalomjegyzék

- [1] Cisco: Istio operator - banzai cloud. <https://banzaicloud.com/products/istio-operator/>. (Hozzáférés dátuma: 2023 május 31.).
- [2] GitHub Docs: About git. <https://docs.github.com/en/get-started/using-git/about-git>. (Hozzáférés dátuma: 2023 május 31.).
- [3] GitHub Docs: Understanding github actions. <https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>. (Hozzáférés dátuma: 2023 május 31.).
- [4] Istio: The istio service mesh. <https://istio.io/latest/about/service-mesh/>. (Hozzáférés dátuma: 2023 május 31.).
- [5] Jacob Schmitt: What is helm? a complete guide. <https://circleci.com/blog/what-is-helm/>. (Hozzáférés dátuma: 2023 május 31.).
- [6] Alan A. A. Donovan · Brian W. Kernighan: *The Go Programming Language*. 2015, Addison-Wesley, xiii–307. p. ISBN 978-0-134-19044-0.
- [7] Kev Zettler: What is cloud computing? an overview of the cloud. <https://www.atlassian.com/microservices/cloud-computing>. (Hozzáférés dátuma: 2023 május 31.).
- [8] Marko Lukša: *Kubernetes in Action*. 2017, Manning Publications, 3–513. p. ISBN 978-1-617-29372-6.
- [9] Red Hat: What is a kubernetes operator? <https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator>. (Hozzáférés dátuma: 2023 május 31.).
- [10] The Kubernetes Authors: kind - initial design. <https://kind.sigs.k8s.io/docs/design/initial/>. (Hozzáférés dátuma: 2023 május 31.).
- [11] Traefik Labs: Understanding multi-cluster kubernetes: Architecture, benefits, and challenges. <https://traefik.io/glossary/understanding-multi-cluster-kubernetes/>. (Hozzáférés dátuma: 2023 május 31.).