BUDAPESTI GAZDASÁGI

EGYETEM PÉNZÜGYI ÉS

SZÁMVITELI KAR

SZAKDOLGOZAT

Földvári Ádám

Levelező

Gazdasági-
informatikus

Üzleti adatelemző

2021

BUDAPESTI GAZDASÁGI EGYETEM

PÉNZÜGYI ÉS SZÁMVITELI KAR

Automatizálás a szoftverfejlesztésben folyamatos integráció és folyamatos szállítás eszközökkel

Belső konzulens: Kaderják Gyula

Külső konzulens: Drahos Tamás

Földvári Ádám

Levelező

Gazdasági-informatikus

Üzleti Adatelemző

2021

NYILATKOZAT

Alulírott Földvári Ádám büntetőjogi felelősségem tudatában nyilatkozom, hogy a szakdolgozatomban foglalt tények és adatok a valóságnak megfelelnek, és az abban leírtak a saját, önálló munkám eredményei.

A szakdolgozatban felhasznált adatokat a szerzői jogvédelem figyelembevételével alkalmaztam.

Ezen szakdolgozat semmilyen része nem került felhasználásra korábban oktatási intézmény más képzésén diplomaszerzés során.

Tudomásul veszem, hogy a szakdolgozatomat az intézmény plágiumellenőrzésnek veti alá.

Budapest, 2021. december 2.

Földvári Ádám
....
hallgató aláírás

Tartalomjegyzék

1. Bevezetés 2

2. Szoftverfejlesztési módszertanok 4

2.1 Lineáris módszertanok 4

2.2 Iteratív módszertanok 5

2.3 DevOps 7

3. Az automatizálás lépéseinek és eszközeinek bemutatása 12

3.1 Verzió követés 12

3.2 Tesztautomatizálás 14

3.3 Konténerizáció 15

3.4 Automatikus élesítés és szállítás 17

3.5 Monitorozás 18

3.6 CI/CD pipeline 19

4. Esettanulmányok elkészítése 22

4.1 Tervezés 22

4.2 Alkalmazás elkészítése 24

4.3 HerokuCI és Heroku platform (A eset) 25

4.4 Github Actions és Heroku platform (B eset) 27

4.5 Github Actions és Kubernetes (C eset) 29

4.6 Google Cloud Build és Kubernetes (D eset) 31

5. Eredmények 32

5.1 Empirikus eredmények 32

5.2 Kvalitatív eredmények 35

6. Összefoglalás 40

Irodalomjegyzék 44

Mellékletek 47

1. Bevezetés

A szoftverfejlesztés az egyik leggyorsabban fejlődő iparág világszerte. A folyamatosan fejlődő technológiákkal lépést tartani az egyik legjelentősebb kihívás a szektorban dolgozóknak. Újabb és újabb trendek, metodológiák és szakzsargonok ütik fel a fejüket, azonban sokszor évekbe telik, míg egy újonnan megjelenő kifejezésből valóban kidolgozott, a gyakorlatban alkalmazható és akadémiai szinten is kutatható modell lesz.

Lassan négy éve dolgozom szoftverfejlesztőként, ezért az én munkámnak is része a technológiák és trendek követése és alkalmazása különböző feladataim megoldása során. Az elmúlt néhány évben egyre többször találkoztam a „DevOps” kifejezéssel munkahelyeimen, az interneten, konferenciákon és meetupokon. Sőt, mikor legutóbb állást kerestem, arra lettem figyelmes, hogy kifejezetten sok DevOps mérnöki pozíciót hirdetnek a munkáltatók.

A DevOps fogalmát különböző aspektusból közelítik meg az emberek attól függően, hogy milyen szerepet töltenek be az IT szektorban. Egy vezetőnek például inkább a kollaboratív kultúra kialakítását és a fejlesztői és üzemeltetői csapat munkájának összehangolását jelenti. Mivel én szoftver fejlesztői szemszögből közelítem meg a kifejezést, számomra a régóta megszokott agilis módszertanok alkalmazása mellett főleg olyan folyamatok automatizációját jelenti, melyeket másként hosszú időbe telne kivitelezni, ráadásul nagy lenne az emberi hiba lehetősége. Ilyenek például a szoftvertesztelés, az integráció és a szoftver éles környezetbe való beüzemelése. Az automatizálandó feladatokat megfelelő sorrendben kell kivitelezni, ehhez pedig egy speciális rendszert, úgynevezett *pipeline*-t kell létrehozni. Ezt *CI/CD pipeline*-nak is szokás nevezni: a kifejezés a „continuous integration, continuous delivery” szavak rövidítéséből származik, a folyamatos integrációra és folyamatos szállításra utal.

Szakdolgozatom első felében szeretném bemutatni a DevOps fogalmát elméleti oldalról, áttekintve a vonatkozó szakirodalmat. Megvizsgálom a DevOps kapcsolatát a szoftverfejlesztési metodológiákkal és az automatizációval. A második fejezetben arra keresem a választ, hogy milyen kapcsolatban áll a DevOps az agilis szemlélettel, milyen kapcsolódó elméletek és gyakorlatok léteznek az automatizáción túl. A harmadik fejezetben bemutatom a DevOps automatizáció eszköztárát és a megvalósításához szükséges lépéseket, úgymint: verzió

követés, tesztautomatizálás, konténerizáció, automatikus élesítés és szállítás, valamint monitorozás. Áttekintem a rendelkezésre álló felhő szolgáltatásokat is, amelyek segítik a rendszer működését.

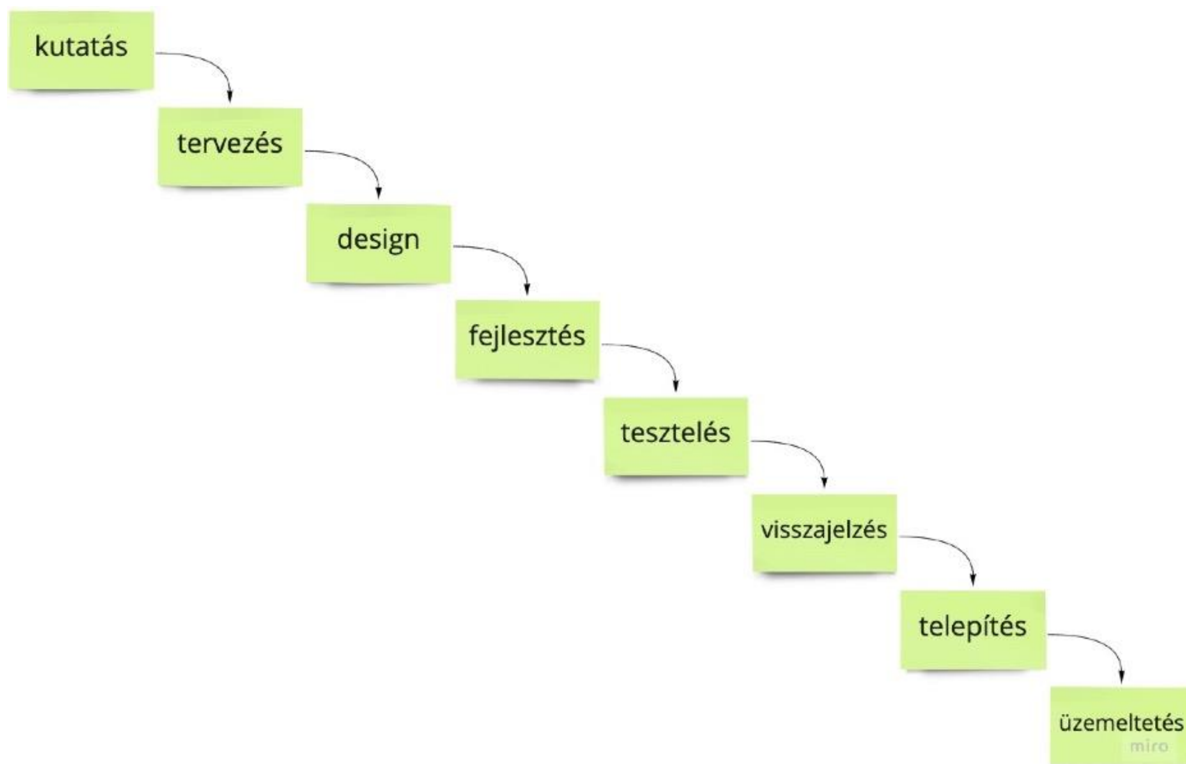
A téma elméleti áttekintése után pedig be fogom mutatni, milyen feladatok várnak az informatikus csapatokra, amikor úgy döntenek, hogy DevOps szemléletben kezdenek neki egy projektnek. Az ezzel járó automatizáció különböző szoftverek és eszközök együttes használatának és megfelelő konfigurációjának az eredménye. Ezért a negyedik fejezetben elkészítek négy esettanulmányt, amelyeken keresztül áttekintést adok a legnépszerűbb eszközökről és egyben gyakorlati példát hozok a lehetséges megoldásokra. Az ötödik fejezetben először empirikus módon fogom összehasonlítani az elkészült esettanulmányokat a következő szempontok alapján: fejlesztésre fordított idő, DevOps gyakorlatok érvényesülése, kompatibilitás és limitációk. Végül pedig az elkészült CI/CD pipeline-okat teljesítmény és skálázhatóság szempontjából is meg fogom vizsgálni három kísérlet mérési eredményeinek az összehasonlításával. Ezzel szeretném szemléltetni a DevOps automatizációval járó feladatokat, a feladatok elvégzéséhez szükséges programozói eszköztárat és tudást, valamint megtalálni azokat a szempontokat, amelyeket érdemes figyelembe venni egy ilyen komplex folyamat megtervezése és kivitelezése során.

2. Szoftverfejlesztési módszertanok

A szoftverfejlesztés története az 1960-as évekig nyúlik vissza. Ekkora már más iparágakban, mint például az építőipar, mérnöki munka és hadiipar, felismerték, hogy az emberi, anyagi és pénzügyi erőforrások tudatos tervezése és szervezése elengedhetetlen amennyiben sikeresen szeretnénk kivitelezni egy egyedi tervet. Így alakultak ki az első projektvezetési módszertanok, amelyek hatással voltak a szoftverfejlesztésre is. A kezdetben létrejövő szoftverfejlesztési módszertanok sokban hasonlítottak a korábban más iparágakban kialakult metodológiákhoz, például a mérnöki tervezés során használt megközelítés a vízesés modellhez hasonló. Ezek kezdetben jól működtek, viszont a technológia gyors fejlődése miatt egyre kevésbé voltak képesek alkalmazkodni a gyors változáshoz. Szükség mutatkozott tehát újabb és újabb módszertanok kialakítására, melyek egyre inkább a szoftver projektek egyedi tulajdonságaira vannak szabva és képesek lépést tartani a gyorsan változó technológiai környezettel és piaci versennyel.

2.1 Lineáris módszertanok

Az első módszertan - a Vízesés modell (Waterfall lifecycle) - 1970-ben jött létre, Winston W. Royce-nak köszönhetően. A modell egy lineáris folyamatot ír le, ahol a lépések szorosan egymást követik, az előző lépés kimenete szükséges a következő megkezdéséhez. Például a tervezés kimenetele egy részletes terv, mely szükséges a design kialakításához. Ugyanígy a design elkészülése szükséges ahhoz, hogy meg lehessen kezdeni a fejlesztést. Később a vízesés modell mintájára újabb lineáris módszertanok jöttek létre, mint például a V-modell, melyek most nem kerülnek kifejtésre. (M. Liviu, 2014)



1. ábra: A vízesés modell

Forrás: saját szerkesztés (M. Liviu, 2014 alapján)

Általánosságban elmondható az ilyen típusú módszertanokról, hogy jól működnek kicsi, jól definiált projekteknél, illetve ahol az elvárások és körülmények egyértelműek és nagy valószínűséggel változatlanok maradnak a projekt végéig. A legnagyobb gyengeségük a rugalmatlanság. A gyorsuló technológiai fejlődés és a piaci verseny miatt jelenleg egyre gyorsabban változnak a követelmények, a környezet és az elvárások. Emiatt az ilyen típusú módszertanokat jelenleg már csak egy-egy specifikus iparágban használják, olyan projektekhez, ahol a követelmények valószínűleg nem fognak változni. (pl.: autóiparban fejlesztett szoftverek).

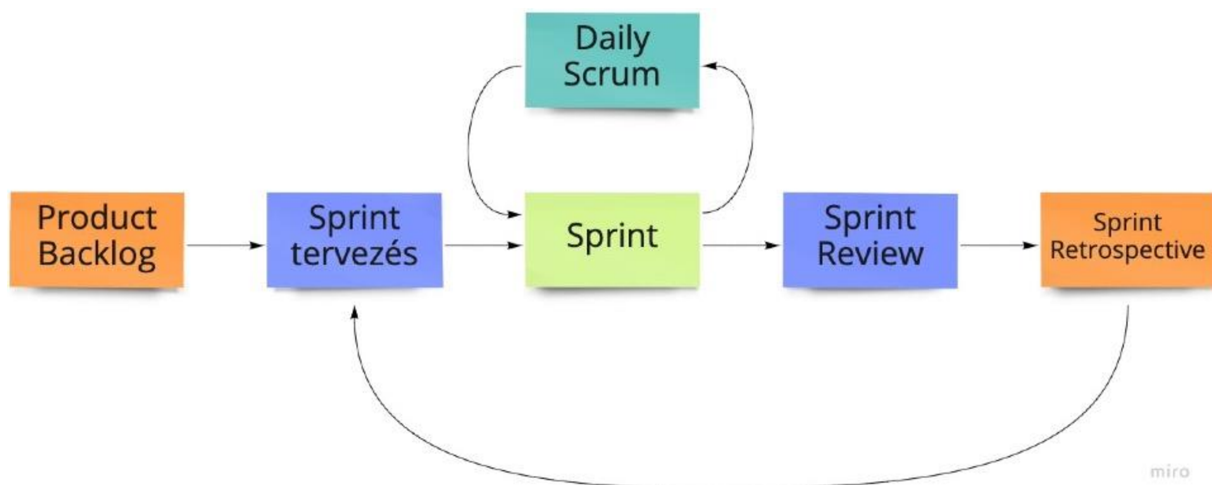
2.2 Iteratív módszertanok

Az iteratív módszertanok kifejezetten olyan projektekhez lettek kifejlesztve, ahol a követelmények nem tiszták, vagy nagy valószínűséggel változni fognak a projekt során. Adaptív képességük abból származik, hogy a szoftvert több, egymást követő ciklus alatt szállítják, melyek mindegyike egy működő verziója az alkalmazásnak. Előnyben részesítik az önszerveződő csapatokat, a visszajelzés kultúráját és a működő kód mielőbbi szállítását. Céljuk

az ügyfél elégedettség maximalizálása. Ilyen módszertanok például a Scrum, Crystal Clear és az eXtrém Programozás (XP).

2001-ben 17 szoftverfejlesztési szakértő az Agile Manifesto (magyarul Agilis Kiáltvány) nevű publikációban foglalta össze az agilis szoftverfejlesztés alapelveit, azóta az ilyen típusú módszertanokra agilisként is szoktunk hivatkozni. (K. Beck et al. 2001)

Az első, és a mai napig igen elterjedt agilis keretrendszer a Scrum. Részletes leírása, a Scrum útmutató 60 nyelven elérhető, köztük magyarul is. Az agilitás elemeit használja és az értékei megegyeznek az agilis manifesztóban foglaltakkal. A Scrum iteratív megközelítést alkalmaz: rövidebb, beláthatóbb időtartamra tervezi erőforrásainak felosztását. Kettő-négy hetes ciklusok (sprintek) követik egymást, melyekben a tevékenységek mindig ismétlődnek, a tervezéstől a szállításig. A fejlesztő csapat egy ciklus elején vállalást teszt az általuk elvégezhetőnek vélt feladatokra, amiket a ciklus végéig szállítaniuk kell. Ezeken a hosszabb ciklusokon kívül megtalálható benne egy rövidebb napi szintű iteráció is, amely a napi feladatok ismétlődését foglalja magában (daily scrum). A Scrum a ciklusokon kívül meghatározza a csapattagokat és szerepüket, ceremóniákat (megbeszélések) és a munkaanyagokat (dokumentáció). (K. Schwaber, J. Sutherland 2020)



2. ábra: Scrum modell

Forrás: Saját szerkesztés

A Scrum népszerűségét rugalmasságának köszönheti. Mivel nem tartalmaz a szoftverfejlesztésre vonatkozó technológia megkötéseket ezért előszeretettel használják más területen lévő projektek menedzselésére is.

A Scrumon kívül számos más agilis metodológia is megjelent, az egyik, a Scrumnál szigorúbb, kifejezetten informatikai alapelveket és gyakorlatot is meghatározó eXtrém programozás (eXtreme Programing – XP). Az eXtrém programozásban az iterációk rövidebbek, mindössze egy hetesek. Az XP szerint a csapat minden fejlesztési tevékenységet (analízis, design, tervezés, kódolás, tesztelés, szállítás) iterációnként újra ismétel. Így nem szükséges egy egész projektre vonatkozó terveket és designt készíteni, ezzel még rugalmasabbá válik a fejlesztés. Értékeinek tekinti a kommunikációt, a visszajelzés kultúráját, az egyszerűségre való törekvést a bátorságot és a tiszteletet. Az értékeit alapul véve pedig gyakorlati megfontolásokat is tartalmaz, négy kategóriába sorolva: visszajelzés, folyamatos fejlődés, közös megértés és a programozó jólléte (J. Shore, S. Warden 2007)

Látható, hogy itt már megjelennek kifejezetten a szoftverfejlesztés területén alkalmazható és adott esetben akár automatizálható gyakorlatok (teszt-automatizáció, folyamatos integráció), bár ezek szükségességét az eXtrém programozás nem hangsúlyozza. Továbbá nem tartalmaz olyan gyakorlatokat, melyek az élesítés, szállítás és üzemeltetés területéhez tartoznak.

2.3 DevOps

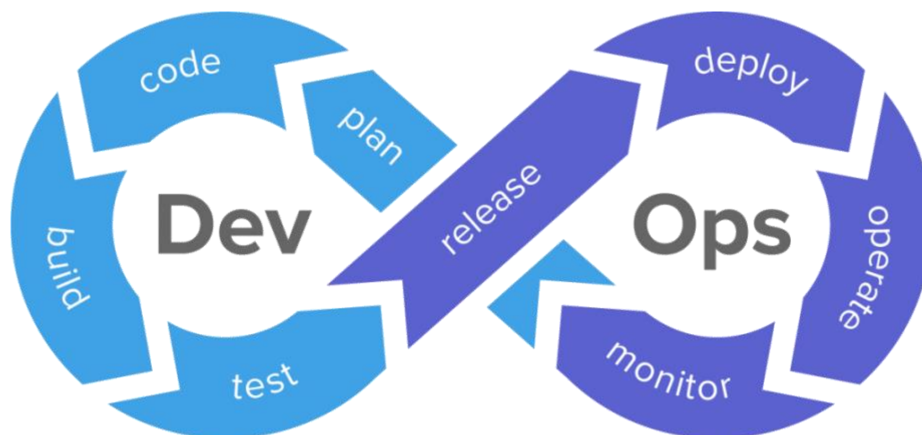
A fejlesztés (development) és üzemeltetés (operations) angol szavak összefonódásából származó DevOps kifejezés először 2009-ben jelent meg a szakirodalomban. (C. Ladas 2009) (J. Guerrero, K. Zuniga, C. Certuche and C. Pardo 2020) Napjainkban egyre több helyen találkozhatunk ezzel a szóösszetétellel, viszont mind az értelmezése, mind a gyakorlatba való átültetése rendkívül eltérő a szoftverfejlesztő cégek között.

Különböző megközelítések léteznek, több szisztematikus feltérképező tanulmány is foglalkozott már a létező értelmezések összehasonlításával és egy átfogó definíció megfogalmazásával. R. Jabbari, N. Ali, K. Petersen (2016) tanulmányában például így határozta meg a DevOps fogalmát:

„A DevOps egy olyan szoftverfejlesztési módszertan, amely megpróbálja áthidalni a távolságot a fejlesztés és az üzemeltetés területei között. Ehhez nagy hangsúlyt fektet a kommunikációra és az együttműködése, a folyamatos integrációra, minőségbiztosításra és szállításra, mindezt automatikus élesítési folyamatokkal és specifikus fejlesztői gyakorlatokkal támogatva."[1]

A definíció jól rávilágít arra, hogy a DevOps a szoftver életciklusának mindegyik szakaszára kiterjed, így nem csupán a fejlesztést, de az üzemeltetést is szabályozza. Erősen épít az agilis metodológiákra, hiszen a kollaborációt, a visszajelzés fontosságát és a szervezeti szintű kultúra kialakítását éppúgy fontosnak tartja, de több annál, mivel az agilis szemléletet kiterjeszti az üzemeltetésre is.

A fejlesztők és az üzemeltetők közötti együttműködést agilis felfogásból közelíti meg. Míg a Scrumnál egy hosszabb ciklust láthattunk, mely a fejlesztésre összpontosít és folyamatosan ismétlődik, a DevOps ezt kibővíti az üzemeltetés ciklusával. Hiszen minden egyes fejlesztési ciklust követ a szoftver egy újabb verziójának üzemeltetése. A DevOps már ezt a ciklust is magában foglalja, illetve az innen származó tapasztalatot, tudást és visszajelzéseket visszavezeti a fejlesztési ciklus tervezési szakaszába.



3. ábra: DevOps modell

Forrás: codecool.com

---

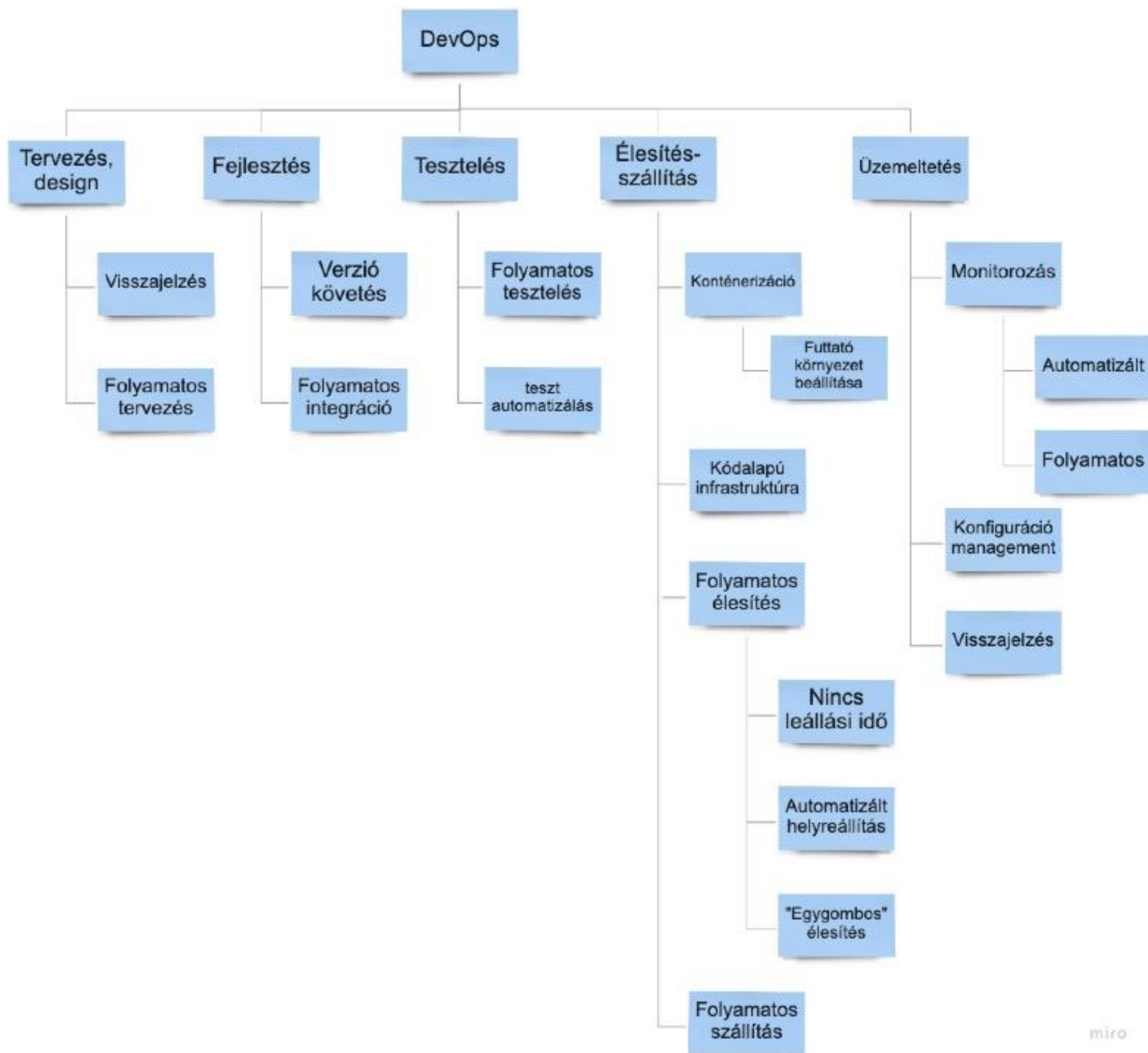
[1] "DevOps is a development methodology aimed at bridging the gap between Development (Dev) and Operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices."

Természetesen nem csak a Scrum, de a többi agilis módszertan is kiváló alap lehet, ha egy szervezet úgy dönt, hogy adaptálni szeretné a DevOps-ot. Amennyiben korábban nem agilis szemléletű volt a szervezet, nagyobb kihívások elé nézhet, viszont a sikeres adaptáció esetén agilissá is válik. Egyes esetekben az adaptáció olyan jól sikerült, hogy a fejlesztő és az üzemeltető csapat teljesen egybe olvadt, és így a két funkciót egyszerre ellátó DevOps csapatok jöttek létre. (L.E. Lwakatare, T. Kilamo, T. Karvonen et al. 2019)

A DevOps-ot gyakran hozzák összefüggésbe a „microservice architecture”[2] (MSA) és a felhő-alapú rendszerekkel. (M. Waseem, P. Liang, M. Shahin 2020) Annak ellenére, hogy az MSA valóban sok szempontból segíti a DevOps folyamatokat, meg lehet valósítani monolit architektúrában is, csak nehezebben. (M. Shahin, M. Zahedi, M. A. Babar, L. Zhu 2018) Ebben az esetben akadályok fognak felmerülni a csapat autonómiában, a gyors visszajelzésben, az automatizációban és a skálázható környezetben. Ezért ezekben az esetekben célszerű a monolit architektúrát lecserélni, mielőtt egy cég DevOps adaptálásba kezd.

Az eXtrém programozáshoz hasonlóan a DevOps konkrét informatika folyamatokat, gyakorlatokat is magában foglal, ezeket szintén kiterjeszti az üzemeltetés területére, illetve megjelenik az automatizálás is, mint elvárás. A DevOps az együttműködés kultúráját helyezi a középpontba. Ennek eléréséhez fontosak a szociális készségek és a szervezeti kultúra kialakítása: termék orientált gondolkodás, visszajelzés csapaton belül és a csapatok között, közös felelősség, folyamatos tervezés. Viszont az együttműködést sok gyakorlati koncepció és szoftveres eszköz is tudja segíteni, melyek a szoftver teljes életciklusára kiterjednek. A 4. ábra mutatja a DevOps-hoz kapcsolódó gyakorlatokat fejlesztési ciklus szerint csoportosítva.

[2] mikroszervíz alapú architektúra



4. ábra: DevOpshoz kapcsolódó gyakorlatokat fejlesztési ciklus szerint csoportosítva
Forrás: Saját szerkesztés (a szakirodalmakban említett gyakorlatok csoportosítása)

Az élesítés és a szállítás a fejlesztés és az üzemeltetés határterülete, ezért ahogyan az ábrán is látható, a legtöbb DevOps gyakorlat ezt a területet célozza meg. Ezek közül a gyakorlati koncepciók közül valamilyen módon mindegyiknek szerepe van az automatizált folyamatok kialakításában, ebből is látszik, hogy mennyire meghatározó része ez a DevOps adaptációnak. Sikeres adaptáció eseten ezeket az automatizált lépések nagy részét egy lineáris rendszerré alakítják, ezt szokták CI/CD pipeline[3]-nak nevezni. „Humble and Farley szerint:

[3] continuous integration / continuous delivery, magyarul: folyamatos integráció / folyamatos szállítási vezeték

„A CI/CD pipeline a teljes szoftverfolyamat automatizált megnyilvánulása, amely a szoftverváltoztatások élesítésének minden szakaszát tartalmazza a verziókezeléstől egészen addig, amíg azok láthatóvá nem válnak a végfelhasználók számára."[4]

[4] "The deployment pipeline, is an automated manifestation of the entire software process constituting to all stages of getting software changes from version control until they are visible to end-users"

3. Az automatizálás lépéseinek és eszközeinek bemutatása

3.1 Verzió követés

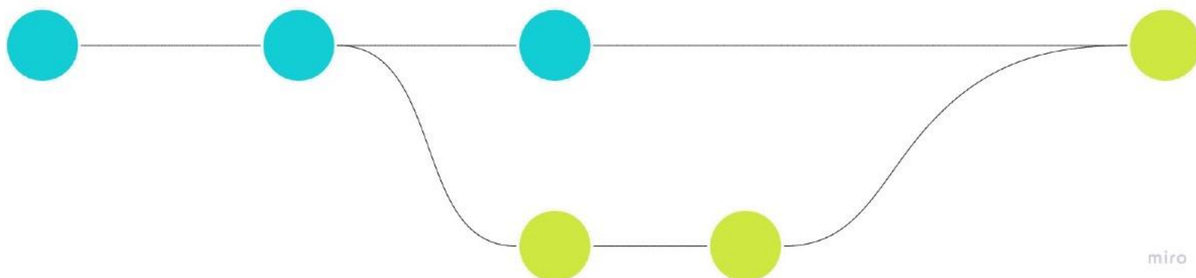
A verzió követő rendszerek nélkülözhetetlen részei a modern szoftverfejlesztésnek. A verziókövetés (vagy forráskövetés) célja a szoftver kód (vagy forrás kód) változásainak követése és menedzselése. A verzió követő rendszerek szoftveres eszközök, amik segítik a csapatokat, hogy menedzselni tudják a szoftverkódjuk változásait és így gyorsabban, nagyobb összhangban tudják végezni a munkájukat. Különösen hasznos a DevOps csapatok számára, mivel csökkenti a fejlesztési időt, segít a folyamatos integráció megvalósításában és növeli a sikeres élesítések számát.

A verzió követő rendszer nyomon követi a forráskód minden egyes változását, és az azt készítő fejlesztő adatait egy speciális adatbázisban, ezzel segít megvédeni a szoftverkódot az emberi hibáktól. A fejlesztő csapatok folyamatosan új kódot adnak hozzá a kódbázishoz és változtatják a meglévőt. Ezt általában párhuzamosan csinálják a kód különböző részein. A kód egyik részén történt módosítás lehet, hogy nem kompatibilis egy másik helyen, más valaki által írt változtatással. A verzió követő rendszer segít feloldani az ebből eredő ellentmondásokat, vagyis segíti az integrációt. Minden változtatás hozhat nem várt hibákat (bugokat), a verziókövetés segít a fejlesztőknek összehasonlítani a kódot egy régebbi (működő) verzióval, ami segíthet megoldani egy-egy problémát. Amennyiben nem sikerül máshogy megoldani, vissza is lehet állítani egy korábbi verziót. (www.atlassian.com)

A Stack Overflow 2021-ben készült kutatásából kiderül, hogy napjainkban az egyik legnépszerűbb verziókövető rendszer a Git. (insights.stackoverflow.com) A legtöbb szoftverfejlesztő csapat és cég ezt használja. Ingyenes és nyílt forráskódú, valamint platform független, minden operációs rendszerre elérhető. A Git egy osztott rendszer, általában van egy központi, hálózaton keresztül elérhető szerver, amin raktárakban (repository) tárolja a forrás kódokat. Innen minden fejlesztő gépére készít egy másolatot a teljes verzió történetből. Ez nagyban növeli a biztonságot, hiszen, ha a központi szerveren adatvesztés történik, akkor is vissza lehet állítani a kódot valamelyik fejlesztő gépéről. A központi szerver lehet saját hálózaton működtetett (on-premise), mint például a Gitlab, vagy lehet egy felhő szolgáltató által fenntartott, az interneten keresztül elérhető szerver, például: Github, Bitbucket.

A verzió követő rendszernek fontos szerepe van a DevOps automatizációban is. A CI/CD pipeline kezdőpontja minden esetben egy kódváltoztatás, amit a fejlesztő csapat szeretne élesíteni. Ehhez a központi szerveren lévő repository-ba küldik az elvégzett kódmódosításokat megfelelő verzióját, amely elindítja a folyamatot. Ahhoz, hogy ez megtörténhessen, a szervernek képesnek kell lennie integrálódni más rendszerekkel, mivel a teljes élesítési folyamatot ki kell szolgálni a forráskóddal. A pipeline-ban résztvevő rendszerek nagy részének szüksége van arra, hogy le tudja másolni a forráskódot a Git szerverről. Éppen emiatt a Git szerverek mindegyike sok integrálódási pontot biztosít (pl.: API-n keresztüli elérés, SSH-val való kapcsolódás stb.), sőt a legtöbb már saját folyamatos integráció megoldást is kínál (pl.: Github Actions).

Ahhoz, hogy a fejlesztők összhangban tudjanak dolgozni, különböző munkamenet koncepciók (workflow) alakultak ki. Az egyik népszerű koncepció az elágazás (branching). Ez alapján mikor egy fejlesztő új funkció megvalósításába kezd, a főágból (ami egy élesített verzió) egy mellékágat hoz létre a Git segítségével. A változtatásait ezen a mellékágon követi, majd mikor készen van, egy kérést indít a kódváltoztatások érvényesítésére a főágban. Így képesek a fejlesztők egyszerre több mellékágon párhuzamosan is dolgozni. A CI/CD pipeline egy része, a tesztek futtatásáig ilyenkor elindul a mellékágakra is, így értesítve a fejlesztőt arról, ha az integráció nem lesz sikeres. Viszont a teljes, élesítést is magában foglaló pipeline csak a főágra történő bekerülést követően fut le. Ennek a munkamenetnek megvan az a veszélye, hogy a főágba hamarabb integrált változtatások befolyásolhatják a többi ágon dolgozó fejlesztő változtatásait, így azokat folyamatosan újra és újra összhangba kell hozni a főággal.



5. ábra: Elágazásos munkamenet

Forrás: Saját szerkesztés (atlassian.com alapján)

Ezt a problémát próbálják sokan egy olyan munkamenettel orvosolni, ahol nincsenek (vagy csak nagyon ritkán vannak) mellékágak. Minden fejlesztő a főágon dolgozik, és törekszik arra, hogy csak olyan változtatásokat tegyen közzé, amik elemiek és működnek. Így gyakrabban történik az integráció és a CI/CD pipeline-nak csak a főágon kell futnia, így minden kódváltoztatást élesít, amennyiben a tesztek sikeresen lefutnak. Hátránya, hogy mikor valaki hibázik, és „eltöri" a teszteket, addig a vezeték „eldugul" és a többi fejlesztő sem tud változtatásokat közzétenni. Így nő a felelősség a fejlesztőkön, viszont ez elősegíti az eredményesebb munkát, az elemibb változtatásokat, a naponta akár többszöri élesítést.

3.2 Tesztautomatizálás

A tesztautomatizálás a szoftver tesztelés egyik technikája, amivel automatikusan lehet a valós és elvárt eredményeket összehasonlítani. Célja a tesztelés pontosságának, sebességének és hatékonyságának növelése a manuálisan csak nehezen vagy lassan végrehajtható tesztek számának minimalizálásával. A tesztelők – de sok esetben a fejlesztők is – teszt parancsokat írnak (általában a forráskód programnyelvével azonos nyelven), melyeket tesztautomatizáló eszközökkel futtatnak. Ennek köszönhetően az ismétlődő tesztek pontosabbak, gyorsabbak és az emberi hibától mentesek lesznek.

A tesztelés a minőségbiztosítás része, így a tesztek automatizálásának köszönhetően a szoftver minősége is megnövekedhet. Lehetővé teszi a tesztelők számára, hogy az ismétlődő tesztek elvégzése helyett új tesztesetek kidolgozásával töltsék az idejüket. Ezáltal képes növelni a tesztlefedettséget, ami egy fontos mutató minőségbiztosítási szempontból. Olyan nagyszámú teszt elvégzését teszi lehetővé percek alatt, melyeket akár napokba telne kézzel elvégezni. Így a teszteredmények és a visszajelzések sokkal gyorsabban visszajutnak a fejlesztő csapathoz. A hibák mihamarabbi megtalálása – még a fejlesztési szakaszban –, bizonyítottan pénzt spórol a vállalkozásnak, illetve növeli az ügyfelek bizalmát a termékben.

A teszt automatizáció fontosságát jól jelzi, hogy különböző szoftver fejlesztési technikák alakultak ki a mentén, mely során a teszt kódot a szoftver kódjával egyidőben, vagy előbb írják meg. Ilyen technika a Teszt Vezérelt Fejlesztés (Test-Driven Development – TDD) és a Viselkedés Vezérelt Fejlesztés (Behaviour-Driven Development – BDD). Bizonyos

szoftverfejlesztési módszertanok ezek használatát is előírhatják (pl.: eXtrém programozásban a TDD).

A folyamatos integráció megvalósításához elengedhetetlen lépés a tesztek automatizálása. A fejlesztőknek gyorsan meg kell tudniuk bizonyosodni róla, hogy a kódváltoztatás, amit a verziókövető rendszerbe küldtek, nem okozott-e hibát a szoftver bármelyik részében. Éppen ezért a CI/CD pipeline fontos lépése az automatizált tesztek futtatása. A szoftver tesztelésnek különböző típusai és szintjei ismertek (pl.: unit teszt, integrációs teszt stb.), ezek közül számos automatizálható és a CI/CD pipeline-ban egymással párhuzamosan futtatható, aminek köszönhetően a tesztelés sebessége tovább nőhet. Ebben a lépésben az alkalmazás tesztjeinek futtatásán kívül, lehetőség van kód minőség ellenőrző szoftverek futtatására is. Amennyiben a pipeline, teszt futtatási lépése közben valamelyik teszt hibát jelez, a folyamatot meg kell szakítani, és értesíteni kell a fejlesztőket, az élesítés további lépéseit pedig nem szabad elvégezni. A fejlesztők értesítése a CI/CD szoftver, illetve valamelyik a fejlesztő csapat által használt kommunikációs eszköz (Slack, email) összekapcsolásával történik.

3.3 Konténerizáció

A virtuális gépek használata mára esszenciális része lett a modern informatikai rendszerek infrastruktúrájának. A legtöbb esetben a fizikai gépeken úgynevezett. „hypervisor"-ok futnak. Ezek olyan szoftverek, amik virtuális gépeket hoznak létre és futtatnak. Így a rendszer adminisztrátorok képesek optimalizálni a rendszer fizikai erőforrásait, illetve elkülöníteni a rajtuk futó alkalmazásokat. Viszont mivel a hypervisor minden egyes virtuális gépnek külön kernelt futtat a fizikai gépen, az alkalmazások és folyamatok elkülönítése erőforrás igényes művelet. A konténer alapú virtualizáció ezt a problémát hivatott megoldani. A Linux Containers (LXC) egy kernel technológia, amely képes a folyamatokat a saját elkülönített környezetükben (konténer) futtatni. A Docker pedig egy olyan szoftver, ami képes egy alkalmazást és az összes függőségét egy reprodukálható „kép"-be (docker image) csomagolni és ezeket konténerekben futtatni. (T. Scheepers 2014)

6. ábra: Virtuális gép és Konténerizált infrasturktúra
Forrás: Saját szerkesztés (docker.com alapján)

Konténerizációnak nevezzük, mikor egy alkalmazást ilyen módon csomagolunk be, hogy aztán különböző számítógépeken platform függetlenül tudjuk futtatni (pl.: a fejlesztők saját gépe, a CI/CD pipeline-ban a tesztek futtatására használt gép vagy a szerver, ahol az éles környezet fut stb.). A konténerizáció lehetőséget ad arra, hogy a fejlesztők maguk tudják menedzselni az applikációjuk futtató környezetét és függőségeit. Így a futtató környezet bármikor, bármilyen számítógépen (ahol van konténerizációs szoftver) reprodukálhatóvá válik és a fejlesztők sokkal hamarabb értesülnek az ebből adódó hibákról, valamint így sok feladatot át tudnak venni az üzemeltetéstől.

Ezáltal a konténerizáció, része a kód alapú infrastruktúra (infrastructure as a code) DevOps gyakorlat megvalósításának. A gyakorlat lényege, az alkalmazás futtatásához szükséges infrastruktúra leírása fájlokban és a verziókövetőben tárolása. A konténerizáció során is ez történik, viszont ez csak a futtató környezet leírására alkalmas, a teljes infrastruktúra leírása később még kifejtésre kerül és az esettanulmányok példát is fognak rá hozni.

A CI/CD pipeline készítésekor elengedhetetlen lépés az alkalmazás konténerizációja, hiszen a legtöbb CI/CD szoftver ezt a technológiát használja az alkalmazás tesztjeinek futtatásakor,

illetve az élesítés is sok esetben egy felhő alapú konténer orkesztrációs[5] vagy arra épülő alkalmazás platformon történik. Ez utóbbi, habár nem feltétele a sikeres DevOps adaptációnak, - történhet az élesítés saját szerverre is- viszont nélküle az automatikus és a leállási idő nélküli élesítést nagyobb kihívás megvalósítani, illetve a konténerizációból származó platform függetlenség is elveszik.

Az alkalmazás konténerizációját (vagy dockerizációját) csak egyszer kell elvégeznünk manuálisan. Viszont ahogyan egy alkalmazás fejlődik, úgy a függőségei is változhatnak és frissülnek is. Emiatt a konténerekben is szükség lehet egyes függőségek frissítésére. A CI/CD pipeline-ban külön lépés a konténerek megépítése és eltárolása egy felhő alapú konténer jegyzékben (container registry). Jó gyakorlat, ha a teszteket, ugyanazon vagy nagyon hasonló konténerben futtatjuk, mint amiben az alkalmazás is futni fog. Ebben az esetben ez a lépés előrébb kerül a pipeline-ban és még a tesztek futtatása előtt történik meg a konténerek építése. Abban az esetben, mikor hibára fut, ugyanúgy kell eljárni, mint a többi lépésnél, meg kell szakítani a folyamatot és értesíteni kell a fejlesztő csapatot.

3.4 Automatikus élesítés és szállítás

A konténerizáció önmagában számos üzemeltetéssel kapcsolatos kérdésre nem nyújt megoldást. A konténer orkesztrációs platformokat (pl.: Kubernetes, Docker Swarm, Red Hat, OpenShift) az olyan feladatok ellátására hozták létre, mint magas rendelkezésre állás, felhasználó kezelés, terheléseloszlás, kifinomult hálózatkezelés és integráció. Ezek a szoftverek képesek nagy számú konténerre épülő infrastruktúra létrehozására, valamint a benne szereplő konténerek életciklusának teljes kezelésére, a konténerekere épülő szolgáltatások felépítésére és a működésük támogatására (pl.: hálózat, tárhely stb.) Ilyen eszköz használatára mindenképpen szükség van, amennyiben egy alkalmazás mikroszervíz architektúrában van fejlesztve, azonban monolit rendszerek esetében is jól működnek és megoldást kínálnak a skálázhatóság nehézségeire, a folyamatos, leállás nélkül történő élesítésre és az automatizált helyreállításra.

---

[5] A konténerizált szoftverek futtatásához kapcsolódó üzemeltetési feladatok automatizálása. pl.: élesítés, skálázás, hálózati beállítás

A CI/CD pipeline utolsó lépése az élesítés és a szállítás. A gyakorlatban azonban általában a pipeline nem az igazi (production) környezetben élesíti az alkalmazást, - így szállítás nem történik - hanem egy különálló tesztelésre alkalmas (staging) környezetben. Ez a lehető legjobban meg kell egyezzen az éles környezettel és a célja, hogy kiderülhessenek olyan hibák, melyeket az automatikus tesztek nem mutattak meg. Így a valódi szállítás általában egy különálló folyamat, mely szintén automatizált és a dolga csak annyi, hogy a staging környezetben lévővel megegyező konténereket indít az élesre. A folyamat általában kézzel indítható, egy gombnyomásra innen kapta a nevét az „egy gombos élesítés" nevű DevOps gyakorlat. Az élesítés és szállítás során akár éles, akár teszt környezetről van szó, az alkalmazás nem állhat le egy másodpercre sem, illetve, ha valamiért az új verzió nem tud sikeresen elindulni, akkor az előző, működő verziónak automatikusan vissza kell állnia. Ezt a konténer orkesztrációs platformok olyan módon oldják meg, hogy a régi verziót futtató konténereket csak az után állítják le, hogy az újak már sikeresen elindultak, amennyiben az újak indulása közben hiba történik, akkor nem is állítja le a régieket.

3.5 Monitorozás

A monitorozás egy rendszer állapotának, működőképességének ellenőrzését jelenti. Ez irányulhat különböző funkciók működésére, teljesítményre (sebesség, válaszidő stb.), erőforrásokra (tárhely, memória) és a hálózat működésére. Egy rendszer folyamatos fejlesztése és szállítása potenciálisan megnöveli a hibák számát, illetve a rendszer leállásának kockázatát. Ez rákényszeríti a DevOps-ot gyakorló vállalatokat, hogy több hangsúlyt fektessenek a naplózásra (logging) és monitorozásra. M. Shahin, M. Zahedi, M. A. Babar, L. Zhu (2018)

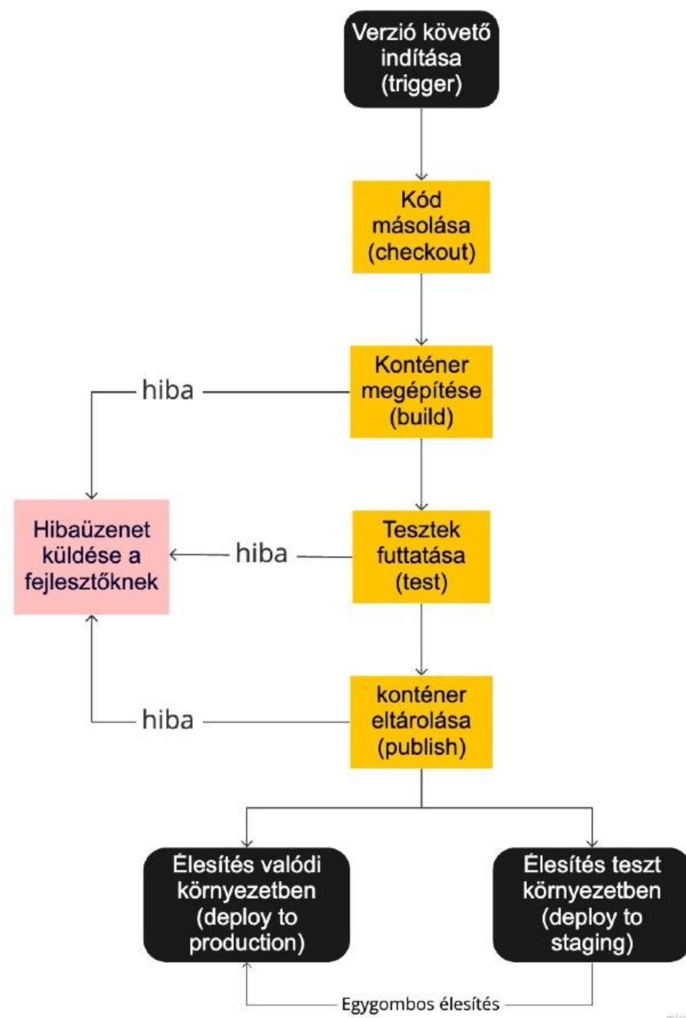
Mivel a működő rendszer állapotát folyamatosan figyelni kell, a monitorozás nem egy esemény által kiváltott folyamat, hanem egy folyamatos tevékenység. Emiatt nem is része a CI/CD pipeline-nak, szerepe nem az élesítésben és szállításban, hanem azok után a rendszer folyamatos ellenőrzésében van. Elengedhetetlen az automatizálása, hiszen anélkül aligha lehetne folyamatosan kivitelezni. A mikroszervíz architektúra és a felhő alapú rendszerek – melyek rendkívül elterjedtek a DevOps szemléletű vállalatoknál - kihívást jelentenek monitorozás szempontjából, mivel így nemcsak egy monolit rendszert kell megfigyelni, hanem sok különálló szolgáltatást, illetve azok kapcsolatát is. (M. Waseem, P. Liang, M. Shahin, 2020)

Megvalósítását több szoftveres eszköz segíti (pl.: Grafana, Loggly, OP5), melyek különböző tulajdonságok ellenőrzésére szolgálnak, így egy komplex rendszer esetében ezek közül akár többet is használni kell. A megfigyelésen kívül az is fontos, hogy az azonosított hibák minél hamarabb eljussanak a fejlesztőkhöz és az üzemeltetőkhöz, hogy gyorsan javítani tudják azokat. Emiatt a monitorozás elengedhetetlen lépése a riasztás (alerting), melyhez szintén rendelkezésre állnak integrált eszközök (pl. PagerDuty). Mivel a monitorozás nem része a CI/CD pipeline-nak, ezért a gyakorlatban nem fogom bemutatni.

3.6 CI/CD pipeline

Végigtekintettük tehát a sikeres DevOps automatizáció elengedhetetlen lépéseit és eszközeit. Ezek alapján fel lehet rajzolni a CI/CD pipeline folyamatábráját (7. ábra), amely a 4. ábrán vázolt DevOps gyakorlatok közül a fejlesztéshez, a teszteléshez és az élesítés-szállításhoz tartozó, automatizálható gyakorlatokat foglalja magában.

Egy ilyen CI/CD pipeline megvalósításához több szoftveres eszközt is használni kell. A legminimálisabb esetben ez három eszköz kiválasztását és összehangolását jelenti. Az egyik egy verziókövető rendszer, amely tárolja a kódot, képes indító jelzést adni a változtatásokról, ezzel elindítva a folyamatot, illetve a többi eszköz számára elérhetővé teszi a kódot (pl.: Github, GitLab, BitBucket). Mellette szükség van egy folyamatos integrációs eszközre (CI tool), amely a 7.ábrán sárgával jelölt lépéseket futtatja a kód másolásától a konténer eltárolásáig (pl.: CircleCi, Codeship, Jenkins). Szintén a CI tool indítja el az élesítés folyamatát, amely azonban már egy másik platformon megy majd végbe. A harmadik eszköz egy élesítési felület (deployment platform), ahol a működő alkalmazás fut, ez általában egy konténer orkesztrációs platform vagy egy arra épülő alkalmazás üzemeltetési felület (pl.: Heroku, Google App Engine).



7. ábra: CI/CD pipeline folyamatábrája

Forrás: Saját szerkesztés

Mindhárom típusú szoftverből több is elérhető a piacon, ezért nagy kihívást jelenthet kiválasztani a számunkra és projektünknek a legmegfelelőbbet. Számba kell venni többek között a teljesítményüket, a függőségeiket, az áraikat, a skálázhatóságukat, az egymáshoz és a meglévő rendszereinkhez kapcsolódó integrációs pontokat, valamint azt is, hogy mennyire illeszkednek az architektúrához és a megvalósítani kívánt célhoz.

A dolgozat második felében kiválasztok, majd bemutatok néhány, jelenleg az iparban előszeretettel használt eszközt. Ezután elkészítek egy minimális funkcióval rendelkező web alkalmazást és hozzá négy különböző CI/CD pipeline-t a kiválasztott eszközökkel. Célom az összehasonlíthatóság segítése és a teljesítményük mérése. A használt eszközök online

dokumentációi és az esettanulmány kódját tartalmazó Github repository linkje megtalálható a mellékletben. Ez utóbbi mindig a legfrissebb állapotot mutatja, viszont az egyes esettanulmányok elkészüléséhez szükséges változtatások megtekinthetők a verzió történetben.

4. Esettanulmányok elkészítése

4.1 Tervezés

A CI/CD pipeline létrehozásához szükséges szolgáltatások lehetnek saját magunk által üzemeltetett szervereken (on-premise), viszont ez egyre kevésbé népszerű megoldás, mert a szükséges számítógépek beszerzése és üzemeltetése rendkívül költséges. A felhő szolgáltatásoknak ezzel szemben megvan az az előnye, hogy havi vagy éves díj ellenében egy teljes funkcionalitással rendelkező rendszert tudunk használni és az üzemeltetésre sem kell más erőforrást áldozni. A számlázás gyakran használati alapon történik, így percre vagy tárhelyre pontosan annyit kell fizetni, amennyit használtuk a felhő által biztosított virtuális erőforrásokat. Ez a kis és közepes méretű projektekhez teljesen alkalmas az igények kiszolgálására, de nagyobb projektekhez is megfelelő lehet. A legtöbb felhőszolgáltatás kínál ingyenes kipróbálási lehetőséget, ami nem elég ugyan ahhoz, hogy naponta többször élesítsünk, viszont meg lehet állapítani belőle, hogy az adott szolgáltatás megfelel-e az igényeinknek, illetve a jövőbeni költségek kiszámítására is alkalmas. Az esettanulmányok elkészítése során kizárólag a felhő alapú szolgáltatásokra fogok koncentrálni.

Jelenleg a legnépszerűbb verziókövető rendszer a Git, melyhez több felhőszolgáltató is kínál központi szervert. Ezek nagyrésze hasonló működéssel bír és az alapvető funkciók nagy része ingyenesen használható. Az egyik legelterjedtebb ilyen felhő szolgáltatás a Github, így én is ezt fogom használni. Minden piacon lévő CI eszköz és alkalmazás platform biztosít Github integrációt a folyamatok automatikus indításához és a kód másolásához. Minden változtatást a főágon fogok csinálni, branch-elés nélkül. Egy új változtatás beküldése Githubra fogja elindítani a CI/CD pipeline-ok futását.

Az élesítés és üzemeltetés bemutatására két felületet választottam ki, amelyek sok mindenben különböznek egymástól, ezek a Heroku és a Kubernetes. A Heroku egy PaaS[6] felhő szolgáltatás, mely nyolc különböző programnyelvben írott webes alkalmazás futtatását támogatja. Népszerűségét annak köszönheti, hogy saját konténerizációs és orkesztrációs rendszerrel rendelkezik, így az alkalmazás konténerizációját nem szükséges a fejlesztőknek elvégezni, ez

---

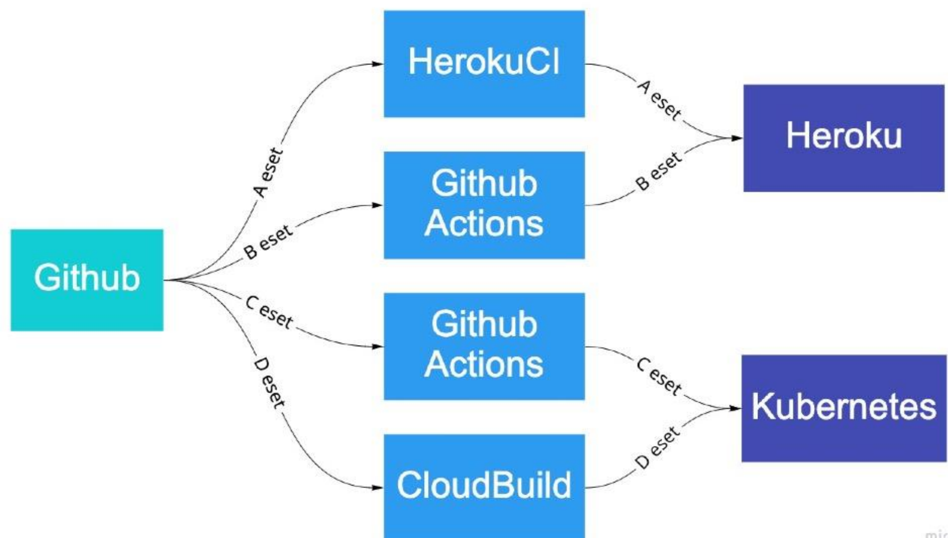
[6] platform as a service

pedig sok időt tud megspórolni. Üzemeltetési szempontból sem igényel bonyolult konfigurációt, csak a megfelelő teljesítményű erőforrás csomagot kell kiválasztanunk a négy felkínált lehetőségből. Kisebb projekteknél tökéletes választás lehet, viszont bonyolultabb rendszerek esetében, ahol specifikusabb elvárásokat kell az üzemeltetés során teljesíteni vagy az erőforráselosztást pontosabban kell az alkalmazás igényeihez szabni, ennél rugalmasabb konténer orkesztrációs platform szükséges.

Az első ilyen platform a Kubernetes volt, amelyet eredetileg a Google fejlesztett, azóta azonban nyílt forráskódúvá tett, így több felhő szolgáltatónál is elérhető (Amazon, Microsoft, RedHat Openshift). Ezek közül a Google Felhő Platformon elérhető szolgáltatást fogom használni. Mivel, ez egy SaaS[7] szolgáltatás, ezért csak egy futtató környezetet biztosít, az alkalmazásokat és adatbázisok kezeléséről magunknak kell gondoskodnunk (egy PaaS szolgáltató, mint a Heroku ezekről is gondoskodik helyettünk). Ez nagyobb szabadságot nyújt, viszont bonyolultabb a használata. Most, hogy már tudjuk, honnan kell élesíteni a kódot (Github) és azt is, hogy hova (Heroku vagy Kubernetes), következhet a megfelelő folyamatos integráció szolgáltatások kiválasztása.

A HerokuCI a Heroku platform saját folyamatos integrációs megoldása, amit a Herokura történő élesítéshez egyszerű használata miatt érdemes kipróbálni. A Herokun kívül a Github is nyújt saját folyamatos integrációs eszköz, ez a Github Actions. Működési elve és konfigurációja rendkívül hasonlít a többi külső féltől származó CI eszközhöz (pl.: CircleCI, Codeship). Ezekhez hasonlóan támogatja az élesítést a legtöbb felületre, így ezt ki tudom próbálni mind Herokura mind Kubernetesre való élesítéshez. Google Felhő Platformon pedig elérhető a Google saját CI eszköze, a Cloud Build, amit a Kubernetesre történő élesítéshez lehet a legjobban használni.

[7] System as a Service



8.ábra: A négy esettanulmány eszközeinek szemléltetése

Forrás: Saját szerkesztés

Ahhoz, hogy a folyamatos integrációt és élesítés meg tudjuk vizsgálni, szükség lesz egy minimális működéssel rendelkező webes alkalmazásra. Az alkalmazásnak egyetlen API végpontja lesz (/healthcheck), mely egy JSON formátumú üzenettel fog válaszolni a beérkező kérésekre. Egy ilyen végpont gyakori eleme a mikroszervíz architektúrával készülő alkalmazásoknak, elsősorba a monitorozásban van szerepe, a monitorozó rendszer ugyanis ennek a végpontnak a hívásával lesz képes megállapítani, hogy a rendszer valóban működik-e. A webes alkalmazás elkészítéséhez Python programozási nyelvet fogok használni és a hozzá elérhető webes keretrendszert, a Flask-et. A fejlesztést TDD-alapon fogom végezni, vagyis előbb a teszteket fogom megírni és csak azután a végpontot megvalósító kódot. Így rögtön automatikusan futtatható tesztek is létre fognak jönni, melyeket be lehet építeni a CI/CD pipeline-ba.

4.2 Alkalmazás elkészítése

Verzió követést a projekt legelejétől kezdve érdemes használni, így az első lépés, hogy a már meglévő Github felhasználómmal létrehozok egy új raktárat (repository). Nyilvánosan elérhetővé teszem, mivel nem fog titkos adatot tartalmazni.

A létrejött üres repository-t lemásolom a számítógépemre. Mivel a Herokura való élesítésnél nem szükséges a konténerizáció, ezért ezt a lépést csak később fogom kivitelezni. Konténerizálás nélkül viszont kénytelen vagyok telepíteni a számítógépemre a fejlesztéshez szükséges környezetet. A python programozási nyelv telepítő csomagja elérhető a honlapjukról, ezután a használt csomagok (Flask, gunicorn, pytest) telepítése a python saját csomag kezelő moduljával a „pip"-el könnyen elvégezhető. A Flask egy egyszerű web alkalmazás keretrendszer, a gunicorn pedig a futtatáshoz szükséges szervert biztosítja. A pytest nevű csomag a tesztautomatizálást teszi lehetővé, alkalmas egység szintű és integrációs tesztek megírásához is.

Két mappát hozok létre a projekten belül, a „healthcheck_api" fogja tartalmazni az alkalmazást megvalósító kódot, a másik „tests" könyvtár pedig az automata tesztek kódját. Először három tesztesetet írok: az első le ellenőrzi, hogy az alkalmazás megfelelően el tud-e indulni. A másik kettő pedig egy kérést küld a kezdőoldalra, illetve a „/healthcheck" végpontra és a „{success: true}" üzenetet várja el válaszul. Miután a megírtam a teszteket, „pytest" parancs kiadásával tudom őket lefuttatni. A tesztek alkalmazás kód hiányában hibát jeleznek, ezért a tesztvezérelt fejlesztés következő lépése a teszteket kielégítő alkalmazás kód megírása a „healthcheck_api" nevű mappában.

A python moduláris felépítése miatt minden mappában létre kell hozni egy „__init__.py" nevű fájlt, ez jelzi a programnyelvben, hogy egy modulról van szó. Mivel ez az alkalmazás minimális, összesen húsz sor kódból megvalósítható, ezért nem hozok létre más fájlokat az alkalmazás könyvtárába, hanem ebbe a fájlba írom a kódot. A fejlesztés közben akár többször is lefuttathatom a teszteket, egészen addig, amíg mindegyik át nem megy, ami a tesztvezérelt fejlesztés egyik előnye. Ezzel el is készült a minimális web alkalmazás, amelyhez a CI/CD pipeline-okat készíteni fogom.

4.3 HerokuCI és Heroku platform (A eset)

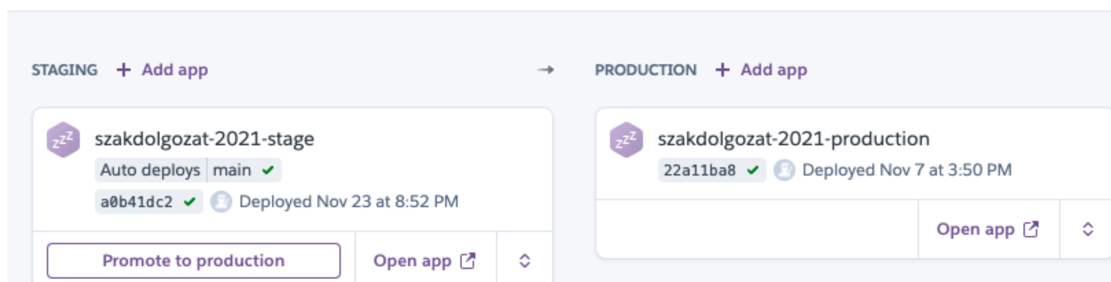
A Heroku webes felületén (regisztráció után) néhány kattintással létre tudok hozni egy projektet. Ezt – továbbra is a webes felületen maradva – a Heroku-Github integráció miatt könnyedén össze lehet kötni a Githubon tárolt alkalmazás kóddal, illetve a főág automatikus élesítését is be lehet állítani. Utóbbinál ügyelni kell rá, hogy az élesítés csak a sikeres

tesztfuttatások esetén történjen meg. Ezután néhány kattintással be lehet kapcsolni a HerokuCI funkciót is, illetve el lehet különíteni egy staging és egy production környezetet.

Ahhoz, hogy a Heroku saját konténerizációs folyamata meg tudjon valósulni, a dokumentáció alapján két fájlt kell elhelyeznem a projektemben. Az egyik a python csomagfüggőségeket leíró „requirements.txt" nevű fájl. Ebbe kerülnek a már korábban említett Flask és gunicorn nevű csomagok. A másik szükséges fájl neve „Procfile", ami azt írja le a Heroku számára, hogy milyen paranccsal kell elindítani az alkalmazás folyamatait, jelen esetben csak egy webszervert. Ezen kívül a HerokuCI számára létre kell hozni egy „app.json" nevű fájlt, ami a HerokuCI beállításait, konfigurációját tartalmazza, például a teszt futtatásához szükséges parancsot. A „reguirements-test.txt" fájlban fel kell sorolni azokat a függőségeket, amelyek csak a tesztek futtatásához szükségesek (esetünkben a pytest csomag).

Miután létrehoztam a fájlokat és a verziókövetőben elmentettem, beküldöm Githubra. A Heroku webes felületén nyomon lehet követni a folyamatokat és kimeneteiket. Először elindul a HerokuCI, benne a tesztek futtatásával. Ezt követően pedig elindul az élesítés a staging környezetben. A Heroku automatikusan felismeri, hogy python applikációt szeretnék élesíteni, a megadott fájlok segítségével elvégzi az applikáció konténerizálását, majd automatikusan élesíti azt. A https://szakdolgozat-2021-stage.herokuapp.com/ címen le is tudom ellenőrizni, hogy tényleg működik-e az alkalmazásom. A Heroku webes felületén ezután aktívvá válik egy gomb „promote to production"[8] felirattal, aminek megnyomására megtörténik az élesítés a valódi környezetben is. A 9. ábrán látható a Heroku webes felülete, benne a staging és production alkalmazásokkal. Látható, hogy a staging környezethez be van állítva az automatikus élesítés („Auto deploys main") és az egygombos élesítéshez használt gomb.

[8] éles környezetbe másolás



9. ábra: Heroku élesítési felülete

Forrás: Képernyő fotó a webes felületről

Ez a folyamat egy CI/CD pipeline definíciójának és lépéseinek eleget tesz, viszont mégis eltér a megszokottól, mert a lépéseket nem kellett egy fájlban rögzítenem, amit felolvas egy folyamatos integrációs szoftver. Ehelyett a lépéseket a webes felületen való beállításokkal és több különböző fájl létrehozásával tudtam megvalósítani. Futás közben nem egy ablakban látom a teljes folyamatot lépésenként, hanem külön el kell navigáljak a tesztek futtatásához, illetve az alkalmazás építéséhez, élesítéséhez. A megoldásnak az az előnye, hogy gyorsan kivitelezhető, nem szükséges az alkalmazás konténerizációját elvégezni, és a lépéseket sem kell leírni. A tesztek futtatása és az élesítés ugyanazon a webes felületen beállítható és kezelhető. Ezeken kívül csak néhány sor konfigurációt kell megadni, igaz, ezt négy különböző fájlban.

4.4 Github Actions és Heroku platform (B eset)

A Github Actions a Github saját automatizációs platformja. A feladatok elvégzéséhez egy virtuális gépet biztosít, amely a három legnépszerűbb operációs rendszert támogatja (Ubuntu, Windows és macOS). A virtuális gépein a fejlesztéshez, teszteléshez és élesítéshez gyakran használt szoftverek is előre fel vannak telepítve (pl.: Heroku CLI, Docker stb.), de az automatizálás lépéseként további szoftverek is telepíthetőek. A CI/CD pipeline lépéseit egy vagy több ilyen virtuális környezetben lehet futtatni. Elérhetőek továbbá olyan előre megírt lépés-gyűjtemények (action), amelyek a gyakran előforduló feladatokat elvégzik, így nekünk már nem kell őket implementálni: ilyen például a kód lemásolása, vagy a python megfelelő verziójának telepítése.

A Heroku platform – saját konténerizációs megoldása mellett – támogatja a docker konténerek futtatását is. Így a Docker használatából eredő előnyöket akkor is ki tudjuk használni, ha

Herokun szeretnénk élesíteni az alkalmazásunkat. Míg a HerokuCI nem támogatja ezt a megoldást, a Github Actions-ös élesítéssel automatizálható a dockerizált élesítés is. Ehhez a megoldáshoz először dockerizálom az alkalmazást, ami nem jelent mást, mint egy „Dockerfile" nevű fájl létrehozását a projektben, amiben leírom az alkalmazást futtatni képes konténer megépítésének parancsait. A 10. ábra mutatja a Dockerfile tartalmát. Látható, hogy ez nem más, mint a Docker által használt parancsok kombinálása az konténer alapját képező Linux Alpine operációs rendszer parancsaival.

```
1  FROM python:3.7-alpine
2
3  # Heroku needs to set the port
4  ENV PORT 8000
5
6  ADD ./requirements.txt /tmp/requirements.txt
7
8  RUN apk add --no-cache --update bash libpq postgresql-dev  && \
9      apk add --no-cache build-base
10
11 RUN pip install --no-cache-dir -q -r /tmp/requirements.txt
12
13 ADD ./healthcheck_api /opt/healthcheck_api
14 ADD ./migrations opt/migrations
15
16 WORKDIR /opt
17
18 CMD gunicorn --bind 0.0.0.0:$PORT "healthcheck_api:create_app()"
```

10. ábra: Az alkalmazás dockerizálása

Forrás: Képernyő fotó a Dockerfile tartalmáról

A CI/CD pipeline létrehozásához először létrehozok egy új projektet Herokun, most azonban nem állítom be az automatikus élesítést a projekthez. A Github Actions munkafolyamat (workflow) parancsait leíró fájlokat a „.github/workflows" mappa alatt kell tárolni, ezért itt létrehozok egy „build_and_deploy_heroku.yml" nevű fájlt, és leírom a pipeline lépéseit. A kód másolására és a python telepítésére az előre megírt lépés-gyűjteményeket használom. A tesztek futtatását ebben a megoldásban nem a Docker konténerben futtatom, hanem a Github Actions által biztosított virtuális gépen. Ennek oka, hogy a Herokura történő

docker konténeres élesítés esetén a Heroku CLI[9]-nek kiadott parancs akkor is újra megépítené a konténert, ha ezt már korábban tesztek futtatásához megtettem volna. Így el tudom kerülni a konténerépítés duplikálását, amivel időt lehet megspórolni.

Ahhoz, hogy a virtuális gépről a Heroku CLI hozzá tudjon férni a felhasználói fiókom projektjeihez, szükség van egy API kulcsra. Mivel ez a kulcs titkos, ezért nem tárolhatom a nyilvánosan elérhető forráskódban, ehelyett lehetőségem van Githubon titkos változókat hozzáadni a projekthez, amelyek utána használhatóak a Github Actions leíró fájlban.

A lépések megírása után beküldöm Githubra a változtatásokat, és az Actions menüpont alatt látszik, ahogy elindul a pipeline. Sikeres lefutása után látszik Herokun az új élesítés, és az előzőhöz hasonlóan ezen a linken ki tudom próbálni az alkalmazást: https://szakdolgozat-docker-stage.herokuapp.com. Sajnos dockerizált élesítések esetén nem működik a Heroku automatikus szállítás funkciója, így ahhoz, hogy az egygombos élesítést meg tudjam valósítani, ennek a lépéseit is el kell készíteni Github Actions-el. Az éles környezetbe történő szállításhoz egy másik munkafolyamatot definiálok, amit a „release_production_heroku.yml” nevű fájlba írok. Úgy állítom be, hogy ne fusson automatikusan minden változtatáskor, hanem egy gomb jelenlen meg a Github Action webes felületén, aminek megnyomása indítja a folyamatot.

4.5 Github Actions és Kubernetes (C eset)

A Kubernetes platform alacsonyabb absztrakciós szinten orkesztrálja a konténereket, mint a Heroku. Ennek előnye, hogy nagyobb rugalmasságot biztosít a fejlesztőknek és az üzemeltetőknek, mivel a futtató környezet és az infrastruktúra finomhangolása is lehetséges vele. Hátránya, hogy megértése és elsajátítása sokkal több időt vesz igénybe. Míg a Heroku platformon a beállítások nagy része a grafikus felületről néhány kattintással elérhető és módosítható, a Kubernetes esetében a beállításokat konfigurációs fájlokba kell írni. Az alkalmazás élesítése csak dockerizált konténerekkel valósítható meg, így az előző megoldásnál létrehozott Dockerfile-t itt is ki tudom majd használni. A Kubernetes infrastruktúrában „pod”-nak nevezzük a legkisebb élesíthető egységet, ez lényegében az alkalmazás konténerét futtatja. A több, egyforma konténerből épített egységet pedig „deployment”-nek nevezzük. Ahhoz, hogy

[9] parancssoros kezelőfelület

az alkalmazás kívülről – az internetről – is elérhető legyen, egy „service”-t is létre kell hoznunk a Kubernetes infrastruktúrában. A service, a deployment és a benne található podok konfigurációját a „kubernetes/base/deployment.yml” fájlban adom meg. Itt kell továbbá beállítani, hogy az élesítés módja „rolling update” legyen, ez a leállás nélküli élesítési stratégia. Ehhez az szükséges, hogy legalább két podot tartalmazzon a deployment. Amíg az egyik ki nem cserélődött az újra, addig a másik fut tovább, így mindig van legalább egy, ami ki tudja szolgálni a bejövő kéréseket. Szintén ebben a fájlban kell megadnunk a korábban a konténer jegyzékbe feltöltött docker konténer verzióját, amiből az élesítést szeretnénk végezni, illetve a futtatás egyéb részleteit is itt állíthatjuk be. Vannak olyan beállítások, amiket dinamikusan, a pipeline futása közben szeretnék beállítani, mint például a konténerek verziószáma. Ezekhez a Kuberneteshez készített „kustomize” nevű csomagot használom, ami képes arra, hogy a Kubernetes konfigurációs fájlokban található értékeket lecserélje. Ennek beállításához létre kell hoznom a „kubernetes/base/kustomize.yml” nevű fájlt, ami ennek a csomagnak a beállításait tartalmazza.

Ahhoz, hogy a Kubernetes Engine és Container Registry szolgáltatásokat használni tudjam, először a Google Felhő Platformon be kell őket kapcsolnom, majd létre kell hoznom egy „service-accountot”, ami a CI/CD pipeline authentikációját fogja biztosítani. A service-account adatait ebben az esetben is hozzáadom Githubon a projekthez titkos változóként (ahová a HEROKU_API_KEY is került).

A pipeline leírásához, az előző megoldáshoz hasonlóan a „.github/workflows” mappa alatt létrehozok egy új fájlt, „build_and_deploy_kubernetes.yml” néven, amiben leírom a lépéseket. Itt a kód másolása után rögtön a docker konténer megépítése fog következni, mivel a teszteket is ezen szeretném futtatni. A tesztek docker konténerben futtatásához a Docker egy kiegészítő csomagját is be kell állítanom, ehhez a „docker-compose.yml” nevű fájlt hozom létre. A tesztek futtatása után az élesítés a Google Felhő parancssoros kezelőfelületének, a „gcloud”-nak, a megfelelő parancsaival történik.

4.6 Google Cloud Build és Kubernetes (D eset)

A Google Cloud Build, hasonlóan a Github Actions-höz, virtuális gépeken futtatja a CI/CD pipeline-okat, viszont itt a virtuális gép környezetéről semmit sem tudunk, csak egy erőforrás csomagot lehet választani. A pipeline lépéseit docker konténerekben futtatja, ezért minden egyes lépés esetén meg kell adnunk, hogy az melyik docker konténerben fusson: egy nyilvánosan elérhetőben, vagy épp egy helyileg építettben. Az előző esethez készített Kubernetes konfigurációt itt is fel tudom használni, a Kustomize nevű eszköz segítségével, amivel lehetőségem van, csak a pipeline specifikus beállításokat módosítani.

A kód Githubról történő másolása és a pipeline elindítása új változás beküldésekor beállítható a Google Felhő Platform webes felületén. Ugyanitt kell megadni azt is, hogy a verzió követőben melyik fájl tartalmazza a pipeline lépéseit. Ez alapértelmezetten „cloudbuild.yml”, így én is ezt az elnevezést fogom használni. A lépések sorrendje ugyanaz, mint az előző megoldásban, különbség csupán a lépések leírásának szintaktikájában, illetve abban van, hogy az élesítéshez nem szükséges külön authentikációt beállítani, mivel eleve a Google Felhőben fut a pipeline.

11. ábra: Részlet a Github Action és a Cloud Build lépéseit leíró fájlokból

Forrás: Képernyő fotó

5. Eredmények

5.1 Empirikus eredmények

Az esettanulmányok készítése során lehetőségem volt megfigyelni a különböző DevOps eszközök komplexitását. Legegyszerűbbnek a HerokuCI és a Heroku kombinációja (A eset) bizonyult: a konfiguráció nagy részét a webes felületen kódolás nélkül össze lehetett állítani. A hátralévő beállításokhoz négy fájlt is létre kellett hozni, viszont mindegyikben csak néhány sor konfigurációra volt szükség. A teljes pipeline összeállításához valamivel több, mint egy órára volt szükség. A pipeline lépéseinek eredményeit a webes felületen külön-külön menüpontok alatt lehet elérni, ami megnehezíti az áttekintést.

Az összes többi megoldáshoz az alkalmazás konténerizálását saját magamnak kellett elvégeznem, ami egy egyszeri extra időbefektetést jelentett a megoldások elkészítésében, viszont ez a lépés az A eset kivételével mindegyik pipeline-hoz szükséges volt. Mivel az alkalmazás minimális, ez nem vett sok időt igénybe.

A B és a C esetben ugyanazt a CI eszközt használtam (Github Actions). Mindössze egy fájlt kellett létrehozni – pipeline-onként - benne a pipeline összes lépésével. A webes felületen – a titkos változók kivételével – nem kellett semmit beállítani, minden konfiguráció ebbe az egy fájlba került.

A dockerizálást követően a B esetet már, gyorsan, szintén valamivel több mint egy óra alatt el tudtam készíteni. Ez egyrészt annak köszönhető, hogy a teszt futtatási lépést nem az alkalmazás konténerében futtattam, hanem a Github Actions virtuális gépén. Szintén megkönnyítette a megvalósítást, hogy a Heroku parancssori alkalmazása összesen két parancs kiadásával elvégzi a konténer megépítését, tárolását a Heroku konténer jegyzékébe és az élesítést.

A C eset elkészítése ennél sokkal több időt vett igénybe, aminek kisebb részét (kb. 2 óra) az tette ki, hogy ennél a megoldásnál már külön-külön lépésekben kellett gondoskodnom az alkalmazás konténer megépítéséről, tárolásáról és élesítéséről. Viszont emiatt a teszteket is lehetőségem volt a megépített konténerben futtatni – anélkül, hogy redundáns lépéseket iktatnék be - ami egy újabb eszköz (docker-compose) bevezetését tette szükségessé. Ezeknél

azonban sokkal több időbe telt (kb. 5-6 óra) a Kubernetes infrastruktúra megértése és konfigurálása. Mivel a Kubernetes platformon a konténerek futtatása, orkesztrációja és erőforrás eloszlása egészen apró részletekig szabályozható, így egy optimális konfiguráció elkészítése is sokkal több időbe telik. Ezeken kívül pedig még arra is külön időt kellett szánnom (kb. 1 óra), hogy létrehozzak a Google Felhő Platformon egy olyan ún. „service account" -ot, amivel a Github Actions műveleteket tud végezni a platformon.

A D esetben, ahol már a Google Felhő Platform saját CI megoldását használtam, ilyen jogosultságok beállítására nem volt szükség és a Github integráció beállítása is csupán néhány percet vett igénybe. A C esteben létrehozott Kubernetes konfigurációt újra fel tudtam használni, fontos megjegyezni azonban, hogy az arra szánt idő itt is szükséges lett volna, amennyiben csak egy pipeline-t készítek. A Cloud Build lépéseit szintén egyetlen fájlba kell leírni, viszont itt nincs lehetőség a virtuális gépen közvetlenül parancsokat futtatni, hanem minden lépést egy Docker konténerben kell futtatni. Emiatt a lépéseket leíró fájl szintaktikája bonyolultabb, megértése nehezebb és a működő megoldás létrehozása több időbe telt (kb. 2-3 óra), mint C esetben.

Az A eset nemcsak a lépéseket leíró fájl hiányában tér el a többitől, hanem az eredmények megmutatásában is. Míg a B, C és D esetekben létezik egy, a futtatásokat, azok lépéseit és a lépések kimenetelét (logját) is megmutató oldal a webes felületen, az A esetben egy külön oldalon nézhetjük meg a tesztek futtatását és azok kimenetelét és egy másikban a konténerizálás és élesítés kimenetét. Ez rendkívül kényelmetlen és nehézséget is okoz az adott verziójú kód teszt futtatási és élesítési lépések kimenetének összekapcsolásában. Sőt a később még elemzésre kerülő teljes pipeline futási idő megállapításában is.

A CI/CD pipeline-hoz köthető DevOps gyakorlatok közül a verziókövetést, a folyamatos integrációt és a tesztautomatizálást mind a négy esettanulmány megvalósítja. Az élesítés-szállításhoz kapcsolódó gyakorlatok közül viszont néhány – bizonyos esetekben – csak részben vagy egyáltalán nem teljesül. Az A esetben a konténerizáció a Heroku saját rendszerében történik, ezért elveszíti a reprodukálhatóságát, illetve a platform függetlenségét. Az itt épülő konténereket, nem tudjuk felhasználni a fejlesztéshez vagy egy másik helyen történő élesítéshez. Szintén emiatt a futtató környezet beállítására is csak korlátozott eszközeink vannak (az „app.json" nevű konfigurációs fájl használatával, néhány további beállítás azért lehetséges). A kód alapú infrastruktúra szinte egyáltalán nem valósul meg ebben az esetben,

egyedül az erőforrások használatát van lehetőségünk meghatározni, de csak a kész erőforrás csomagokból választhatunk és ezt sem muszáj kód alapon konfigurálni, megtehetjük a webes felületen is. A folyamatos élesítés minden kritériumának eleget tesz: élesítés közben nincs leállási idő, hiba esetén automatikus a helyreállítás és az egygombos élesítéshez is kész megoldást jelent a „promote to production” gomb.

A B esetben ehhez képest már Docker alapon történik a konténerizálás, így a futtató környezet beállítása lehetséges. Mivel ez a megoldás is a Heroku platformra élesít, a kód alapú infrastruktúrára és a folyamatos élesítésre ugyanaz érvényes, mint az A esetben. Azzal a kivétellel, hogy a „promote to production” gomb docker alapú Heroku-s alkalmazások esetében nem működik, így az egygombos élesítést egy másik, a Github Actions felületéről indítható automatizáció segítségével tudtam csak megvalósítani.

C és D esetben a konténerizáció szintén Docker segítségével valósul meg, aminek az előnyeit még jobban kihasználja, hogy a tesztek is az alkalmazás konténerben futnak, így a futtató környezet és az alkalmazás kód inkompatibilitásából adódó hibák már a tesztek futtatása során előjöhetnek, ami segít megvédeni az alkalmazást attól, hogy ilyen problémákra, már csak az élesítés után derüljön fény. Érdemes megjegyezni, hogy ez részben az A esetben is igaz, hiszen, a HerokuCI ugyanazt a konténerizációs folyamatot végzi el a tesztek futtatásához, mint a Heroku élesítési platform, viszont ez esetben a konténer megépítése kétszer történik, nem pedig újra használódik. Mivel ezekben az esetekben az élesítés Kubernetes platformra történik, a kód alapú infrastruktúra teljes mértékben megvalósul, az alkalmazás részeként tárolt konfigurációs fájl(ok)nak köszönhetően. Az egy gombos élesítésre a Kubernetes nem kínál megoldást, így ezekben az esetekben is egy a kezelő felületről indítható automatizáció létrehozásával lehet megvalósítani. Ezt a D esetben a Cloud Build ugyanúgy lehetővé teszi, mint a Github Actions, de a Google Felhő Platform külön, kifejezetten ezt a funkciót megvalósító szolgáltatást is nyújt, ez a Cloud Deploy.

Az automatikus skálázódás, vagyis a kéréseket kiszolgáló konténerek számának automatikus növelése nagyobb terhelés esetén, a Kubernetes platformon automatikusan is működik, de több beállítással is befolyásolhatjuk a működését (akár ki is kapcsolható). Ehhez képest a Heroku platformon, csak az egyik legdrágább csomag választása esetén lehetséges és működése sem állítható.

A HerokuCI csak Herokura történő élesítéshez használható, és oda is csak a saját konténerizációs megoldásával. Abban az esetben, ha Docker alapú konténerizációt akarunk használni, már át kell térnünk egy másik CI eszközre. A Github Actions ezzel szemben képes megvalósítani a Herokura való élesítést a Docker alapú konténerizációval is, valamint egyéb platformokhoz is használható. Sőt a legtöbb alkalmazás platformhoz kínál a közösség előre megírt lépéscsomagokat is. A virtuális gépen amit biztosít, futtathatunk közvetlen parancsokat, vagy az előre telepített Docker segítségével konténereken is végezhetjük a lépéseket. Ez rendkívül nagy szabadságot ad a fejlesztők kezébe, ami biztosítja, hogy a CI/CD pipeline-ja is könnyen tudjon alkalmazkodni az alkalmazás növekedéséhez, változásaihoz.

A Cloud Buildben ez a szabadság már korlátozottabb. Amennyiben a lépések között meg szeretnénk osztani információt (pl.: keletkezett fájlok, telepített programok), ezt külön be kell állítanunk, hiszen minden lépés külön konténerben fut. Előre megírt lépés csomagok itt nem állnak rendelkezésre, viszont a legtöbb CI/CD pipeline-okban gyakran használt eszköz konténerizált verziója elérhető hozzá, ezzel megkönnyítve a lépések kivitelezését. Habár elméletileg lehetséges vele más platformokra (akár Herokura) élesíteni, a felépítésén mégis érződik, hogy elsősorban a Google Felhő Platform szolgáltatásaihoz van kitalálva.

Ezekből levonható tehát az a következtetés, hogy habár a CI/CD pipeline definícióját kielégítő megoldást létre lehet hozni Heroku platformra történő élesítéshez is, az összes DevOps gyakorlatot így nem tudjuk megvalósítani. A Kubernetesre történő élesítés ezzel szemben jobban illeszkedik a DevOps szemlélethez, viszont több időt és szakértelmet kíván a fejlesztőktől. Emiatt nagyon fontos mérlegelni, hogy azokra a gyakorlatokra, amikre a Heroku nem képes, valóban szüksége van-e az adott projektnek, az alkalmazásnak és a stakeholdereknek.

5.2 Kvalitatív eredmények

A négy esettanulmány elkészítése után lehetőségem van arra, hogy ezeket egyszerre futtassam, akár egy üres „változtatás" beküldésével a verziókövetőbe. Így pontosan ugyanaz a kód megy végig az összes CI/CD pipeline-on, ami által a különböző megoldások teljesítménye összemérhetővé válik. A teljesítmény legfontosabb mutatója a pipeline-ok futási ideje, mivel ez határozza meg, hogy a fejlesztők milyen gyorsan értesülnek a változtatások sikeréről, illetve

abban is lehet szerepe, hogy egy hibajavítás vagy egy új funkció mennyire gyorsan lesz elérhető az ügyfelek számára. A szoftverek másik fontos mutatója, hogy a növekedéssel szemben mennyire robusztusak, ezt skálázhatóságnak nevezzük. Emiatt azt is érdemes megvizsgálni, hogy amennyiben bővül az alkalmazás, hogyan változik a CI/CD pipeline-ok teljesítménye, sebessége. Ahhoz, hogy a növekedést modellezni tudjam, kibővítem az alkalmazást egy adatbázissal, mivel a legtöbb webes alkalmazás használ valamilyen adatbázist, ami a tesztautomatizálás és az élesítés során is kihat a pipeline teljesítményére. Ezen kívül a „pytest-benchmark" csomagot fogom használni arra, hogy az automatizált tesztek során bizonyos műveleteket többezerszer lefuttassak, és így modellezni tudjam egy sok tesztből álló projekt teljesítményigényét. A megoldás kiválasztásakor a teljesítményen és skálázhatóságon kívül a harmadik fontos tényező a költségvonzat. Egy CI/CD pipeline esetében ez a különböző szolgáltatások havidíjának és/vagy teljesítmény alapú árazásának az összegét jelenti. Így az alkalmazás növekedésének ebben is fontos szerepe van. Elképzelhető, hogy egy kisebb projektnél más megoldások rendelkeznek a kedvezőbb árral, mint egy várhatóan nagyra növő projekt esetében. Az árak a szolgáltatások honlapjáról elérhetőek, ahol sok esetben még ár kalkulátor is a rendelkezésre áll a számolás megkönnyítése érdekében. Ahhoz, hogy valós képet kapjak egy-egy megoldás várható áráról, a számoláshoz a DORA[10] és a Google Cloud közös felméréséből származó adatokat használom fel, melyben megtalálható, hogy a DevOps-ot gyakorló vállalatok milyen számban futtatnak CI/CD pipeline-okat. (services.google.com)

5.2.1 Teljesítmény mérés

B, C és D esetben a pipeline teljes futási ideje könnyen megállapítható, az eredményeket mutató weboldalon a lépésekre szánt idő külön-külön is megjelenik és a pipeline teljes futási ideje is látható. Az A esetben ez csak a tesztfuttatási lépésnél látszik és az figyelhető meg, hogy ezután csak később – akár néhány perccel - indul csak el az élesítési folyamat, melynek idejét nem mutatja a Heroku felülete, viszont viszonylag stabilan 30-40 másodpercet vesz igénybe. Ez összesen sokkal hosszabb, mint a többi pipeline futási ideje, még a minimális projektnél is. Mivel ennek a pipeline-nak a mérése nem pontos, így az összehasonlításból kihagyom, a különböző állapotokban csak egyszer mérem meg a futási idejét kézzel és ezt tájékoztató jelleggel feltüntetem. A többi esetben viszont három kísérletet végzek, ahol mindegyikben

[10] DevOps Research and Assessment

tízszer fogom futtatni a CI/CD pipeline-okat, a futási idők várható értékeit fogom egymással összehasonlítani varianciaanalízissel.

A kísérletek elvégzése után az IBM SPSS Statistics szoftver segítségével elemeztem az eredményeket. A futási időt másodperc pontosan méri a B, C és D eset. Mindhárom kísérlethez három független, tíz elemű minta áll rendelkezésre. A varianciaanalízis előfeltételeit ellenőriztem, a minták normális eloszlását a Kolmogorov-Smirnov teszttel, a szórás azonosságot Levene teszttel. A varianciaanalízis után pedig a Bonferroni post-hoc eljárást is alkalmaztam az eredmények részletes értékelése céljából. Az SPSS-es eredményei megtekinthetők a mellékletben.

Az első kísérletben a minimális alkalmazáshoz készült CI/CD pipeline-ok futottak. A varianciaanalízis szerint van szignifikáns eltérés a várható futási idők között. A post-hoc Bonferroni teszt pedig azt is megmutatja, hogy a B és C megoldás között van a szignifikáns különbség. Mivel egy minimális alkalmazás tesztjeinek futtatása és konténereinek építése rendkívül gyors és mindkét pipeline Github Actions-ön fut, megállapítható, hogy az eltérést az élesítési platform okozza. A Kubernetes leállás nélküli élesítési stratégiája lassabb, mint a Heroku automatikusan működő megoldása. A B és D esett között viszont valószínűleg azért nincs szignifikáns eltérés, mert a Google Cloud Build képes ezt kompenzálni az alkalmazás konténerek gyorsabb építésével. Ezt megerősíti a második kísérlet eredménye is.

A második kísérlethez kiegészítettem az alkalmazást. Adatbázis kapcsolatot építettem, illetve létrehoztam benne egy táblát, amihez az alkalmazás író, olvasó, frissítő és törlő végpontokat biztosít. A végpontokat lefedtem tesztekkel. A CI/CD pipeline-ok futtatásába beépítettem a teszt adatbázisok létrehozását és az adatbázis migrációk [11]futtatását a tesztadatbázison és az élesítés során az éles adatbázison is. Ezek a műveletek minden esetben lassították a pipeline-ok futási idejét. Az elemzés során megállapítható, hogy a D eset futási ideje szignifikánsan gyorsabb a B és C esetnél. Mivel ebben a kísérletben a tesztadatbázisok létrehozásához, újabb konténerek építésére van szükség, az eredmény azt bizonyítja, hogy a Google Cloud Build optimálisabban kezeli a konténereket, mint a Github Actions.

[11] az adatbázis séma változtatások érvényesítése

A harmadik kísérlet során a pytest-benchmark csomag segítségével újabb teszteseteket adtam hozzá az alkalmazáshoz. Ezeknek a teszteknek a célja nem az alkalmazás működésének az ellenőrzése, hanem az, hogy a végpontok sokszori meghívásával (5000/végpont) erőforrás igényt generáljanak a CI/CD pipeline-okban. Így megvizsgálható, milyen teljesítményt nyújtanának a pipeline-ok egy sok teszttel rendelkező projekt esetében. Az eredmény szerint a C eset szignifikánsan gyorsabb, mint a D. Ez bizonyítja, hogy a Github Actions által biztosított virtuális gép (amennyiben az Ubuntu operációs rendszert használjuk), jobb teljesítménnyel bír mint a Google Cloud Build esetében. B eset valószínűleg azért nem gyorsabb szignifikánsan a D estnél, mert az adatbázis migráció a Herokura történő élesítéskor, egy redundáns lépés beiktatását tette szükségessé (dupla konténer építés), ami B esetet lelassította annyira, hogy a virtuális gép teljesítményének előnye elvesszen.

5.2.2 Árkalkuláció

A legtöbb felhő szolgáltató havi számlázási ciklussal dolgozik, ezért általában az árakat is egyhavi alapon határozzák meg. Előfordul fix havidíjas szolgáltatás, de használat alapú is. Utóbbi esetben a havidíjnak, a használat perc alapú hányada kerül számlázásra. Olyan is előfordul, hogy a kettőt egyszerre alkalmazzák. Az árkalkulációt, én is egyhavi használatra vetítve számoltam, futási időnek pedig az utolsó (leghosszabban futó) kísérlet eredményeinek a futási idő várható értékét használtam. A pipeline futtatások számának, a DORA 2019-es felmérésében szereplő adatokat használtam. (services.google.com) Ebben az szerepel, hogy a jól teljesítő DevOps-ot gyakorló cégek, naponta többször is élesítenek, átlagosan évente 1460-szor, ami havi ~122 élesítést jelent. A kalkuláció nem tartalmazza az üzemeltetés költségeit, kizárólag a CI/CD pipeline-ok futtatásából eredőt költségeket tartalmazza. Emellett az is fontos megjegyezni, hogy a legtöbb vállaltnak általában nem csak egy alkalmazása van. A mikroszervíz architektúrával dolgozó vállalatoknál, a költségek szolgáltatásonként értendők, mivel minden egyes szolgáltatásnak külön CI/CD pipeline-t kell fenntartani.

A HerokuCI havi tíz dolláros alapdíjjal rendelkezik, ezen felül a tesztek futtatása során használt erőforrás csomag perc arányos havidíja kerül felszámításra. Az esettanulmányban én a legolcsóbb lehetőséget használtam (standard-1x) aminek havidíja huszonöt dollár. A Heroku saját konténerizálási folyamatáért külön díjat nem számol fel, minden egyéb költség az

üzemeltetéshez tartozik. A teszt futtatása lépést másodpercre pontosan méri a HerokuCI felülete, így a harmadik kísérletből származó, az A esetre vonatkozó adatokat fel tudom használni az ár becsléshez. A kalkuláció eredménye, 10,42 dollár, aminek jelentős része a fix havidíj, a futtatásból származó költségek minimálisak.

A Github Actions ingyenes csomagjával havi 2000 perc futtatási idő áll rendelkezésünkre és csak ezen felül számol fel percdíjat. Mivel B és C esetben is kevesebb mint 600 perc lett a futási idők várható értéke, így ezekben az esetekben ingyenes a szolgáltatás.

A Google Cloud Build szolgáltatása esetében napi 120 perc, ingyenes ezen felül percenként 0.003 dollárt kell fizetni. Ha feltételezzük, a DORA által meghatározott napi négy futtatás normális eloszlását, akkor bőven a napi limit alatt maradunk, így ezt a szolgáltatást is ingyenesen tudjuk használni. Az eredményekből látható, egyetlen CI/CD pipeline fenntartása minimális költségekkel járhat. Azonban több projekt esetében, az azonos CI eszközt használó pipeline-ok futási ideje, közösen csökkenti a havi, illetve napi limiteket. Egy pontosabb összehasonlítást kapunk, ha kiszámoljuk, hány ugyanilyen CI/CD pipeline fenntartása lépné túl a megengedett limiteket. Ez a Github Actions esetében, ~3,5, a Cloud Build esetében kb.: ~5,5. Ezen kívül a Google Cloud Build, a limitek túllépése után is alacsonyabb percdíjat számol fel, mint a Github Actions. A HerokuCI esetében minden további pipeline 10,4 dolláros számlát generál. Így azt mondhatjuk, hogy több szolgáltatás üzemeltetése esetén a Google Cloud Build az olcsóbb megoldás.

6. Összefoglalás

Szakdolgozatom elején visszatekintettünk a szoftverfejlesztés múltjára. Bemutattam az első szoftverfejlesztési módszertanok kialakulását, melyeket még más iparágakból vett át az informatika. Ezek később saját fejlődésnek indultak, hogy jobban tudjanak alkalmazkodni a szoftverprojektekre jellemző gyors változásokhoz és a gyorsan fejlődő technológiákhoz. Az így kialakult iteratív módszertanok jobb adaptív képességgel rendelkeztek. Míg az első ilyen módszertan, a Scrum még nem tartalmazott technológiai előírásokat, inkább csak a csapatok működését, a ceremóniákat és a fejlesztéshez tartozó dokumentumokat határozta meg, a később kialakuló agilis módszertanok, mint például az eXtrém programozás, már technológiai előírásokat is tartalmaznak. A technológiai gyakorlatok kiterjesztésével az üzemeltetésre, illetve az automatizáció, mint elvárás megjelenésével a 2000-es évek elejére kialakult a DevOps szemlélet, mely átfogó megoldást kínált hosszú távú szoftverprojektek kivitelezésére és üzemeltetésére is. Megismerkedtünk a CI/CD pipeline definíciójával, is, ami a DevOps automatizációs gyakorlatokat egy komplex, lineáris rendszerré alakítja.

Ez a rendszer képes minden lépést elvégezni a kódváltoztatások verziókövetőbe való bekerülésétől kezdve egészen a változtatás élesítéséig. Egy ilyen rendszer kialakítása több lépésből áll, és több eszköz összehangolt, integrált működését igényli. Amennyiben ilyen rendszereket szeretnénk létrehozni, fontos az egyes lépések, és a hozzájuk tartozó technológia elméleti hátterének megértése. A verziókövetés a szoftver kód és annak változatainak menedzselését teszi lehetővé, valamint egyben a CI/CD pipeline indító eleme is. A tesztautomatizáció segít megbizonyosodni arról, hogy a változtatások, amiket a pipeline élesíteni fog, nem okoznak-e hibát a rendszerben. Ez egy fontos minőségbiztosítási eszköz és a napi többszöri élesítés esetén elengedhetetlen. A konténerizáció képes az alkalmazást platform független csomaggá alakítani, annak futtató környezetével együtt. Ezáltal a különböző verziókból készült konténereket el lehet tárolni, és több különböző környezetben újra felhasználva lehet élesíteni. Segítheti a tesztelést is, amennyiben a teszteket is az alkalmazás konténerekben futtatjuk. A konténerizáció elvégzése manuális feladat, a CI/CD pipeline futtatása során pedig a konténerek megépítése, tárolása és élesítéshez való előkészítése történik. Több konténerből álló komplex infrastruktúra kialakítását konténer orkesztrációs platformok teszik lehetővé, amelyek képesek a konténerek teljes életciklusáról gondoskodni, ezáltal

megvalósítani a leállási idő nélküli, folyamatos élesítést. A CI/CD pipeline lépéseit és folyamatábráját a 7.ábra mutatja.

Az elméleti összefoglaló után négy esettanulmányt terveztem, melyben ugyanahhoz a minimális web alkalmazáshoz különböző eszközökkel valósítottam meg CI/CD pipeline-okat. Verziókövetőnek minden esetben a Github-ot használtam, élesítési platformnak a Heroku-t és a Kubernetes-t választottam, folyamatos integrációs eszköznek pedig a HerokuCI-t, a Github Actions és a Cloud Build-et. A pipeline-ok összeállításáról a 8. ábra ad átfogó képet. Az esettanulmányok kivitelezése során a könnyebb, gyorsabban kivitelezhetőtől a nehezebben kivitelezhetőig haladtam. Egyes esetekben sok beállítást a felhőszolgáltatások webes felületén is meg lehetett csinálni, viszont mindegyik pipeline kialakításához kellett konfigurációs fájlokat is elhelyezni az alkalmazás könyvtárába.

Az eredmények empirikus értékelése során az alábbi megállapításokra jutottam. Az A esetben a legtöbb beállítást a webes felületen meg lehetett tenni, a konfigurációs fájlokba csupán néhány sor került, és az alkalmazás konténerizálását sem kellett elvégezni. A többi eset előtt már el kellett végezni az alkalmazás konténerizálását, de láthattuk, hogy ez egy egyszeri tevékenység, tehát elkészítése után a B, C, D esetben is fel lehetett használni. Ezek az esetek abban is eltértek az elsőtől, hogy mindegyik tartalmazott egy, a pipeline lépéseit leíró fájlokat, illetve a pipline futások lépéseit és eredményeit egy oldalon meg lehetett tekinteni, így ezek sokkal átláthatóbbak voltak. B esetben a Herokura való Dockeres élesítést megkönnyítette a Heroku parancssoros kezelőfelülete, ami a konténerek építését, tárolását és élesítését is támogatja. Így azonban a teszteket a Github Actions virtuális gépén közvetlenül voltam kénytelen futtatni. A C és D esetben már tudtam a teszteket az alkalmazás konténerében futtatni, viszont ezeknek az eseteknek a komplexitását mégsem ez, hanem a Kubernetes megértése és konfigurálása okozta. Mivel Kubernetesen sokkal több beállítás, konfiguráció lehetséges, mint Herokun, így ezzel a fejlesztési idő is növekszik. D esetet tovább bonyolította, hogy a Cloud Buildben már nem tudtam közvetlenül a virtuális gépen futtatni parancsokat, csak Docker konténerekben, illetve a lépéseket leíró fájl szintaktikája is bonyolultabb volt.

Láthattuk, hogy azokban az esetekben, ahol az élesítés Herokura történik (A és B eset), az élesítéshez-szállításhoz tartozó DevOps gyakorlatok közül nem mindegyik teljesül. A kód alapú infrastruktúra nem megvalósítható, mivel a Heroku csak kész csomagokat kínál az infrastruktúra beállítására. B esetben a dockerizációnak köszönhetően a futtató környezet

beállítása már lehetséges (míg A esetben ez sem teljesült). Az egygombos élesítésre viszont csak az A eset kínál kész megoldást. A többi esetben újabb automatizációt szükséges hozzá létrehozni. A Kubernetest használó esetekben (C és D) a kód alapú infrastruktúra teljesül, ugyanakkor több mindent kell konfigurálni, mint például a leállás nélküli élesítést. (A Heroku esetében ez is automatikusan működik.) Ezekből levonható az a következtetés, hogy a Heroku kevésbé illeszkedik a DevOps szemlélethez, mint a Kubernetes vagy más konténer orkesztrációs platformok.

A HerokuCI csak Herokura történő élesítést támogat, ezzel szemben a Github Actions és a Cloud Build kompatibilis egyéb platformokkal is. Mivel azonban a Cloud Build a Google Felhő Platform része, elsősorban Kubernetesre és más Google Felhő alapú folyamatok automatizálására használható a legkönnyebben.

Az eredmények empirikus értékelése után 3 kísérletet végeztem, melyben az esettanulmányok teljesítményét vizsgáltam. Az A esetben pontos méréseket sajnos nem lehetett végezni, de az látható volt, hogy a teljesítménye jóval elmarad a többihez képest. Az eredményeket varianciaanalízissel vizsgáltam. Az első kísérletből kiderül, hogy a Kubernetes leállás nélküli élesítési stratégiája lassabb a Herokuénál. A második kísérlet megmutatta, hogy a Cloud Build konténerkezelési teljesítménye jobb, mint a Github Actions-é. A harmadik kísérlet pedig rávilágított, hogy amennyiben az alkalmazás és a tesztek erőforrás igényesek, a Github Actions által biztosított virtuális gépek jobb teljesítményt nyújtanak.

Az A eset volt az egyetlen, amely még egy alkalmazás esetében is nagyjából tíz dolláros költséggel jár. A többi esetben akkor is ingyenes lenne a CI/CD pipeline fenntartása, ha naponta többször is futnának, viszont több alkalmazás esetén a legolcsóbb megoldás a Cloud Build használatával érhető el.

Jelenleg a DevOps nagy népszerűségnek örvend, a legtöbb piacvezető szoftverfejlesztő cég alkalmazza a gyakorlatait. Láthattuk azonban, hogy készíthető olyan CI/CD pipeline is, amely nem valósítja meg az összes gyakorlatot. Még ha sikerül is kialakítanunk ilyen pipeline-t, az még nem jelenti azt, hogy sikeresen adaptáltuk a DevOps szemléletet. Ehhez szükségesek egyéb automatizációk is, mint például a monitorozás, amelynek bemutatására a szakdolgozatban nem tértem ki. Szintén elengedhetetlen a kollaboratív kultúra kialakítása a cégen belül, ez viszont főleg a menedzsment feladata, így fejlesztési szempontból erre se tudtam kitérni.

Az itt bemutatott esettanulmányok elsősorban webalkalmazások élesítésére szolgálnak, azonban a szoftverfejlesztés egyéb területein is hasznos lehet az élesítési folyamatok automatizálása, például a mobilapplikáció fejlesztésben. Ez a terület újabb kihívások elé állítja az informatikusokat, mivel még kevésbé kiforrottak az eljárások, ezért ez egy érdekes további kutatási terület lehet a jövőben. Létezik már olyan cég, ami kifejezetten ezt a területet célozza, ez a Bitrise. Szintén érdekes kutatási lehetőséget teremt a mostanában megjelenő DevSecOps kifejezés, ami a fejlesztési és üzemeltetési szempontok mellett a kiberbiztonsági szempontokat is figyelembe veszi.

Az eredményeket tekintve, saját hobbi projektjeimhez a C esetben használt technológiákat használnám, illetve fogom használni a jövőben, főleg annak rugalmassága és egyszerűsége miatt. Komplexebb, nagyobb projektek esetében viszont inkább a D esetben használt eszközöket választanám, elsősorban az alacsony ár miatt. Illetve amennyiben például egy cég vagy projekt infrastruktúrája részben, vagy teljes egészében a Google Felhőn alapszik, a Cloud Build jól fog illeszkedik a projekthez, a fejlesztők így mindent ugyanazon a platformon tudnak elvégezni.

Irodalomjegyzék

Folyóirat cikkek:

C. Ladas 2009, „Scrumban-essays on kanban systems for lean software development.” Modus Cooperandi Press, Seattle, USA

J. Díaz, D. López-Fernáandez, J. Pérez, A. Gonzáleza-Prieto (2021) „Why are many businesses instilling a DevOps culture into their organization?” Empirical Software Engineering, 26: 25

J. Guerrero, K. Zuniga, C. Certuche and C. Pardo (2020) “A systematic mapping study about Devops”, Jou. Cie. Ing., vol. 12, no. 1, pp. 48-62

L.E. Lwakatare, T. Kilamo, T. Karvonen et al. (2019) „DevOps in practice: A multiple case study of five companies”, Information and Software Technology, vol. 114, pp.217-230

M. Liviu (2014) “Comparative study on software development methodologies” Database Systems Journal, vol. V, no. 3, pp. 37-56

M. Shahin, M. Zahedi, M. A. Babar, L. Zhu (2018) „An empirical study of architecting for continuous delivery and deployment” Empirical Software Engineering, vol. 24, pp. 1061–1108

M. Waseem, P. Liang, M. Shahin (2020) “A Systematic Mapping Study on Microservices Architecture in DevOps”, The Journal of Systems & Software, vol. 170, 110798

R. Jabbari, N. Ali, K. Petersen (2016) „What is DevOps? A Systematic Mapping Study on Definitions and Practices”, Conference: XP ’16 Workshops, Edinburgh, Scotland Uk

T. Scheepers (2014) “Virtualization and Containerization of Application Infrastructure: A Comparison” 21st Twente Student Conference on IT, University of Twente, Enschede, Netherlands

W.P. Luz, G. Pinto, R. Bonifácio (2019) „Adopting DevOps in the real world: A theory, a model, and a case study", The Journal of Systems and Software, vol. 157, 110384

Könyvek:

A. Ravichandran, K. Taylor, P. Waterhouse (2016) DevOps for Digital Leaders: Reignite Business with a Modern DevOps-Enabled Software Factory. New York, Springer Science and Business Media LLC

J. Shore, S. Warden (2007) The Art of Agile Development. USA, O1Reilly Media Inc.

Szakdolgozatok:

G. Kocsis (2020) „Szabadságnyilvántartó rendszer tervezése Agilis módszerrel és UX design felhasználásával" Budapesti Gazdasági Egyetem, Pénzügyi és számviteli kar

V. Csiki-Mara (2019) „Termék-, projekt-, és technológiai kockázatok minimalizálása webes ügyviteli rendszerek agilis módszertan szerinti fejlesztése során" Budapesti Gazdasági Egyetem, Pénzügyi és számviteli kar

Online források:

K. Beck et al. (2001) „Manifesto for Agile Software Development" (URL): *http://agilemanifesto.org* Letöltés időpontja: 2021. 10. 28.

K. Schwaber, J. Sutherland: „A Scrum útmutató Meghatározó útmutató a Scrumhoz: A játék szabályai", (2020) (URL): https://www.scrum.org/resources/scrum-guide
Letöltés időpontja: 2021. 10. 28.

https://www.atlassian.com/git/tutorials/what-is-version-control (Hozzáférés időpontja: 2021. 10. 29)

https://codecool.com/wp-content/uploads/2021/01/devops-1024x475.png(Hozzáférés időpontja: 2021. 11. 28)

https://insights.stackoverflow.com/survey/2021#most-popular-technologies-tools-tech-prof (Hozzáférés időpontja: 2021. 11. 28)

https://services.google.com/fh/files/misc/state-of-devops-2019.pdf (Hozzáférés időpontja: 2021. 11. 28)

Dokumentációk:

https://devcenter.heroku.com/articles/heroku-ci *(Hozzáférés időpontja: 2021. 11. 28)*
https://devcenter.heroku.com/categories/deployment *(Hozzáférés időpontja: 2021. 11. 28)*
https://docs.github.com/en/actions *(Hozzáférés időpontja: 2021. 11. 28)*
https://cloud.google.com/kubernetes-engine/docs *(Hozzáférés időpontja: 2021. 11. 28)*
https://cloud.google.com/build/docs *(Hozzáférés időpontja: 2021. 11. 28)*

Mellékletek

Az esettanulmányok forráskódja

https://github.com/adamFoldvari/szakdolgozat_2021

Spss számítások

1. Kísérlet:

One-Sample Kolmogorov-Smirnov Test

		B	C	D
N		10	10	10
Normal Parameters[a,b]	Mean	62.30	93.10	78.10
	Std. Deviation	6.237	24.292	10.440
Most Extreme Differences	Absolute	.219	.233	.191
	Positive	.219	.118	.191
	Negative	-.192	-.233	-.144
Test Statistic		.219	.233	.191
Asymp. Sig. (2-tailed)[c]		.190	.132	.200[d]

Tests of Homogeneity of Variances

		Levene Statistic	df1	df2	Sig.
mp	Based on Mean	3.041	2	27	.064
	Based on Median	2.067	2	27	.146
	Based on Median and with adjusted df	2.067	2	12.268	.168
	Based on trimmed mean	2.724	2	27	.084

ANOVA

mp

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	4744.267	2	2372.133	9.643	<.001
Within Groups	6641.900	27	245.996		
Total	11386.167	29			

Multiple Comparisons

Dependent Variable: mp
Bonferroni

(I) tipus_kod	(J) tipus_kod	Mean Difference (I-J)	Std. Error	Sig.	95% Confidence Interval Lower Bound	95% Confidence Interval Upper Bound
B	C	-30.800*	7.014	<.001	-48.70	-12.90
	D	-15.800	7.014	.098	-33.70	2.10
C	B	30.800*	7.014	<.001	12.90	48.70
	D	15.000	7.014	.125	-2.90	32.90
D	B	15.800	7.014	.098	-2.10	33.70
	C	-15.000	7.014	.125	-32.90	2.90

*. The mean difference is significant at the 0.05 level.

2. Kísérlet:

One-Sample Kolmogorov-Smirnov Test

		B	C	D
N		10	10	10
Normal Parameters[a,b]	Mean	214.70	210.70	174.40
	Std. Deviation	20.581	30.159	22.941
Most Extreme Differences	Absolute	.236	.208	.238
	Positive	.236	.200	.238
	Negative	-.138	-.208	-.187
Test Statistic		.236	.208	.238
Asymp. Sig. (2-tailed)[c]		.120	.200[d]	.115

Tests of Homogeneity of Variances

		Levene Statistic	df1	df2	Sig.
mp	Based on Mean	1.502	2	27	.241
	Based on Median	.800	2	27	.460
	Based on Median and with adjusted df	.800	2	25.013	.460
	Based on trimmed mean	1.463	2	27	.249

ANOVA

mp

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	9859.267	2	4929.633	7.954	.002
Within Groups	16734.600	27	619.800		
Total	26593.867	29			

Multiple Comparisons

Dependent Variable: mp

Bonferroni

(I) tipus_kod	(J) tipus_kod	Mean Difference (I-J)	Std. Error	Sig.	95% Confidence Interval	
					Lower Bound	Upper Bound
B	C	4.000	11.134	1.000	-24.42	32.42
	D	40.300*	11.134	.004	11.88	68.72
C	B	-4.000	11.134	1.000	-32.42	24.42
	D	36.300*	11.134	.009	7.88	64.72
D	B	-40.300*	11.134	.004	-68.72	-11.88
	C	-36.300*	11.134	.009	-64.72	-7.88

*. The mean difference is significant at the 0.05 level.

3. Kísérlet:

One-Sample Kolmogorov-Smirnov Test

		B	C	D
N		10	10	10
Normal Parameters[a,b]	Mean	286.00	268.50	327.90
	Std. Deviation	29.859	20.951	52.412
Most Extreme Differences	Absolute	.169	.190	.237
	Positive	.169	.105	.237
	Negative	-.102	-.190	-.185
Test Statistic		.169	.190	.237
Asymp. Sig. (2-tailed)[c]		.200[d]	.200[d]	.119

Tests of Homogeneity of Variances

		Levene Statistic	df1	df2	Sig.
mp	Based on Mean	1.467	2	27	.248
	Based on Median	1.436	2	27	.255
	Based on Median and with adjusted df	1.436	2	16.293	.266
	Based on trimmed mean	1.376	2	27	.270

ANOVA

mp

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	18634.067	2	9317.033	6.855	.004
Within Groups	36697.400	27	1359.163		
Total	55331.467	29			

Multiple Comparisons

Dependent Variable: mp
Bonferroni

(I) tipus_kod	(J) tipus_kod	Mean Difference (I-J)	Std. Error	Sig.	95% Confidence Interval	
					Lower Bound	Upper Bound
B	C	17.500	16.487	.894	-24.58	59.58
	D	-41.900	16.487	.051	-83.98	.18
C	B	-17.500	16.487	.894	-59.58	24.58
	D	-59.400*	16.487	.004	-101.48	-17.32
D	B	41.900	16.487	.051	-.18	83.98
	C	59.400*	16.487	.004	17.32	101.48

*. The mean difference is significant at the 0.05 level.