GÁBOR DÉNES EGYETEM

MÉRNÖKINFORMATIKUS

SZAKDOLGOZAT

CI/CD PIPELINE MEGVALÓSÍTÁSA FELHŐALAPÚ KÖRNYEZETBEN: EGY GYAKORLATI MEGKÖZELÍTÉS GITHUB ACTIONS ÉS AWS HASZNÁLATÁVAL

Széles Ervin
XWB4XL
300/2025

Kivonat

A dolgozat célja egy Python/Django alapú webalkalmazás folyamatos integrációs és telepítési (CI/CD) folyamatának megtervezése és automatizálása felhőalapú környezetben. A megoldás a GitHub Actions és az Amazon Web Services Elastic Beanstalk szolgáltatásait ötvözi, biztonságos OpenID Connect hitelesítéssel, elkerülve a statikus kulcsok használatát. A módszertan gyakorlati példán keresztül mutatja be a modern felhőalapú fejlesztés fő lépéseit - a kódellenőrzéstől a teljesen automatizált bevezetésig. A pipeline teszteléséhez demonstrációs szintű alkalmazás került kialakításra, hogy a figyelem az automatizálási folyamatra összpontosuljon. A megoldás alkalmazásfüggetlenül adaptálható, így általános mintát kínál a felhőalapú automatizálás oktatási és ipari környezetben történő alkalmazásához.

Tartalomjegyzék

1. Bevezetés

1.1. A téma aktualitása, célkitűzések

A sebesség vált a XXI. századi gazdasági környezetben az alkalmazkodóképesség és a versenyképesség alapfeltételévé. Különösen a szoftverfejlesztésben, ahol a digitalizáció kényszeríti a vállalatokat gyorsabb innovációra.

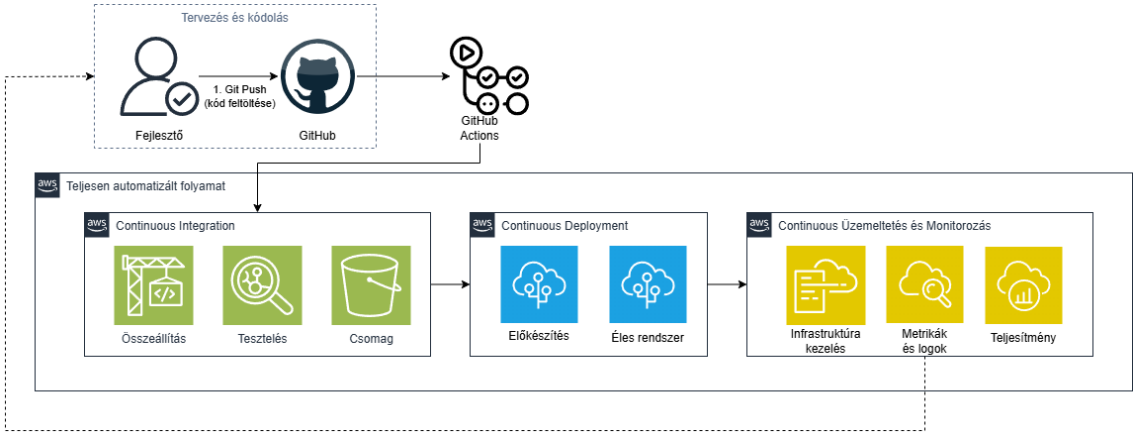
Pontosan erre a problémára született válaszul a DevOps kultúra. A DevOps lényege nem az eszközökben, hanem a gondolkodásmódban rejlik. Célja, hogy az együttműködés, a kommunikáció és ami a legfontosabb, az automatizáció révén végre áthidalja a fejlesztői (Dev) és az üzemeltetési (Ops) csapatok közötti, már-már közismertnek mondható szakadékot [1].

Ezen a területen (és a dolgozatom fókuszában) a folyamatos integráció (CI) és a folyamatos szállítás/bevezetés (CD) gyakorlata áll. A CI/CD célja leegyszerűsítve az, hogy a fejlesztők ne csak a "fióknak" dolgozzanak, hanem naponta többször integrálják a kódjukat a közös repozitóriumba. Ideális esetben ez a lépés azonnal elindít egy automatikus build-folyamatot és egy tesztsorozatot.

A Red Hat összefoglalója kiválóan rávilágít egy fontos részletre: a „CD” rövidítés alatt két, egymástól élesen elkülönülő fogalmat is értünk. Beszélhetünk folyamatos szállításról (Continuous Delivery), ami automatikusan előkészíti a kibocsátásra kész csomagokat, de a "gombot" még ember nyomja meg. Ennél egy lépéssel tovább megy a folyamatos bevezetés (Continuous Deployment), ami már a produkciós telepítést is teljesen automatizálja. Ezen gyakorlatok egyértelmű, mérhető előnyökkel járnak: gyorsabb a kihelyezés és drasztikusan csökken a hibák kockázata. Ugyanakkor egy biztonságkritikus rendszernél a folyamatos szállítás használandó. [2]

A szakdolgozatom gyakorlati célja egyértelmű volt: egy Python/Django alapú webalkalmazás CI/CD pipeline-jának teljes körű megtervezése és megvalósítása. Nem csak elméletben, hanem a gyakorlatban. A választott eszközök a GitHub Actions - a CI/CD folyamatok motorja - és az Amazon Web Service (AWS) Elastic Beanstalk PaaS szolgáltatása lettek. A projekthez stabil, hosszú távon támogatott (LTS) verziókat kerestem, így esett a választás a Python 3.11-re és a Django 4.2 LTS-re [3]. A felhőszolgáltatóval történő biztonságos hitelesítés kulcsfontosságú volt; ezt az OpenID Connect (OIDC) protokoll segítségével oldottam meg, elkerülve a statikus, lejáratra

hajlamos kulcsok használatát. A forráskód nyilvánosan elérhető GitHub-on - a kudarcokkal és a sikerekkel együtt - ez lehetővé teszi a reprodukálhatóságot és az ellenőrzést. Az 1. melléklet lépésről lépésre, végigvezet a teljes telepítési és integrációs folyamaton. Ezt olyan stílusban próbáltam megírni, hogy akár a témával ismerkedő számára is könnyen követhető legyen. A végeredmény egy olyan rendszer, ahol a kód módosításától a felhőben történő éles telepítésig minden lépés emberi beavatkozás nélkül, automatikusan fut le, ahogy az 1. ábra mutatja.



1. ábra A CI/CD pipeline elvi modellje AWS architektúrában (2025)
(Forrás: Amazon Web Services [4] alapján a szerző által átdolgozva)

Az elmúlt hónapokban komplex technikai kihívást jelentett a pipeline kialakítása. A GitHub Actions integrációja az AWS Elastic Beanstalkkal rugalmas és intuitív megoldást kínált, ugyanakkor a szolgáltatók közötti OIDC-hitelesítés beállítása komoly nehézség és tanulási folyamat volt. Megfigyeltem, hogy a gyors visszajelzés és az automatikus telepítések nemcsak a fejlesztési kedvet növelik, hanem a hibák korai felismerése révén valós üzleti előnyt is jelentenek.

1.2.A dolgozat felépítése, módszertan

Szakdolgozatom hét fő fejezetből áll, melyek logikusan követik egymást a problémakör felvezetésétől a gyakorlati megvalósításon át az eredmények értékeléséig. A bevezetést követő Elméleti háttér fejezetben tisztázom a DevOps-szemlélet, a CI/CD fogalmainak lényegét, valamint a pipeline felépítésének kulcselemeit. Ezután a harmadik fejezetben bemutatom a felhőalapú szolgáltatási és telepítési modelleket, az AWS Elastic Beanstalk működését és alternatív megoldásokat. A negyedik fejezetben

részletesen elemzem a legnépszerűbb CI/CD eszközöket, köztük a GitHub Actions, Jenkins, GitLab CI/CD és AWS CodePipeline-t, és összevetem őket költség, rugalmasság és támogatottság szempontjából. Az ötödik fejezet maga a gyakorlati megvalósítás lépéseit mutatja be, a projekt követelményeitől és a kódbázis felépítésétől kezdve a pipeline lépésein át a hibakezelésig. A hatodik fejezetben elemzem a pipeline teljesítményét, költségeit, skálázhatóságát és biztonságát, és végül a hetedik fejezetben összegzem az eredményeket, jövőbeli fejlesztési irányokat és személyes reflexióimat. A mellékletek a rendszer reprodukcióját, forráskódjának megértését és a rendszer teljesítményének megértését segítik.

Tekintettel a tématerület rendkívül dinamikus fejlődésére, a szakirodalmi áttekintés - a hagyományos, nyomtatott kiadványok helyett, melyek a gyors elavulás veszélyének vannak kitéve - elsősorban a legaktuálisabb, naprakész online forrásokra támaszkodik. Ezek közé tartoznak a hivatalos gyártói dokumentációk, iparági blogok és a nyílt forráskódú közösségek által publikált gyakorlati útmutatók, melyek híven tükrözik a terület jelenlegi állását. A gyakorlati rész a saját projekt implementációján alapul: a forráskód és a workflow-k elemzése során a GitHub repozitóriumon keresztül ellenőriztem a pipeline beállításait, az AWS konzolon pedig a telepítési és konfigurációs lépéseket.

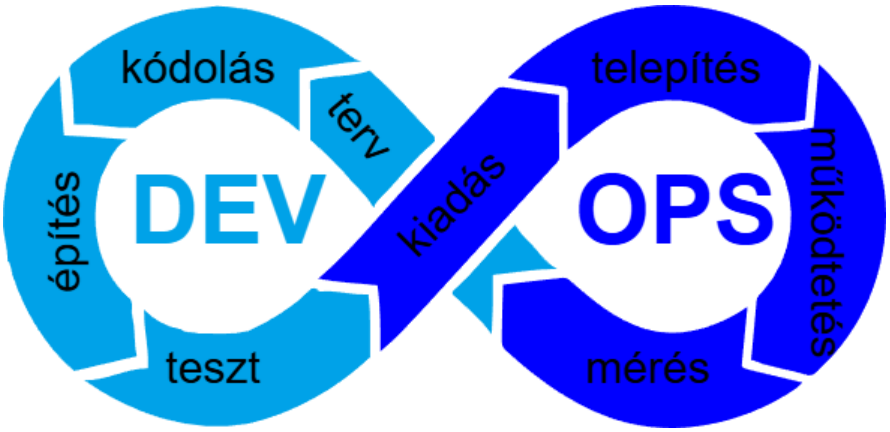
Amikor a szakirodalom és a gyakorlat közötti hidat építettem, rá kellett jönnöm, hogy a megvalósítás során sok elméleti fogalom konkrét jelentést nyer. Például a „folyamatos integráció” gyakorlati értékét akkor értettem meg igazán, amikor a pipeline első futásai során a hibák automatikusan napvilágra kerültek. A strukturált felépítés segített abban is, hogy a kutatás során ne vesszek el a részletekben, hanem minden fejezetet tudatosan, a célnak megfelelően alakítsak ki.

A szakdolgozat központi kérdése, hogy hogyan valósítható meg egy biztonságos és teljesen automatizált CI/CD pipeline felhőalapú környezetben.

2. Elméleti háttér

2.1. A DevOps és CI/CD alapfogalmai

A szoftverfejlesztési életciklus (SDLC) a fejlesztés gerincét alkotja. Ez egy strukturált, több fázisból álló folyamat, amely a szoftvertermék ötletétől annak üzemeltetéséig és karbantartásáig vezeti végig a fejlesztőket. Az SDLC fő szakaszai: a követelményelemzés, a tervezés, a megvalósítás, a tesztelés és a telepítés, majd a fenntartás [5]. E lépések együtt biztosítják, hogy a végtermék megbízhatóan működjön és megfeleljen az ügyfél elvárásainak. Egy jól definiált életciklus átláthatóságot és ellenőrizhetőséget teremt, ami különösen fontos a minőségbiztosítás és a költséghatékonyság szempontjából. A modell különböző formákban valósulhat meg - például vízesés-, agilis-, iteratív- vagy DevOps-alapú megközelítésben -, amelyek mind a projekt természetéhez igazíthatók. [6]



2. ábra a DevOps életciklus

(Forrás: iparági szabvány alapján a szerző által átdolgozva)

A DevOps a „Development” és „Operations” szavak összeolvadásából született, de valójában többet jelent technikai együttműködésnél. Egy olyan kultúra és gondolkodásmód, amely a fejlesztési és üzemeltetési csapatok közötti falakat lebontva a gyorsabb és stabilabb szoftverszállítást célozza. A klasszikus DevOps életciklust a 2. ábrán rajzoltam le. A DevOps alappillérei az együttműködés, a kommunikáció, az automatizáció és a folyamatos visszajelzés. A régi modellben gyakori volt, hogy a fejlesztők „átdobták a kerítésen” a kódot az üzemeltetőknek, ami időveszteséget és feszültséget okozott. A DevOps ezzel szemben közös felelősségvállalásra épít:

automatizált pipeline-ok, rendszeres tesztelés és folyamatos tanulás biztosítják, hogy a szoftver mindig készen álljon a telepítésre. Saját tapasztalat alapján az OIDC-hitelesítés és a telepítési beállítások csak akkor haladtak hatékonyan, amikor a „fejlesztői” és „üzemeltetői” feladatokat összehangoltam - ez mutatta meg igazán, mit jelent a DevOps együttműködés a gyakorlatban.

2.2. Folyamatos integráció (CI)

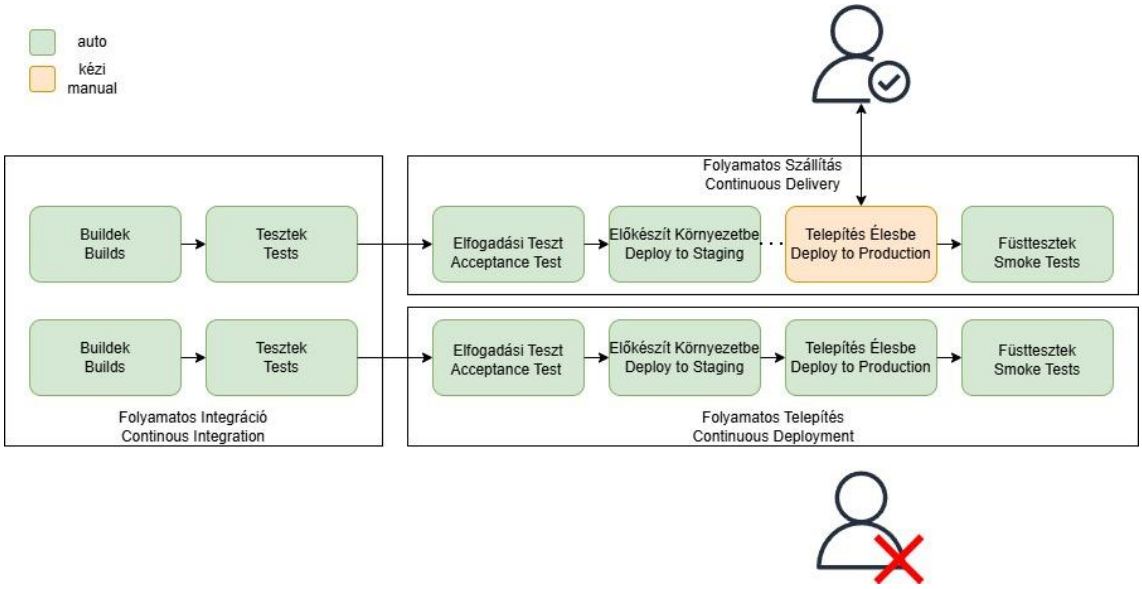
Az IBM leírását vegyük alapul, hogy a folyamatos integráció (CI) olyan fejlesztési gyakorlat, amelyben a programozók rendszeresen, akár naponta többször is integrálják a változtatásokat egy központi repozitóriumba. Minden commit automatikusan elindítja a buildelést és a teszteket, így a hibák már korán kiderülnek. A CI segít megelőzni a „build törés” jelenséget, amikor az új kód összeomlasztja a projektet, és csökkenti az összeolvasztási problémákat („merge hell”). A gyors visszajelzés a CI legnagyobb értéke: az automatizált tesztek pillanatokon belül jelzik, ha valami nem működik. [7]

A CI rendszerek alapját a verziókezelés adja (Git), melyet különféle automatizáló eszközök figyelnek (Jenkins, GitHub Actions, GitLab CI). Ezek a szerverek minden új kódváltozás esetén elindítják a build folyamatot: lefordítják a kódot, csomagolják az alkalmazást, majd futtatják az egység- és integrációs teszteket. A statikus kódelemzés (többek között biztonsági vizsgálatok) már a futtatás előtt figyelmeztethet a potenciális hibákra. Ezek a folyamatok teszik lehetővé, hogy a fejlesztés gyors, de biztonságos legyen. Amikor elkészítettem az első GitHub Actions ci.yml workflow-t, az azonnali visszajelzések segítettek felismerni a kódstílus-hibákat és a tesztelési hiányosságokat - ez a gyakorlatban több órányi kézi hibakeresést is megspórolhat.

2.3.Folyamatos szállítás és bevezetés (CD vs CD)

A CI/CD folyamat második lépcsője a folyamatos szállítás (Continuous Delivery), amely a tesztelt build kiadásra való előkészítését automatizálja. A Red Hat szerint ennek lényege, hogy a rendszer a kód összeolvasztásától kezdve a telepíthető artefaktum elkészítéséig mindent automatizál, de a végső élesítés még emberi döntéshez kötött [2]. Ezzel szemben a folyamatos bevezetés (Continuous Deployment) már a telepítést is automatizálja: ha minden teszt sikeres, a változtatások azonnal meg is jelennek az éles rendszerben. A különbséget a 3. ábra szemlélteti. A két módszer közti választás attól függ, mekkora kockázatot vállal a szervezet és mennyire szigorú a szabályozási környezet. Ebben a projektben a teljesen automatizált megoldás volt a cél: a pipeline automatikusan

elkészítette a telepíthető csomagot, és deploy előtt már nem végeztem kézi ellenőrzést. A Continuous Deployment bevezetésének egyik legjelentősebb gyakorlati előnye a manuális telepítési feladatok teljes kiiktatása. Ezért is törekedtem prioritásban, hogy a CD ágat minél előbb beállítsam.



3. ábra A Folyamatos Szállítás és a Folyamatos Telepítés összehasonlítása (Forrás: RedHat [2] alapján a szerző által átdolgozva)

2.4.A CI/CD pipeline felépítése

A CI/CD folyamat központi eleme a verziókezelés! A Git és a branch-stratégiák (a feature branch és GitFlow) segítenek a csapatmunkában és a változások visszakövetésében. A Pull Requestek biztosítják a kódáttekintést és a minőségellenőrzést mielőtt bármi a fő ágba kerülne. A trunk-based megközelítés, amelynél a fejlesztők gyakran commitolnak a fő ágba, tovább gyorsítja az integrációt és rövidíti a hibajavítási időt. A gyakorlatban, ahogy szakmán belül ez bevett módszer: rendszeresen teszteltem a kódot még a merge előtt, így sosem történt „build törés”.

Az automatizált tesztelés a következő kulcslépés. A pytest-tel írt unit és integrációs tesztek ellenőrizték, hogy az új funkciók nem rontják el a már működő részeket. A flake8 segít a kódstílus egységesítésében, a bandit pedig biztonsági ellenőrzéseket végzett. Mindez a GitHub Actions workflow-ba építve minden push után lefutott, így a hibákat gyorsan azonosítottam. A build során keletkező artefaktumok (konténerképek, ZIP fájlok) külön tárolókba kerülnek (AWS S3, JFrog Artifactory), ami biztosítja a verziózást és visszakövethetőséget. A függőségek gyorsítótárazásának bevezetése (GitHub Actions

cache / cache: pip) a projektben ~44%-kal csökkentette a CI futási időt (0,48 percről 0,27 percre), ahogy az a 3. melléklet 1. részében bemutatott mérésekből is látható. Komplex, több szolgáltatásból álló rendszernél ez a nyereség a naponta futó sok build és teszt miatt összességében órákra skálázódhat egy csapat szintjén.

Az infrastruktúra, mint kód (IaC) szemlélet a gyakorlatban is megjelent. Tökéletes példa erre az .ebextensions/django.config fájl. Bár csak néhány soros, mégis deklaratív módon rögzíti a környezet egy kritikus beállítását (a WSGI-útvonalat). Ez biztosítja, hogy a rendszer bármikor újraépíthető legyen azonos paraméterekkel. Ez a kiszámíthatóság nemcsak stabilitást, hanem valódi biztonságérzetet is adott: tudtam, hogy egy hiba esetén az új példány garantáltan a helyes konfigurációval indul újra.

2.5. Megfigyelhetőség és monitoring

Az IBM meghatározása kimondja: a megfigyelhetőség (observability) három pillére a metrikák, a logok és a trace-ek, ahogyan a 4. ábra mutatja. Ezek együtt segítenek a rendszer belső állapotának megértésében külső jeleiből. A metrikák (hibaarány, válaszidő, CPU-használat) számszerű információt adnak, a logok időrendben rögzítik az eseményeket, a trace-ek pedig megmutatják, hogyan halad végig egy kérés a rendszeren. A felhős és elosztott környezetekben ez kulcsfontosságú, mert a hibák gyakran nem egyértelműen egy komponenshez kötődnek. [8]

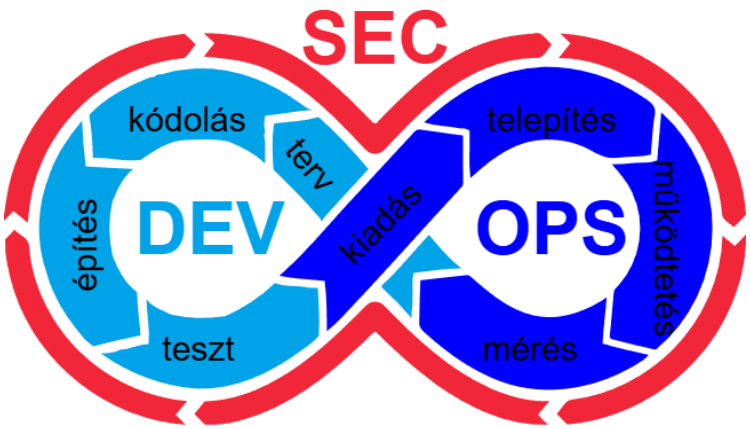
4. ábra A megfigyelhetőség három pillére

(Forrás: Az IBM definíciója [8] alapján a szerző által átdolgozva)

A monitoring eszközei lehetővé teszik a folyamatos felügyeletet és a proaktív beavatkozást. A riasztási küszöbök (CPU, latency, hibaarány) segítenek a gyors reakcióban, az MTTD és MTTR mutatók pedig mérik, mennyi idő telt el az észlelés és a helyreállítás között. A pipeline-ban az AWS CloudWatch-ot használtam a logok és metrikák gyűjtésére; a riasztások emailben és Slack-en is érkeztek, így azonnal tudtam reagálni. Gyakorlatilag ez annyit jelent, hogy nem kellett várakozni a számítógép előtt, hanem ha felállt vagy hibára futott a rendszer, az AWS CloudWatch szolgáltatótól email jött. Ez az automatikus figyelés sokkal rövidebbre csökkentette az incidensek felismerési idejét, és segített a teljesítmény optimalizálásában is. A teljesítményre egy valami volt nagyon negatív hatással a biztonsági ellenőrzések bevezetése, melyet a következő részben részletezek.

2.6. Biztonság a CI/CD-ben (DevSecOps)

A CI/CD folyamat egyik legfontosabb, hanem a legfontosabb területe a biztonság. Az 5. ábrával is szeretném aláhúzni, a modern DevSecOps megközelítésben a biztonság (SEC) már nem egy utólagos lépés, hanem a teljes fejlesztési (DEV) és üzemeltetési (OPS) ciklust átszövő, folyamatos tevékenység. A fejlesztési és üzemeltetési automatizmusok mellett kiemelt figyelmet igényel a titkos adatok kezelése, a hitelesítés és a jogosultságkezelés. A GitHub Actions lehetőséget biztosít arra, hogy a fejlesztő érzékeny adatokat - például API-kulcsokat, adatbázisjelszavakat - titkos változóként (secrets) kezeljen, így azok nem jelennek meg a naplókban és nem kerülnek a forráskódba.



5. ábra A DevSecOps életciklus modellje, ahol a biztonság a teljes folyamatot integrálja (Forrás: iparági szabvány alapján a szerző által átdolgozva)

A projektben, a jelenlegi csúcstechnológiás, ennél is biztonságosabb megközelítést alkalmaztam: az OpenID Connect (OIDC) alapú hitelesítést. Ez a módszer lehetővé teszi, hogy a pipeline futása közben ideiglenes tokeneket kérjen az AWS Security Token Service-től. Ennek előnye, hogy nincs szükség statikus, hosszú élettartamú kulcsokra, amelyek könnyen kompromittálódhatnak. Az AWS Identity and Access Management (IAM) szerepkörök precíz beállítása azonban elengedhetetlen: ha a jogosultság túl szűk, a pipeline lefagyhat, ha viszont túl tág, az biztonsági kockázatot jelent. Hosszabb kísérletezés után sikerült megtalálni az optimális egyensúlyt, ahol a futtatáshoz szükséges engedélyek megvoltak, de semmi nem volt feleslegesen hozzáférhető.

A biztonság nem csupán a titokkezelésre korlátozódik. A DevSecOps szemlélet szerint a biztonsági ellenőrzéseknek már a fejlesztési ciklus elején meg kell jelenniük („shift-left security”). A statikus kódelemző eszközök - például a bandit Python projektekhez - képesek már a fejlesztés során észrevenni ismert sebezhetőségeket, rossz kódmintákat vagy gyenge titkosítási megoldásokat. A projekt során a bandit integrálva volt a CI pipeline-ba, így minden push után automatikusan lefutott, és figyelmeztetett azokra a kódrészletekre, amelyek potenciálisan biztonsági kockázatot hordozhattak.

A megfelelőség (compliance) szintén kulcskérdés a felhőalapú fejlesztésben. Az AWS Shared Responsibility Model világosan kimondja, hogy a felhőinfrastruktúra biztonságáért a szolgáltató, míg az alkalmazás és az adatok védelméért a felhasználó felel. Ezért a CI/CD pipeline tervezése során a fejlesztő felelőssége, hogy betartsa a vonatkozó adatvédelmi és jogszabályi előírásokat (például a GDPR-t), különösen akkor, ha személyes adatokat kezel.

A DevOps magas szintű automatizálása ugyanakkor kockázatokat is hordoz. Egy hibás konfiguráció, például egy rosszul megadott IAM-policy vagy egy engedélyezett, de nem ellenőrzött deploy script, gyorsan nagyobb károkat okozhat, mint egy hagyományos kézi telepítés. Emellett a vendor lock-in, vagyis a beszállítói függőség veszélye is jelen van: az AWS-specifikus szolgáltatások, mint például az Elastic Beanstalk, nehezen vihetők át más felhőplatformokra. Ennek enyhítésére célszerű nyílt szabványokat (pl. Terraform) és többfelhős (multi-cloud) stratégiát használni, de ez a jövőbeli fejlesztések résznél jobban kifejtem.

A szerző tapasztalatai alapján a biztonsági konfiguráció volt az egyik legérzékenyebb és legtöbb finomhangolást igénylő rész. Több iterációra volt szükség

ahhoz, hogy az IAM-policy csak a feltétlenül szükséges erőforrásokhoz adjon hozzáférést (1. melléklet 78.-82. lépés). Kezdetben a statikus AWS-kulcsok miatt többször előfordult jogosultsági hiba, de az OIDC átállás után ezek megszűntek. A bandit vizsgálatok szintén hasznosnak bizonyultak: olyan rejtett sebezhetőségeket is jeleztek (például felesleges „eval” hívásokat), amelyek első ránézésre ártalmatlannak tűntek.

Összességében a DevSecOps megközelítés a projekt során nem csupán biztonsági követelmény volt, hanem gyakorlati értéket is adott: a pipeline megbízhatóbban működött, a hibaelhárítás gyorsabbá vált, és minden futtatás után pontos képet kaptam arról, hogy a kód megfelel-e a biztonsági elvárásoknak.

3. Felhőalapú környezetek

3.1. A felhő fogalma és jellemzői

Felhőszámítás lényege, hogy a számítási erőforrások (szerverek, tárhely, alkalmazások) hálózaton keresztül, igény szerint rugalmasan elérhetők. Ahogy a NIST (National Institute of Standards and Technology) fogalmaz, a felhő lényege az igény szerinti, hálózaton keresztül elérhető erőforrás. A gyakorlatban leginkább a gyors skálázhatóság érződik: például néhány kattintással indítható új példány, ami korábban órákat vett igénybe. A modell öt alapvető jellemzője:

1. on-demand self-service - a felhasználó emberi beavatkozás nélkül kérhet erőforrást
2. széles körű hálózati elérés - különböző eszközökről szabványos protokollokon
3. erőforrás-megosztás (pooling) - a szolgáltató dinamikusan osztja el a kapacitást több ügyfél között;
4. gyors rugalmasság (rapid elasticity) - a kapacitás felfelé és lefelé is gyorsan skálázható
5. mért szolgáltatás (measured service) - a fogyasztás automatikusan mérhető és elszámolható

Ezek együtt különítik el a felhőt a saját tulajdonú, fix kapacitású infrastruktúrától, és emiatt lett ideális alap a rugalmas, költségtudatos rendszerekhez. [9]

3.1.1. Szolgáltatási modellek (IaaS, PaaS, SaaS)

Az „as-a-Service” modellek a felhőszolgáltatások rétegeit írják le, melyek főbb jellemzőit és különbségeit az 1. táblázat foglalja össze.

IaaS: esetén a szolgáltató adja a számítási, hálózati és tárolási erőforrásokat; a felhasználó saját operációs rendszert és alkalmazásokat futtat, de a fizikai infrastruktúra üzemeltetése nem az ő gondja, itt szinte minden egyes részfeladat ránk hárul, cserébe teljes kontrollt kapunk.

PaaS ennél magasabb absztrakció: komplett futtatókörnyezetet kapunk (OS, web-/alkalmazáskiszolgálók, adatbázis-szolgáltatások, fejlesztői eszközök), így a fejlesztés az üzleti logikára koncentrálhat.

SaaS esetén kész alkalmazást használunk böngészőn vagy API-n keresztül, a szolgáltató menedzsel mindent a háttérben, cserébe a konfigurációs lehetőségek minimálisak.

	IaaS (Infrastructure)	PaaS (Platform)	SaaS (Software)
definíció	Alapvető számítási-, hálózati- és tárolási erőforrásokat biztosít	Teljes fejlesztői és futtatókörnyezetet biztosít, elrejtve az alatta lévő infrastruktúrát.	Kész, azonnal használható szoftveralkalmazást biztosít a felhőn keresztül.
szolgáltató felelőssége	**Infrastruktúra:** • Virtualizáció • Fizikai szerverek • Hálózat • Adattárolás	**Platform:** • *Minden, ami IaaS* • Op. rendszer (OS) • Futtatókörnyezet (pl. Java, Node.js) • Middleware	**Alkalmazás:** • *Minden, ami PaaS* • Maga a szoftveralkalmazás • Alkalmazásadatok
ügyfél felelőssége	• Alkalmazások • Adatok • Futtatókörnyezet • Köztesréteg • Operációs rendszer (OS)	• Alkalmazások • Adatok • (Konfiguráció)	• A szolgáltatás használata • Felhasználói adatok kezelése • Hozzáférés-kezelés
példák	• **AWS EC2** • Google Compute Engine • Microsoft Azure VMs	• **AWS Elastic Beanstalk** • AWS Lambda • Heroku • Google App Engine	• Microsoft 365 • Google Workspace (Gmail) • Salesforce • GitHub
előny	• **Magas rugalmasság:** Teljes kontroll az OS és a futtatókörnyezet felett. • Részletes testreszabhatóság.	• **Gyors fejlesztés:** A fejlesztők az alkalmazáskódra fókuszálhatnak, nem az infrastruktúrára. • Automatikus skálázódás.	• **Azonnali használat:** Nincs szükség telepítésre vagy karbantartásra. • Bárhonnan elérhető.
hátrány	• **Magas karbantartási teher:** Az ügyfél felel az OS frissítéséért, biztonsági mentéséért. • Bonyolultabb menedzsment.	• **Korlátozott rugalmasság:** Kötöttség a platform által kínált nyelvekhez és szolgáltatásokhoz. • Kisebb kontroll az OS felett.	• **Minimális testreszabhatóság:** Csak a szolgáltató által engedélyezett beállítások módosíthatók. • Erős függőség (vendor lock-in).

1. táblázat A felhőszolgáltatási modell (IaaS, PaaS, SaaS) jellemzői és összehasonlítása (Forrás: NIST [9]és IBM [10]definíciói alapján, a szerző által szerkesztve)

A klasszikus hármas mellett terjed a FaaS (Function as a Service) is, ahol eseményre futtatott függvényeket telepítünk; a díjazás tipikusan a futási időhöz és az igénybevett erőforráshoz igazodik. A szerver nélküli (serverless) modell extrém skálázást tesz lehetővé, viszont az állapotkezelés és az életciklus-kontroll szűkebb keretek közé kerül.

A demonstrációs rendszerben PaaS irányba mozdultam, mert gyors bevezetést akartam - de néha hiányzott az alacsonyabb szintű kontroll, amit az IaaS ad. [9] [10] [11]

3.1.2. Telepítési modellek (public, private, hibrid, multi-cloud)

A telepítés négy tipikus modellben történhet. A nyilvános (public) felhő több-bérlős: a szolgáltató üzemelteti az infrastruktúrát, az ügyfelek interneten keresztül érik el. Előnye az alacsony belépési költség és a rugalmas skálázás, hátránya a korlátozott

hardver-kontroll és az adatkezelési aggályok. A privát felhő dedikáltan egy szervezetet szolgál ki; magasabb a megfelelőség és az irányítási szint, viszont drágább és kevésbé rugalmas. A hibrid felhő a kettő kombinációja, adat- és munkaterhelés-mozgatással, ami rugalmasságot ad, de bonyolultabb menedzsmentet és biztonsági felügyeletet igényel. A multi-cloud megközelítés több szolgáltatót használ (például csak a „Big Three”-t említve: AWS, Azure és Google Cloud), csökkenti a vendor lock-in kockázatát, ugyanakkor növeli a szinkronizáció és a költségkövetés komplexitását. A public cloud előnye az alacsony költség (felmerül a kérdés vajon megéri az adatbiztonsági aggályokat). A feladat megkövetelte, hogy mérnöki döntést hozzak, ezért skálázhatóság és a szolgáltatási ökoszisztéma döntött, miközben a telepítéseket, a konfigurációt igyekeztem deklaratívvá tenni, hogy később akár multi-cloud irányba is nyitni lehessen. [9]

3.2. AWS mint felhőszolgáltató

Az Amazon Web Services piacvezető nyilvános felhőplatform globális infrastruktúrával és széles szolgáltatásportfólióval. Az alapok közül az EC2 rugalmas virtuális gépeket ad, az S3 objektumtárhelyet, az RDS menedzselt relációs adatbázist, a Lambda szerver nélküli függvényfuttatást, az ECS/EKS konténer-orchesztrációt, az Elastic Beanstalk pedig PaaS-szerű környezetet kínál, ezt a 6. ábrán szemléltetem. A világméretű hálózat földrajzi régiókra és azokon belül több, egymástól fizikailag szeparált rendelkezésre állási zónára (Availability Zone) tagolódik [12]. A régiók és zónák rendszere zseniális, mert elkerüli a hibapontokat - project során is bevált, sosem volt még downtime.

6. ábra AWS Cloud néhány alapvető szolgáltatása (Forrás: saját szerkesztés)

A régiókat nagy sebességű gerinchálózat köti össze, a zónák redundáns energia-, hálózat- és hűtési infrastruktúrával rendelkeznek, így a magas rendelkezésre állás és a „katasztrófaállóság” tervezhető. A saját használatban ez azt jelentette, hogy több zónára támaszkodva el tudtam kerülni az egyetlen hibapontot.

3.2.1. Alapszolgáltatások és régiók

Az EC2 a skálázható számítási kapacitás, a Lambda az eseményvezérelt szerver nélküli futtatás, az ECS és az EKS pedig a konténeres alkalmazások menedzsmentjének alapjai [13]. A Beanstalk alkalmazásszintű platformként automatikusan létrehozza és összedrótozza a szükséges erőforrásokat (EC2 példányok, load balancer, auto scaling csoport), így a fejlesztő a kódra fókuszálhat. Ez a gyakorlatban azt jelentette, hogy az első futtatáskor a Beanstalk szó szerint minden háttérkomponenst felépített helyettem. Régióválasztásnál a földrajzi közelség, a szolgáltatás-elérhetőség és az ár is mérlegelendő, miközben a régiók közötti izoláció segít az adatlokalitási és megfelelőségi követelmények teljesítésében Saját döntésemnél a késleltetés volt a fő szempont - végül az eu-north-1 mellett kötöttem ki, mert olcsóbb és legközelebb van és egyből érezhető volt, hogy lecsökkent a szerver válaszideje. [14]

3.2.2. Biztonsági és költségmodellek

Az AWS biztonsági modellje a megosztott felelősség elvén áll: a szolgáltató a fizikai és virtualizációs réteget védi, a felhasználó feladata az alkalmazások, adatok és konfigurációk helyes beállítása (IAM, hálózati szabályok, titokkezelés) [15]. Az AWS modellje szerint a szolgáltató védi a vasat, azaz a fizikai és virtualizációs réteget, mi pedig az appot. A gyakorlati munka során jól látható, hogy egy hiányzó IAM-szabály könnyen jogosultsági hibát eredményez, ami tovább erősíti a pontos konfiguráció szükségességét.

Költségoldalon a pay-as-you-go alapértelmezés mellett léteznek megtakarítási modellek: hosszabb távú elköteleződéssel (Reserved Instances, Savings Plans) jelentős kedvezmény érhető el; van Free Tier kipróbáláshoz; és elérhetők a Spot/On-Demand opciók eltérő költség-kockázati profillal [16]. A gyakorlatban Free Tier módon indultam, majd gyorsan kiderült, hogy a nem optimális példánytípusok felhajtják a költséget - ezért a monitorozás és a jogosultságok (OIDC-vel támogatott, minimál-jogosultságú IAM-szerepkörök) gondos megtervezése elengedhetetlen.

A saját futtatásban az eu-north-1 régió és az auto scaling együtt olyan alapot adott, amely rugalmasan reagált a terhelés változásaira. A tanulsága az volt, hogy fejlesztés

közben is érdemes „kicsiben” indulni, majd csak szükség esetén feljebb skálázni. A megosztott felelősség modell pedig józan emlékeztető: a biztonság nem „delegálható” teljesen - a kulcsok, titkok és jogosultságok nálunk maradnak felelősségben.

3.3. AWS Elastic Beanstalk bemutatása

Az Elastic Beanstalk menedzselt PaaS-megoldás webalkalmazások és háttérszolgáltatások egyszerű telepítéséhez. A CloudOptimo összefoglalója szerint a Beanstalk automatikusan összeállítja a szükséges infrastruktúrát (EC2, auto scaling, load balancer), és integrálódik a kapcsolódó AWS-szolgáltatásokkal (pl. CloudWatch), így a fejlesztőnek jellemzően az alkalmazás csomagolásával és néhány konfigurációs paraméterrel kell foglalkoznia, ezt a 7. ábrával szemléltetem. Úgyis felfoghatjuk, hogy a 6. ábra utolsó dobozában van benne a 7. ábra.

A platform a népszerű nyelveket (Python, Java, Node.js, Ruby, Go, PHP) natívan támogatja, és lehetőséget ad finomhangolásra is, például .ebextensions fájlokkal vagy Procfile használatával. A gyakorlatban a konkrét feladatnál ez azt jelentette, hogy a platform 'elnyelte' a bonyolultságot - de maradt kontroll személyre szabott beállításokhoz. Ez jól jött, mert például a gunicorn beállításaimat nem a default értékekkel akartam futtatni. [17] [14]

7. ábra Az Elastic Beanstalk által menedzselt alapszolgáltatások
(Forrás: Amazon Web Services [14] alapján a szerző által átdolgozva)

3.3.1. Architektúra és működési elv

A Beanstalk-architektúra lényege a „környezet” (environment) és az „alkalmazásverzió” (application version) szétválasztása. Ez teszi a rendszert zseniálissá. Hiszen úgy tudtam gyorsan deployolni, hogy az infra miatt bármikor is aggódnom kellett volna.

A környezet tartalmazza a futtatókörnyezetet (EC2-példányok csoportja), a load balancert és az auto scaling csoportot. Az alkalmazásverzió telepíthető artefaktum (ZIP vagy konténerkép), amelyből a Beanstalk új kiadást készít. A host manager intézi a telepítést és az egészségfigyelést és a naplózást is. A rendszer a beállított alsó-felső példányszám között automatikusan skáláz, miközben a CloudWatch gyűjti a teljesítménymetriákat. Ez a felépítés tisztán elkülöníti az alkalmazás életciklusát a mögöttes infrastruktúrától, ami a működtetést kiszámíthatóvá teszi. Egy alkalommal a health check hibásan jelezte a példányt 'impaired'-nek - ekkor láttam igazán, mennyit ér a CloudWatch és az automatikus helyreállítás. [15]

3.3.2. Konfiguráció és skálázási lehetőségek

A Beanstalk több szinten konfigurálható: példánytípus, min-/max-példányszám, health check, hálózati beállítások; és finomhangolható a szoftverréteg is (WSGI útvonal, környezeti változók) YAML-alapú .ebextensions segítségével. A .ebextensions fájlban rögzítettem a Django beállításait - ez volt a kulcs, mert így mindent deklaratívvá tettem. Az auto scaling csoport CPU-, hálózati vagy egyedi metrika alapján is tud skálázni. A projektben a Python-alkalmazás WSGI beállításait és a statikus fájlok kiszolgálását az .ebextensions/django.config fájlban rögzítettem, a Procfile-ban pedig a gunicorn indítási parancsát definiáltam. Fejlesztői környezetben tipikusan 1-3 példány elegendő volt; éles üzemnél a minimum 2, de inkább 4 példány és a több zóna adta a kívánt rendelkezésre állást, különben kockázatossá válik. Ezt a skálázást először Locust-tesztelés közben figyeltem meg: amikor a pipeline új verziót indított, ekkor a példányszám tényleg automatikusan nőtt (3. melléklet 2. rész és 19. ábra mutatja). [14]

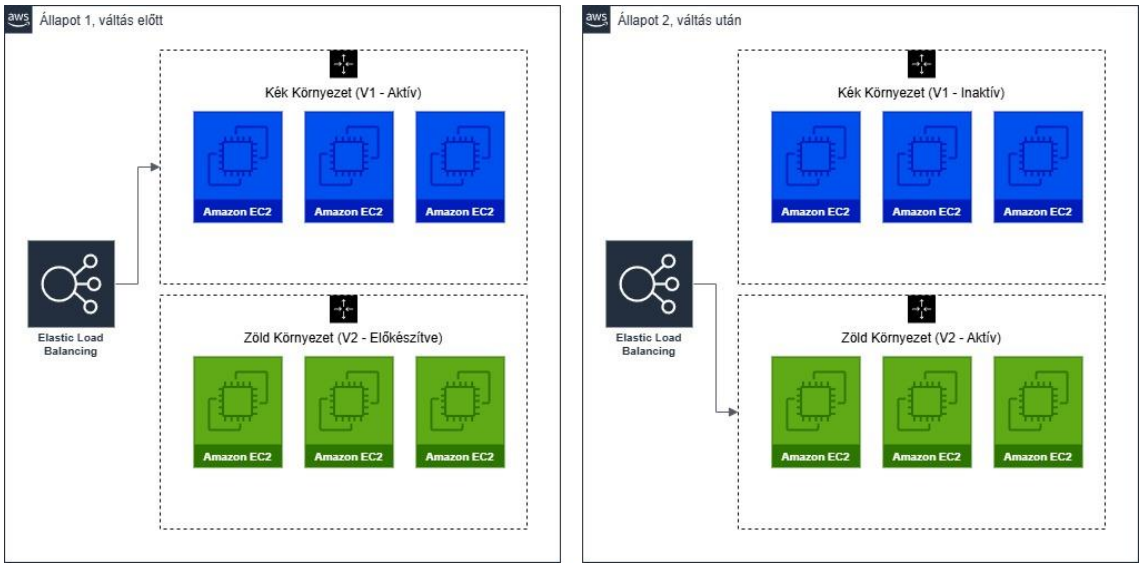
3.3.3. Telepítési stratégiák (blue/green, roll back)

A Beanstalk több frissítési modellt támogat. Az „all at once" gyors, de nagyobb leállási kockázattal jár. A „rolling" fokozatosan cserél példányokat, a „rolling with additional batch" ideiglenes plusz kapacitást is felvesz, az „immutable" külön auto scaling csoportba telepít, és csak siker esetén vált. A „traffic splitting" (canary) a forgalom kis részét tereli az új verzióra, és metrikák alapján dönt a kiterjesztésről. A blue/green stratégia külön környezetet használ az új verzióhoz, és DNS-szinten vált - ezzel gyors visszaállítás érhető el probléma esetén.

A fejlesztett folyamatban GitHub Actions-ból ZIP-be csomagolt artefaktummal indítottam (S3 feltöltés, create-application-version, majd update-environment). Ez az

update-environment parancs egy "in-place" (pl. 'Immutable' vagy 'Rolling') stratégiát indított, ami magában foglalja az automatikus visszaállítás lehetőségét. Egyszer hibás konfiguráció csúszott be; a Beanstalk beépített automatikus rollback funkciója pár perc alatt visszahozta a működő verziót, ami éles környezetben kritikus élmény. A rollback működését szó szerint valós időben néztem végig: 3,5 perc alatt visszaállt az előző verzió. Ez egy kritikus élmény volt, ami megmutatta, hogy még egy egyszerűbb 'in-place' frissítés esetén is van komoly biztonsági háló.

A 8. ábra ehhez kapcsolódóan egy fejlettebb, Blue/Green stratégiát szemléltet, ami még gyorsabb, kiesés nélküli átállást és azonnali visszaállást tenne lehetővé a CNAME cserével (a "nyíl visszakötésével"). Az általam tapasztalt 3-4 perces visszaállás ehhez képest az 'in-place' frissítés automatikus hibajavítása volt, de a konklúzió hasonló: Ha egy biztonság kritikus rendszer (például: 2025-ben történt egészségügyi felhő) meghiúsulása esetén a down time így is csak 3-4 perc, és nem 2 nap! [18]



8. ábra A Blue/Green telepítési stratégia folyamata
(Forrás: Amazon Web Services [18] alapján a szerző által átdolgozva)

3.4. Alternatív megoldások rövid áttekintése

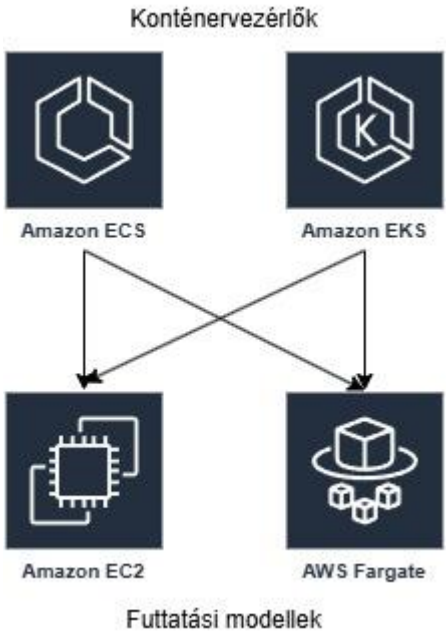
3.4.1. Azure App Service és Google App Engine

Az AWS mellett érdemes megemlíteni a piacon elérhető alternatív megoldásokat is. Az AWS mellett a Microsoft Azure App Service szintén PaaS-alapú web- és API-környezetet kínál. A Microsoft dokumentáció szerint több nyelvet támogat (.NET, Java,

Node.js, Python, PHP), konténerizált futtatást is, integrálható GitHub Actions-szel vagy Azure Pipelines-szal, és beépített staging-képességei vannak [19]. A költségmodell pay-as-you-go, a platform tartománykezelést, SSL-t és magas rendelkezésre állási SLA-t is ad. A Google App Engine hasonló, teljesen menedzselt PaaS: a népszerű nyelvek támogatása mellett automatikus skálázást és natív Cloud Monitoring/Logging integrációt kínál, valamint verziókezeléssel támogatja a fejlesztés-teszt-staging-éles életciklust [20]. Mindkét platformon előny, hogy az üzemeltetés terhei csökkennek, miközben a biztonsági és skálázási alapok „dobozból" érkeznek.

Összességében mindhárom platform (AWS, Azure, GCP) ugyanazokat az alapelveket - automatizációt, skálázhatóságot, PaaS-modellt - követi, de az AWS Elastic Beanstalk jelen projektben a nyíltabb integrációs lehetőségek (GitHub Actions OIDC, S3, CloudWatch) és a széles dokumentációs támogatás miatt bizonyult legmegfelelőbb választásnak.

3.4.2. Konténer-alapú PaaS megoldások (ECS, Fargate, Kubernetes)



9. ábra Az AWS konténer-futtatási modelljei
(Forrás: AWS könyv [15] alapján a szerző által átdolgozva)

A konténerizáció térnyerésével több, skálázható és hordozható alternatíva áll rendelkezésre. Az AWS ECS menedzselt Docker-orchesztráció, amely szorosan

illeszkedik az AWS ökoszisztémába, és futtatható EC2-n vagy Fargate-en. Az EKS Kubernetes-alapú megoldás: rugalmasabb és ökoszisztéma-gazdagabb, cserébe meredekebb a tanulási görbéje. A Fargate teljesen szerver nélküli konténerfuttatás, ahol a szolgáltató skálázza az erőforrást, a díj pedig a tényleges CPU- és memóriaidőhöz igazodik. Az ECS és EKS egyaránt használható Fargate módban, így kombinálható a Kubernetes funkcionalitása a serverless üzemeltetési modellel. A konténervezérlők (ECS/EKS) és a futtatási modellek (EC2/Fargate) közötti választási lehetőségeket a 9. ábra szemlélteti.

Alternatívaként a Kubernetes telepíthető szolgáltatófüggetlenül is (pl. GKE, AKS), ami csökkentheti a vendor lock-int, ugyanakkor nagyobb üzemeltetési terhet és komplexebb konfigurációt jelent [21].

A projekt technológiai stackjének kiválasztását több stratégiai szempont vezérelte. Bár mindhárom nagy platform (AWS, Azure, GCP) hasonló PaaS-alapelveket követ, a választásom több okból is az AWS-re esett. Először is, az AWS-ökoszisztéma mélyebb megismerése elengedhetetlen célkitűzés volt. Míg a Microsoft Azure és a Google Cloud platformjain már rendelkeztem tapasztalattal, az AWS piacvezető szerepe a nagyvállalati szektorban és a DevOps-mérnöki gyakorlatban alapvető elvárássá teszi az ismeretét. Másodszor, a projekt-specifikus célokat (automatizált CI/CD) az AWS Elastic Beanstalk nyíltabb integrációs lehetőségei (mint a GitHub Actions OIDC-hitelesítés, az S3 artefaktum-tárolás és a CloudWatch-monitoring) jobban támogatták, mint a szorosabban integrált versenytársak. Középtávon a Fargate bevezetése logikus következő lépés lehet, azonban ehhez a CI/CD-láncot is újra kell igazítani a konténeres életciklushoz. Multi- vagy hibrid-cloud stratégiában gondolkodva pedig érdemes már most felkészülni a különböző platformok eltérő működési modelljeinek kezelésére.

4. CI/CD eszközök áttekintése

Ebben a fejezetben bemutatom az olvasónak a legelterjedtebb CI/CD eszközöket, és összehasonlítom őket különböző szempontok alapján, előkészítve a terepet a későbbi fejezetben ismertetett saját megoldás eszközválasztásának indoklásához.

4.1. GitHub Actions

A GitHub Actions az egyik leggyakorlatibb CI/CD eszköz, szinte minden GitHub-eseményre automatizált választ adhatunk YAML-fájlok segítségével. Ha GitHubot használunk, akkor a GitHub Actions magától adódik. Egy workflow egy vagy több jobból áll, a jobok egymástól függetlenül - akár párhuzamosan - futnak. Könnyen meg lehet szeretni, tehát jelentősen javítja a fejlesztői élményt, hogy commit&push kérésre azonnal reagál a rendszer. Minden job friss, izolált futtatókörnyezetet kap (runner: VM vagy konténer), a lépések (steps) sorrendben hajtódnak végre, és a jobok a közös fájlrendszeren keresztül adhatnak át artefaktumokat. A GitHub több runner-típust kínál: GitHub-hostolt (Linux, Windows, macOS), nagy kapacitású gépek és saját, önhosztolt futtatók. A választást a runs-on kulcs határozza meg; minden job új, efemer környezetben indul és befejezéskor felszámolódik. Ez egyszerű skálázást ad, ugyanakkor a felhasználható percek a fióktípustól függenek. Az ingyenes keret GitHub Actions-nél meglepően elégnek bizonyult, (ha tudatosan építettem be a cache-lépéseket) a szűk keresztmetszet az AWS Free Tier volt. [22] [23]

4.1.1. Workflow felépítés, jobok és runner típusok

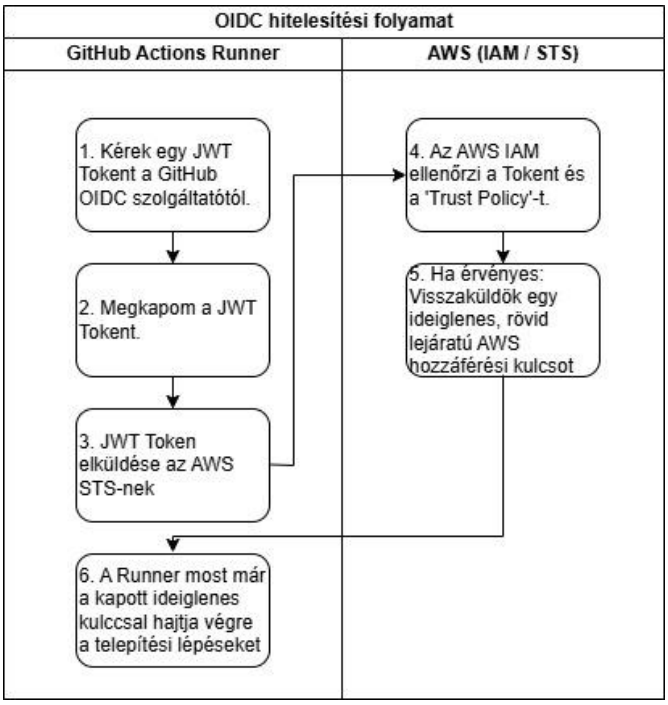
Egy workflow-ban megadhatjuk az eseményt, aztán jönnek a jobok - ezek függetlenül futnak, ami segített a gyorsabb buildben. A jobok párhuzamosan vagy egymás után futtathatók; ha egy job másik eredményére épít, megvárja annak befejezését. A lépések lehetnek parancsok (run:) vagy újrafelhasználható actionök, amelyek gyakori feladatokat automatizálnak (repo klónozás, környezet beállítása stb.). A GitHub Marketplace több ezer hivatalos és közösségi actiont tartalmaz, így a munkafolyamatok kódismétlés nélkül bővíthetők. A kidolgozott alkalmazásunkban ci.yml például Ubuntu runneren futtatja a Python 3.11 környezetet, telepíti a függőségeket, lefuttatja a flake8-at, a Banditot és a pytestet, majd egy rövid verzióellenőrzést is végez (2.melléklet ci.yml vagy GitHub repo). Emlékszem, hogy ennek a résznek a beállítását utólag készítettem el, de az adott beállításokkal semmi gond nem volt. A runs-on rugalmasan cserélhető, így a build-mátrix (külön OS-ek, Python-verziók) is könnyen beállíthatóvá vált.

4.1.2. Reusable workflows és marketplace actionök

Nagyobb kódbázisoknál a build/deploy lépések sokszor ismétlődnek. Ezt akkor értettem meg igazán, amikor két külön repóban kellett ugyanazt a buildet karbantartanom - az újrafelhasználható workflow-val elég lett egyetlen központi verzió. A kódduplikáció elkerülésére a GitHub újrafelhasználható workflow-kat kínál: egy teljes workflow (vagy annak része) központi repóban karbantartható, majd más workflow-k egy sorral hivatkozhatnak rá. A composite action ezzel szemben egyetlen jobon belüli lépések összefűzésére szolgál; a reusable workflow több jobot és saját runner-beállítást is tartalmazhat. [24]

4.1.3. OIDC integráció AWS szolgáltatásokkal

Az automatizált deployoknál a biztonságos hitelesítés kulcskérdés. 2022 óta a GitHub Actions támogatja az OpenID Connectet (OIDC), amellyel a workflow futása közben rövid élettartamú tokenek kérhetők az AWS-től - így nincs szükség hosszú távú, statikus kulcsok tárolására a Secretsben. A megoldáshoz a GitHub OIDC-providerét (https://token.actions.githubusercontent.com) és egy IAM-szerepkört kell konfigurálni, a bizalmi szabályzatban (trust policy) pedig megadni, mely repó/branch veheti át a szerepkört. [25]



10. ábra Az OIDC hitelesítés lépései az alkalmazásunkban

(Forrás: OIDC protokoll [25] logikája alapján a szerző által átdolgozva)

A statikus kulcs nélküli hitelesítés lépéseit a 10. ábra szemlélteti. A workflow-ban permissions: id-token: write szükséges, majd az aws-actions/configure-aws-credentials automatikusan lekéri a JWT-t és átadja az AWS-nek. A saját deploy.yml folyamatunkban ez így néz ki: OIDC-vel bejelentkezik, becsomagolja az appot, feltölti S3-ba, új Elastic Beanstalk verziót hoz létre, végül frissíti a környezetet (2. melléklet deploy.yml vagy GitHub). Első körben nem sikerült jól beállítani a trust policy-t - az AWS 'AccessDenied' hibával dobott vissza. Később jöttem rá, hogy az aud mezőt pontosan kell megadni a token.actions.githubusercontent.com-ra. Amikor először sikerült a teljes deploy OIDC-vel, kifejezetten fontos mérföldkő volt, hogy végre nincs több statikus kulcs a Secrets között és sikerült a legmodernebb, legbiztonságosabb azonosítást beépítenem. A szerző tapasztalatai szerint ez sokkal biztonságosabb, mint a régi kulcsos módszer.

4.2. Jenkins

4.2.1. Architektúra és pluginrendszer

A Jenkins egy régóta népszerű, nyílt forrású CI/CD szerver, amely Java-ra épül és sok szervezetnél még mindig alapvető szerepet tölt be. Az architektúrája master-agent modellre épül: a Jenkins szerver (master) menedzseli a konfigurációkat, időzíti a build-feladatokat és karbantartja az állapotot, míg az agentek (korábban slave-k) futtatják a konkrét feladatokat párhuzamosan. Elsőre bonyolultnak tűnhet, de a Jenkins agentek működését akkor értettem meg a gyakorlatban, amikor egy Windows-hoz tartozó agentem elkezdett 'offline' állapotban maradni - a logban derült ki, hogy Java-verzióütközés volt. A Jenkins bővíthető moduláris plugin rendszerének köszönheti rugalmasságát: több ezer hivatalos és közösségi plugin segíti a SCM integrációt, build eszközöket, értesítési csatornákat és felhő-szolgáltatásokat. Egy plugin például lehetővé teszi az S3-ba való feltöltést anélkül, hogy kézzel kellene AWS CLI parancsokat írnunk. A plugin rendszer azonban rendszeres karbantartást igényel; a plugin verziók inkompatibilitása vagy a Java ökoszisztéma frissítései könnyen instabillá tehetik a rendszert. A Jenkins egyik legnagyobb erőssége a bővíthetőség, ugyanakkor ez a tulajdonsága egyben a legfőbb gyengesége is, mert a jelentős karbantartási kockázatot von maga után. [26]

4.2.2. Deklaratív vagy scripted pipeline

A pipeline-ok a repóban lévő Jenkinsfile-ban definiálhatók Groovy szintaxissal. Két fő stílus terjedt el: deklaratív és script-alapú megközelítés. A deklaratív nyelv

blokkokra (agent, stages, steps) épül, ami jól olvashatóvá és karbantarthatóvá teszi, és csökkenti a hibázás esélyét, ezért ideális választás a témával még csak most ismerkedők számára. A scripted pipeline teljes Groovy-szkript, nagyobb szabadsággal (ciklusok, feltételek, metaprogramozás), de bonyolultabb karbantartással. Mindkettő támogatott; új projekteknél általában a deklaratív az ajánlott, míg régi rendszerekben gyakran marad a scripted megközelítés.

4.3.GitLab CI/CD

4.3.1. Runner kezelés és integrációs képességek

A GitLab CI/CD a platform integrált része: a pipeline-okat a repó gyökerében lévő .gitlab-ci.yml fájl írja le, és automatikusan futnak push után. A végrehajtást a GitLab Runner végzi, amely a GitLab instance-hez csatlakozik, felveszi a jobokat, futtatja őket, majd visszajelent. A Runner több módban működhet: lokálisan, Dockerben, távoli SSH-n, illetve autoscaling módban felhőkben. A gyakorlat azt mutatta, hogy a GitLab Runner kezdeti konfigurálása -különösen az autoscaling esetén- több manuális lépést igényelt, ellentétben a GitHub Actions alapértelmezett beállításai kedvezőbbek voltak a szakdolgozatban vizsgált rendszer felépítéséhez, így ott a kezdeti beüzemelés egyszerűbben ment végbe.

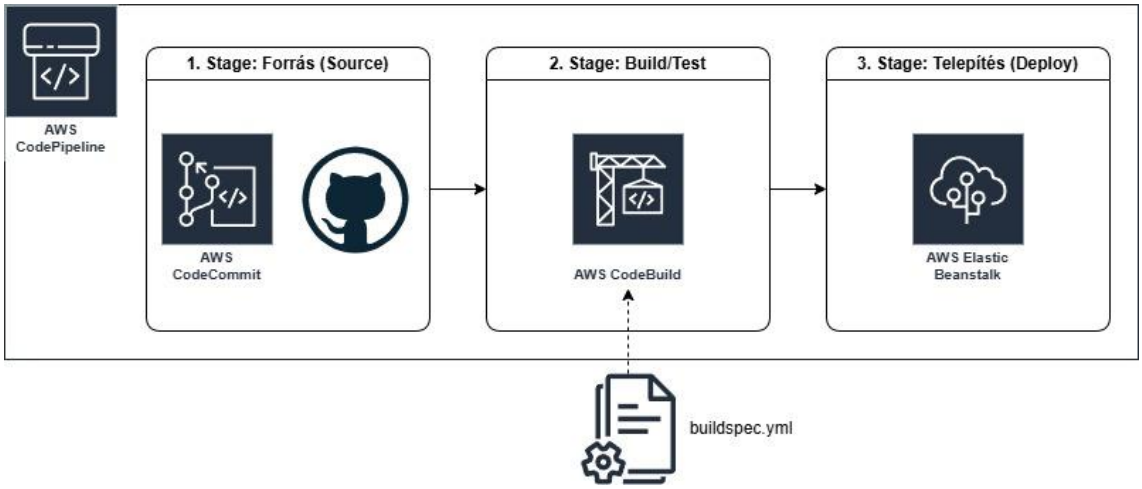
A szervezet dönthet GitLab-hostolt futtatókról (korlátozott beállítási lehetőségek) vagy saját Runner üzemeltetéséről. A Runner cache-t is biztosít a gyorsításhoz, és egyszerre több projektet szolgálhat ki. A GitLab előnye a beépített container registry, a monitoring és a Kubernetes-integráció. A YAML hasonlít a GitHub-szintaxisra, de a stages-nél explicit fáziselválasztást alkalmaz. A stages: és rules: blokkok egyértelműek, de nekem kicsit szigorúbbnak tűnt a szintaxis, mint a GitHub-é. Azonban hiába vonzó a modell, a projekt már a GitHubon élt. Egy teljes szervezeti migráció a GitLabre csak a CI/CD miatt nem lett volna indokolt, így bár technikailag kiváló alternatíva, a gyakorlati környezetben nem bizonyult jó döntésnek. [27]

4.4. AWS CodePipeline és CodeBuild

Az AWS menedzselt CI/CD szolgáltatásai elsősorban azoknak kedveznek, akik mélyen az AWS ökoszisztémában dolgoznak. A CodePipeline a folyamatos szállítás vizuális és automatizált kezelőfelülete: a forrás-build-teszt-deploy lépések stage-ekre és azokon belüli actionökre bonthatók, a kézi jóváhagyási pontok és a canary/blue-green

stratégiák is támogatottak. A technológiai összehasonlítás részeként a vizuális felület előny volt, de lassabbnak éreztem, mint a GitHub-os YAML-build - főleg, ha több stage-et frissítettem. [4]

A CodeBuild ehhez képest menedzselt CI motor: fordít, tesztel, és deploy-ra kész artefaktumokat állít elő. Automatikusan skáláz, párhuzamos futásokat kezel, nincs szükség saját build szerverre. A buildspec.yml-ben testre szabhatók a fázisok és parancsok; több nyelv és környezet támogatott, IAM-alapú jogosultságkezeléssel és használatarányos elszámolással. A buildspec.yml-t eleinte nehéz volt belőni, mert minden apró elírás build-hibát okozott - de később pont ez tette átláthatóvá. A CodePipeline, a CodeBuild és a buildspec.yml fájl közötti kapcsolatot a 11. ábra szemlélteti. [28]



11. ábra Az AWS CodePipeline és CodeBuild kapcsolata
(Forrás: Amazon Web Services [4] alapján a szerző által átdolgozva)

4.5. Eszközök összehasonlítása

4.5.1. Költség és üzemeltetési ráfordítás

A GitHub Actions közösségi repókban ingyenes; privát projektekben havi CI/CD percek állnak rendelkezésre, felette fizetős. A Jenkins licencdíj nélkül elérhető, de saját infrastruktúrát és üzemeltetést igényel. A GitLab CI-nél a GitLab.com hostolt futtatói kvótához kötöttek, a self-managed Runner üzemeltetése pedig hasonló kihívásokat hoz, mint a Jenkins. Az AWS CodePipeline/CodeBuild pay-as-you-go elszámolású: nincs saját szerver, a skálázás automatikus, a költség a használattal arányos. A dolgozatban

bemutatott megvalósításban a GitHub Actions ingyenes kerete elegendő volt, a futásidőt cache-eléssel és gyors tesztekkel kordában lehetett tartani.

A megfelelő CI/CD eszköz kiválasztásához több alternatívát is mérlegelni kellett. Felmerült a Jenkins, amely ugyan ingyenes, de saját szerver üzemeltetését igényli; valamint a GitLab CI, amely viszont a meglévő infrastruktúra áthelyezését tette volna szükségessé. Ezekre a kérdésekre egyértelmű igent egyik esetben sem tudtam adni, ezért végül a GitHub Actions mellett döntöttem. A választást elsősorban a gyors bevezethetőség, az alacsony adminisztrációs igény, valamint főleg az OIDC-alapú, biztonságos deploy lehetősége indokolta.

4.5.2. Rugalmasság, bővíthetőség, közösség és támogatás

A GitHub Actions erőssége a GitHub-integráció és a Marketplace actionök széles kínálata. A Jenkins hatalmas pluginrendszere extrém rugalmas, ugyanakkor a minőség és kompatibilitás vegyes - ez karbantartási költség. A GitLab CI „egyablakos” élményt ad: projektmenedzsment, kódtár, container registry és CI/CD egy helyen. Az AWS CodePipeline/CodeBuild mély AWS-integrációt nyújt; ez AWS-en ideális, más környezetekben viszont kevésbé rugalmas.

4.6. Választási szempontok

	GitHub Actions	**Jenkins**	**GitLab CI/CD**	**AWS**
Konfigu-rációs stílus	YAML (deklaratív)	Groovy (Jenkinsfile) vagy UI	YAML (deklaratív)	UI (konzol) vagy json/yaml
Hosting	Felhő (GitHub) vagy Saját	Saját (Self-hosted)	Felhő (GitLab) vagy Saját	Felhő (AWS Managed)
Bővíthetőség	Nagy (Market-place)	Nagy (Plugin-ek ezrei)	Közepes (CI Catalog, Templates)	AWS Szolgáltatás-integrációk
Futási teljesítmény	Párhuzamosí-tás (Matrix), Cache (Actions)	Párhuzamosí-tás (Agents), Cache (Pluginek)	Párhuzamosí-tás (Parallel), Cache (Jobs)	Párhuzamosí-tás (Actions), Cache (Build)
Biztonsági funkciók	Titkok (Secrets), OIDC, Jogosultságok	Credentials Plugin, Jogosultságok (Matrix)	Titkok (Variables), Szabályok (Rules)	IAM Szerepkörök, KMS (Titkosítás)
Árazás	Ingyenes percek (Free), Percalapú	Ingyenes Infrastruktúra költség	Prémium szintek	Használatalapú (Build percek)

2. táblázat CI/CD eszközök összehasonlítása

(Forrás: Saját elemzés és GitHub/AWS/Jenkins/GitLab dokumentáció alapján)

A vizsgált CI/CD eszközök fő tulajdonságai jelentősen eltérnek. A 2. táblázat ezeket a különbségeket foglalja össze.

4.6.1. Projektméret, csapatkompetenciák

Kis projektek és kezdő csapatok számára a GitHub Actions vagy a GitLab CI gyors bevezetést kínálnak, mivel nincs szükség saját szerver üzemeltetésére és a YAML-szintaxis könnyen megtanulható. Ezt tapasztaltam is: a GitHub Actions-szel egy nap alatt működő pipeline-t kaptam, míg a Jenkinsnél ugyanez legalább pár napot vett igénybe. Közepes méretű vállalatok, akik számos különféle projektet futtatnak, gyakran Jenkins mellett döntenek, mert a pluginrendszerrel bármilyen eszközt integrálhatnak, és rugalmasan skálázhatják a futtatókat. Nagyvállalati, AWS-orientált csapatoknál a CodePipeline/CodeBuild jelenti a legjobb választást, mert képes komplex több régiós telepítések, blue/green deployok menedzselésére és szoros az integráció az IAM-mel, CloudWatch platformmal és más AWS-szolgáltatásokkal. A döntést a csapat kompetenciái, a rendelkezésre álló infrastruktúra és a hosszan távon fenntartható költségstruktúra is befolyásolja.

Míg egy csapati környezetben a közös tudásbázis kulcsfontosságú, ennél a projektnél a technológiai választásaimat más szempontok vezérelték. A GitHubbal indultam, mert ismertem és logikusnak tűnt. GitHub Actions kézzel fogható döntésnek tűnik egy GitHub-on kezelt projecthez. Az infrastruktúra terén az AWS-re esett a választásom, kifejezetten a benne rejlő szakmai kihívás és a portfólióm bővítése céljából, miután az Azure és a Google Cloud platformokon már szereztem gyakorlatot.

4.6.2. Biztonsági és megfelelőségi elvárások

A GitHub Actions OIDC-integrációja nagy előny: nem kell statikus AWS-kulcsokat tárolni, a titkok kezelése központosítható [28]. Jenkinsben a hitelesítést a Credentials rendszer és pluginek oldják meg; rossz beállítás esetén a master gép túl széles hozzáférést kap. GitLab CI-nél a GitLab Secrets és a beépített szabályok védenek, de self-managed Runner esetén a pipeline hozzáfér a környezeti változókhoz. Az AWS CodePipeline/CodeBuild IAM-szerepkörökre támaszkodik - itt a legkisebb jogosultság elvének következetes betartása a döntő. A rendszer megfelelőségét elsősorban a régióválasztás, a titokkezelés és a részletes naplózás (audit trail) együttesen biztosítja.

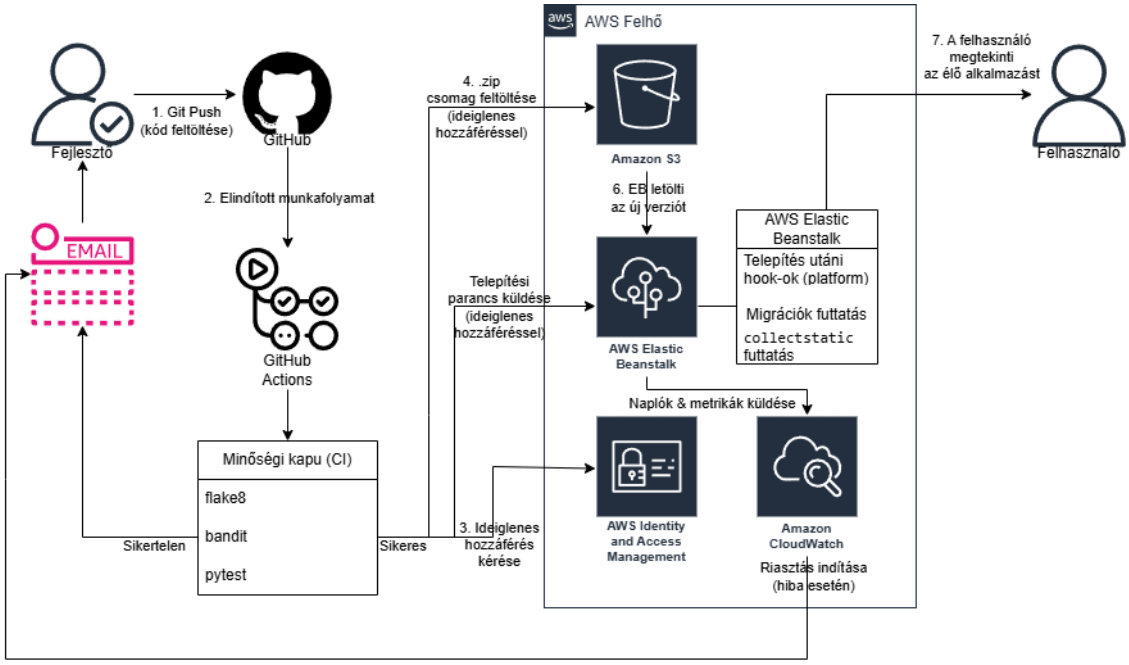
A szakdolgozatban bemutatott feladathoz a GitHub Actions bizonyult a legkézenfekvőbbnek: a kód GitHubon élt, a workflow-k deklaratív YAML-ban leírhatók,

a Marketplace bőséges, és az AWS-integráció (OIDC) megszabadított a statikus kulcsoktól. A Jenkins korábbi projektekben rugalmas volt, de a szerverüzemeltetés és a plugin-karbantartás többletmunkát kért. A GitLab CI és a CodePipeline felé is tettem próbát, de előbbi szervezeti migrációt igényelt volna, utóbbi pedig túl erősen kötődik az AWS-hez. A jelen projektben a GitHub Actions + Elastic Beanstalk páros vált be; a részleteket az 1. melléklet, illetve a GitHub repóm, kiemelve a főbb CI és CD workflowot a 2. mellékletbe és a következő fejezet dokumentálják.

5. Gyakorlati megvalósítás

5.1. Projektbemutatás és követelmények

A gyakorlati rész célja egy teljesen automatizált CI/CD-lánc kialakítása és értékelése egy Django-alapú webalkalmazáshoz. A megvalósított rendszer teljes architektúráját- a fejlesztői géptől az éles telepítésig és a monitorozási visszacsatolásig- a 12. ábra szemlélteti.



12. ábra A CI/CD pipeline teljes architektúrája (Forrás: saját rendszer architektúrája)

A demonstrációs alkalmazás egyszerű, böngészőből elérhető, és az AWS Elastic Beanstalk környezetben fut; a fókusz a pipeline működésén van, nem a frontend látványán. A projekt egy hello nevű appot tartalmaz a mysite Django-projekten belül, amely az alapértelmezett útvonalon a „Hello, CI/CD GitHubActions, AWS! All OK!” üzenetet jeleníti meg. Megtekinthető: http://mysite-env-2.eba-thgvigmp.eu-north-1.elasticbeanstalk.com/ weboldalon. A hangsúlyt egyértelműen a pipeline-ra tettem, mert számomra az jelentette a valódi kihívást. Maga az app csak egy 'hello world' szintű program, ami segített letesztelni, hogy a folyamat jól működik-e. A kódbázis moduláris, így később könnyen bővíthető - például felhasználókezeléssel vagy adatbázis-funkciókkal.

5.1.1. A webalkalmazás fő funkciói

A mysite projekt hello alkalmazása egyetlen nézetet (index) szolgál ki a gyökérútvonalon (Locust terheléses teszt miatt egy CPU-intenzív ággal lett bővítve); az összerendelést a mysite/urls.py végzi, a kiszolgálás HTTP 200 státuszkódot ad vissza. A fejlesztéshez Django 4.2 LTS-t használtam (stabil API és hosszú távú támogatás). Bár a 3.12 és 3.13 kisebb teljesítményjavulást hoztak, de a 3.11 jelenleg is ipari standardnak számít, ezért maradtam ennél. Kihasználja az újabb nyelvi fejlesztéseket (többek között az aszinkron futtatás fejlődéseit). Az architektúra a szokásos Django mintára épít: a logika nézetekre és - a bővítésnél - modellekre bontható, így az egyes appok egymástól elkülönítve fejleszthetők.

5.1.2. Üzleti és nem funkcionális követelmények

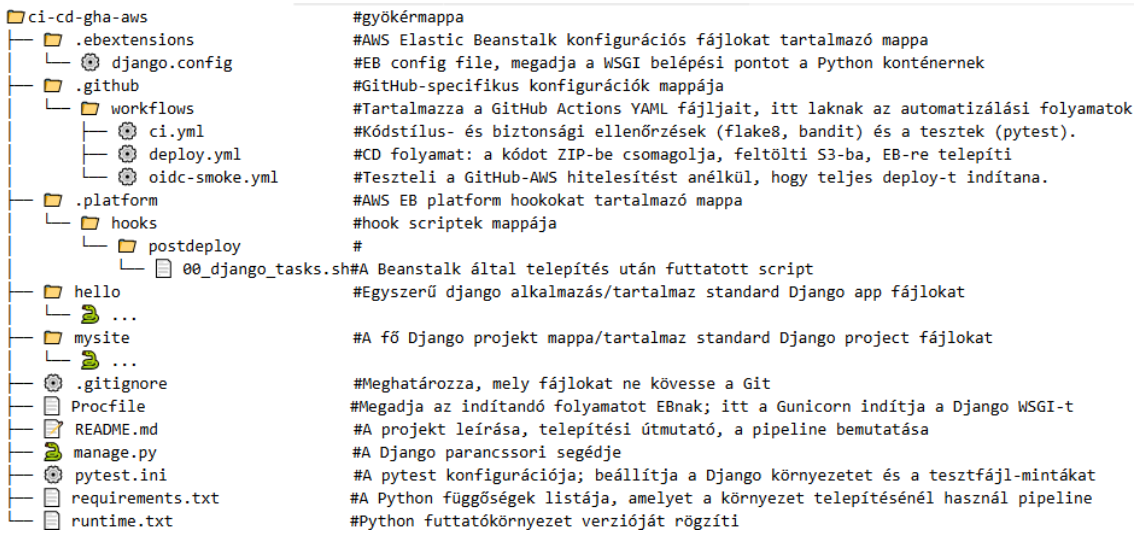
Elsődleges üzleti elvárás a gyors és megbízható kiadás: push után perceken belül fusson le a build és a teszt, majd siker esetén történjen meg az automatikus deploy. Nem funkcionális oldalon a magas rendelkezésre állás és a skálázhatóság volt fontos, ezért esett a választás az Elastic Beanstalkra, amely a terheléselosztást és az autoscalinget menedzseli, ami rengeteg fejlesztési időt spórolt meg. A biztonságos hitelesítéshez - a legkisebb jogosultság elvét követve - OIDC-alapú IAM-integrációt alkalmaztam, amely a legkisebb jogosultság elvét követi, így a pipeline nem tart statikus kulcsokat. Fejlesztői élmény szempontból a YAML-alapú GitHub Actions-konfiguráció és a Beanstalk „menedzselt” modellje csökkenti a belépési küszöböt. Mivel egy személyes csapatként, egyedül fejlesztettem az egész pipeline-t és nem akartam elveszni a részletekben.

5.2. Fejlesztői környezet és kódbázis felépítése

5.2.1. Repository struktúra, virtuális környezet

A forráskód a szelese/ci-cd-gha-aws GitHub-repóban található. A gyökérben helyezkedik el a mysite projekt, a hello alkalmazás és a .github/workflows mappa a pipeline-definíciókkal. A projekt kódbázisának szerkezetét a 13. ábra szemlélteti. Helyben külön Python-virtuális környezetet hoztam létre (python -m venv .venv), hogy a csomagok izoláltan kezelhetők legyenek- ezt a .venv-et mindig helyben használtam, hogy ne keveredjenek össze a csomagok. A venv külön kezelése attól is megkímél, hogy csak a saját gépünkön működik a beállítás típusú hibától. A telepített csomagok: Django, flake8, bandit, pytest, boto3, gunicorn. A repo szerkezete áttekinthető: a kód mellett megtalálható a requirements.txt, .gitignore, runtime.txt és Procfile. A .gitignore kiterjeszti

a Python/Django alapértelmezést, kizárja a virtuális környezetet, cache fájlokat és a deploy ZIP-et. Ha ezt nem tesszük meg, a deploy ZIP idővel felesleges fájlokkal (szeméttel) telne meg, ami hosszú távon karbantarthatatlanná tenné a rendszerfrissítéseket.



13. ábra A projekt könyvtárstruktúrája és a fájlok szerepe (Forrás: saját szerkesztés)

5.2.2. Követelményfájlok és konfigurációk

A requirements.txt a kulcs - nélküle kaotikus lenne a telepítés. Rögzíti a függőségeket, ezzel biztosítva a reprodukálhatóságot CI-ben és Beanstalkon. A runtime.txt az EB Python-verzióját adja meg (python-3.11), összhangban a ci.yml 3.11-es beállításaival. (2. melléklet ci.yml)

A mysite/settings.py-ben az INSTALLED_APPS kiegészült a hello appal, az ALLOWED_HOSTS pedig felvette az EB-domaint. A SECRET_KEY és a DEBUG nincs a repóban: a GitHub Secrets és az EB környezeti változói tárolják. Az .ebextensions/django.config beállítja a WSGI-útvonalat (mysite/wsgi.py) és a statikus fájlok gyűjtésének lépéseit. A Procfile tartalma: web: gunicorn mysite.wsgi:application --bind 127.0.0.1:8000 - ez a platform által elvárt futtatási definíció.

5.3. CI pipeline tervezése és megvalósítása

5.3.1. Build lépések, tesztelés és statikus kódelemzés

A CI-t a ci.yml workflow valósítja meg. Push és pull_request eseményre indul a main ágon, ubuntu-latest runneren. A lépések: repository checkout; Python környezet

beállítása (actions/setup-python@v5); függőségek telepítése; minőségbiztosítási ellenőrzések (flake8 a szintaktikai és stílusproblémákhoz; bandit a tipikus Python-sebezhetőségek azonosításához; pytest az egységtesztekhez). A flake8 és bandit lépéseket utólag adtam hozzá - nélkülük tele lett volna hibával a kód.

A végén egy verzió-ellenőrzés lefut, hogy a Python/Django verziók a vártak legyenek. Bármely lépés hibája megállítja a pipeline-t - helyes, mert így a problémák korán láthatók. Persze ennek a plusz biztonságnak és alaposságnak megvan az ára: míg korábban a build idők nagyjából 20-30 másodpercesek voltak, a CI 'quality gate'-ek bevezetése után ez felugrott kb. 15-20 másodperccel, tehát közel a duplájára.

5.3.2. Artefaktumkezelés és cache-stratégia

A CI jelenleg nem készít bináris artefaktumot - a Django forrásból épül. A deploy-folyamat (deploy.yml) viszont létrehoz egy app.zip csomagot, amely felkerül S3-ba. A buildidő további csökkentéséhez érdemes pip-cache-t használni (actions/cache), és - ha a projekt nő - a teszteredményeket is cache-elni. Ha nő a projekt, cache nélkül lassú lesz - tudom, korábbi tapasztalataim alapján. A verziózás a deploy.yml-ben a dátum és a commit hash alapján generált címkékkel történik, így minden kiadás visszakövethető. A file részletesen megtalálható a 2. mellékletben deploy.yml valamint GitHub repóban.

5.4. CD pipeline és telepítés AWS Elastic Beanstalk-ra

5.4.1. Beanstalk környezet létrehozása és konfigurálása

Az Elastic Beanstalk a választott platformnak megfelelően automatikusan létrehozza az EC2-példányokat, a load balancert és az autoscaling csoportot. Az első melléklet szerint először az eu-north-1 régióban hoztam létre az alkalmazást és a környezetet. A Python 3.11 platformmal single instance-szel indultam (gyorsan validálható a deploy út), később horizontális skálázásra váltottam. A környezeti változók között beállítottam a DJANGO_SETTINGS_MODULE-t, a SECRET_KEY-t és a DEBUG=False értéket. A domain/SSL-hez alapértelmezett AWS-tanúsítványt használtam. A Security Group kizárólag a 80-as és 443-as portot engedi. A lépések konzolról és AWS CLI-vel egyaránt végrehajthatók, így a folyamat teljes mértékben automatizálható és reprodukálható.

5.4.2. Deploy script és workflow (migrációk, statikus fájlok)

A deploy.yml ubuntu-latest környezeten fut; a Python beállítása és a csomagtelepítés után az aws-actions/configure-aws-credentials@v4 OIDC-vel végzi a hitelesítést. A csomagolás a projekt gyökérből készít ZIP-et (a .git, .github és a virtuális környezet nélkül). A workflow dátum+commit hash alapján verziócímkét számol, feltölti a ZIP-et S3-ba, létrehozza az új EB-verziót, majd frissíti a környezetet. Egy várakozó ciklus 10 másodpercenként ellenőrzi az állapotot, és csak Ready + Green/Ok egészségnél fejez be (többször timeoutolt a security beállításnál, de eventsből kiderült a hiba). A migráció és a statikus fájlok gyűjtése a Beanstalk post-deploy hookjain fut (manage.py migrate, collectstatic); igény esetén ez külön lépésként a workflowba is áthelyezhető.

5.4.3. Rollback és roll forward mechanizmusok

Az EB verziókezelése miatt hibás deploy esetén az előző kiadás egy paranccsal visszaállítható (konzolról vagy CLI-vel), ez a rollback a 3. melléklet 2. részében szerepel részletesebben. A blue/green stratégia külön környezeten teszteli az új verziót, és DNS-szinten vált át - így a visszagörgetés gyors és kockázatszegény. Éles hiba esetén a DNS-szintű váltás (Blue/Green) szinte azonnalira (pár másodpercre) rövidíti a down-time-ot, szemben a Beanstalk 3,5 perces automatikus visszaállítási idejével. A jelenlegi megoldás single instance-re épül; roll-forward esetén új build készül a javítással, és megy a frissítés. A jövőben érdemes és izgalmas lesz a többkörnyezetes (blue/green) modellt automatizálni - akár CodeDeploy integrációval.

5.5. Hitelesítés és jogosultságkezelés OIDC-vel

5.5.1. IAM szolgáltató és role definíciók

Az AWS-ben létrehoztam egy OIDC-identitásszolgáltatót a https://token.actions.githubusercontent.com URL-lel és sts.amazonaws.com audienciával, majd egy GitHubOIDC-EBDeploy nevű IAM-szerepkört. A trust policy korlátozza, hogy kizárólag a szelese/ci-cd-gha-aws repo main ága vehesse át a szerepkört (így a forkok nem férnek hozzá). Az inline policy az EB-műveletekhez és az S3 feltöltéshez szükséges jogokat tartalmaz és egy minimalizált iam:PassRole engedélyt.

A gyakorlati beállítás során a trust policy helyes konfigurálása kulcsfontosságúnak bizonyult. A bevezető konfigurációban az aud (audience) mező nem volt megfelelően beállítva, ami - a várakozásoknak megfelelően - azonnal AccessDenied hibát eredményezett. Ez a 'szerencsés' hiba valójában remek éles tesztnek bizonyult: a logfájlok

elemzése után egyértelművé vált, hogy az OIDC-alapú biztonsági korlátozás (a forkok kizárása) valóban hatékonyan működik. [25]

5.5.2. Token kezelés és titokmenedzsment (GitHub Secrets)

Tokenkezelés és titokmenedzsment (GitHub Secrets) OIDC használatával nincs szükség hosszú életű AWS-kulcsokra. A ci.yml és deploy.yml permissions blokkja id-token: write jogot ad a runnernek, így ideiglenes JWT-t kérhet. A nem érzékeny változók (pl. APP_NAME, ENV_NAME, AWS_REGION) a workflowokban, a titkok (Django SECRET_KEY, admin jelszó) a GitHub Secretsben vannak. Futtatáskor a titkok ${{ secrets.NAME }} alakban érhetők el. Az EB-ben a Software-szekció alatt szintén környezeti változókat állítottam be, így nincs szükség külön .env-re, ami tovább csökkenti az érzékeny adatok kiszivárgásának kockázatát.

5.6. Infrastruktúra mint kód és automatizáció

5.6.1. .ebextensions és konfigurációs fájlok

Az Elastic Beanstalk finomhangolása .ebextensions fájlokkal történik. A projektben a django.config állítja a WSGI-útvonalat (aws:elasticbeanstalk:container:python: WSGIPath=mysite/wsgi.py) és engedélyezi a statikus fájlok kiszolgálását. Ugyanitt adhatók meg skálázási paraméterek, log-gyűjtési szabályok és egyéb infrastruktúra-összetevők. A HealthCheckPath átállítása után azonnal eltűntek a ‘Degraded’ riasztások.

5.6.2. Alternatívaként Terraform/CloudFormation

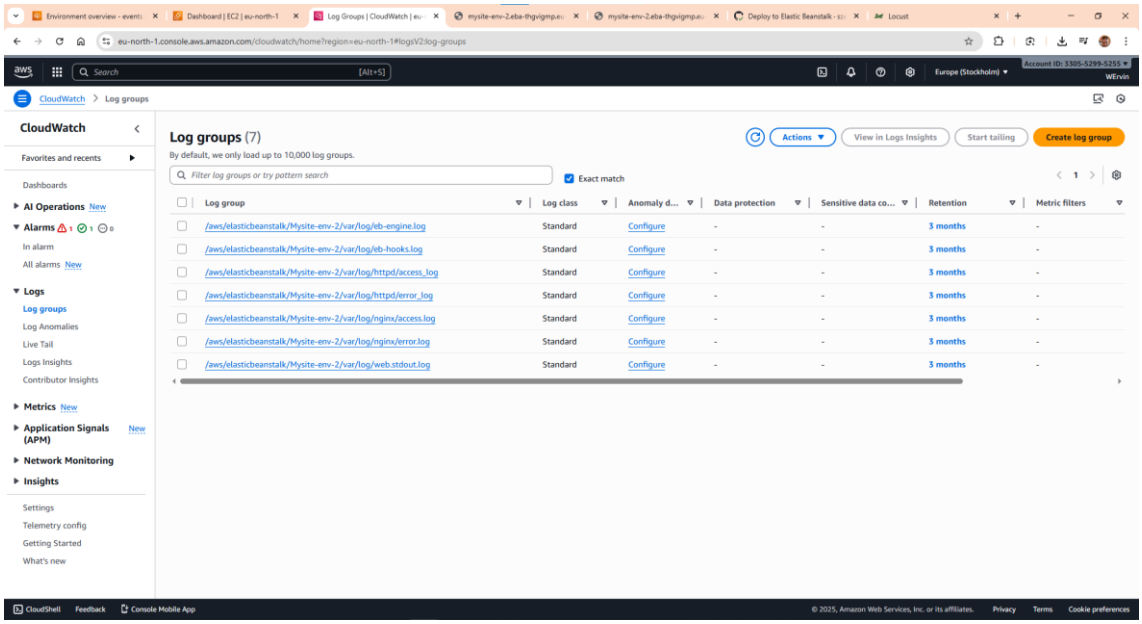
Noha a Beanstalk sok mindent menedzsel, hosszabb távon érdemes Infrastructure as Code (IaC) megközelítést (Terraform, CloudFormation) használni az EB-környezet, az IAM-szerepkörök, az S3-bucket és a kapcsolódó erőforrások deklaratív leírására. Az IaC javítja a reprodukálhatóságot és a verziókövetést; modulokkal és változókkal rugalmasabb is. Trade-off: kezdeti többletmunkát és eszközismeretet igényel. Bevezetéskor plusz 1-2 nap többlet, cserébe nyerünk stabil újraépíthetőséget.

5.7. Monitoring, naplózás és hibakeresés

5.7.1. AWS CloudWatch és log streamek

A Beanstalk alapértelmezetten integrálódik a CloudWatch-csal: az EC2 és az alkalmazás logjai (például beanstalkd.log, web.stdout.log, web.stderr.log) a konzolon visszakereshetők. A deploy utáni várakozási ciklusban használt AWS Elastic Beanstalk

describe-events részletes eseménylistát ad, így a figyelmeztetések és hibák azonnal látszanak. Figyelmeztetéseket kaphatunk: HTTP végpont, Slack, SMS és e-mail formájában. A feliratkozott metrikák (CPU, hálózat, memória) alapján autoscaling-szabályok definiálhatók. CloudWatch mindent gyűjt - logokat, metrikákat, ahogyan az a 14. ábrán látható. [29]



14. ábra A Cloudwatch környezet 'Log groups' felülete (Forrás: saját szerkesztés)

5.7.2. Health checkek, riasztások és teljesítménymérés

15. ábra A GitHub Actions pipeline várakozása az Elastic Beanstalk „Green/Ok” állapotára (Forrás: saját szerkesztés)

Az EB a bejövő HTTP-válaszok alapján határozza meg a környezet állapotát (Green/Yellow/Red). A deploy.yml addig vár, amíg Ready és Green/Ok nem lesz az állapot, ahogyan a 15. ábrán is látszik Enélkül a pipeline 'sikeresnek' jelezné a futást, miközben az app lehet, hogy el sem indult. Időtúllépésnél lekéri a legutóbbi eseményeket a gyors diagnózishoz. A CloudWatch Alarms e-mailben jelez CPU-tüskét vagy 5xx hibákat. A terheléses tesztet (Locust) a projektben ténylegesen elvégeztem - a 3. mellékletben bemutatott eredmények szerint a rendszer automatikusan új példányokat indított a CPU-terhelés hatására, majd a terhelés megszűntével önállóan visszaállt. A kísérlet igazolta az infrastruktúra horizontális skálázhatóságát.

5.8. Találkozott problémák és megoldási stratégiák

5.8.1. Kompatibilitási és konfigurációs hibák

A fejlesztés során több hibát is tapasztaltam, de tanultam belőle. Minden egyes hiba megoldása egy lépést vezet a jó megoldás felé. Az első telepítés után az EB egészségjelentése Degraded állapotot mutatott, mivel a root útvonal 404-et adott vissza, így a Health Check hibának értékelte. Ezt úgy oldottam meg, hogy a Django hello/views.py-ben módosítottam az index nézetet, hogy gyökérszinten is 200-as választ adjon. Ezenkívül gunicorn hiányzott a környezetből, ami 502 Bad Gateway hibát okozott; a requirements.txt-be felvettem a gunicorn csomagot, és létrehoztam a Procfile-t. A WSGI útvonalat a .ebextensions/django.config fájlban kellett pontosan megadni, különben az EB nem találta az alkalmazást.

5.8.2. Skálázási és stabilitási kihívások

Az egy példányos környezet nem tolerálta jól a rövid idejű erőforrás-csúcsokat; a CPU-terhelés 100% körülire emelkedett már rövid ideig tartó tesztelés során is. Az autoscalingre váltva két példányt állítottam be minimumnak, a maximális példányszámot pedig négyre. A CloudWatch metrikák alapján 60% CPU-nál új példány indult. Egy másik probléma a minimális IAM jogosultság megadása volt: az első IAM policy túl szűk volt, így az aws elasticbeanstalk create-application-version parancs jogosultság hibával megszakadt. A dokumentáció alapján kiterjesztettem a policy-t az elasticbeanstalk:* és s3:* műveletekre a deploy bucketre, majd visszavontam az admin jogot. Ezek a hibák tanulságként szolgáltak a legkisebb jogosultság elvének megvalósításában. A fejlesztés során az alábbi 3. táblázat foglalja össze a legfontosabb hibákat, jelenségeiket és megoldási módjaikat.

	Jelenség (Hiba)	Megoldás és Eredmény
EB Health Check Hiba	Az Elastic Beanstalk környezet Degraded állapotot jelzett.	A gyökér (/) útvonal 404-es hibát adott. A Django urls.py és views.py módosításával a gyökérútvonal 200 OK választ kapott, így a Health Check sikeres lett.
Alkalmazás Indítási Hiba	502 Bad Gateway hiba telepítés után.	A logok elemzése kimutatta, hogy hiányzott a gunicorn WSGI szerver. A requirements.txt kiegészítése és egy Procfile létrehozása megoldotta a problémát.
Biztonsági Riasztás (CI)	A bandit eszköz riasztást jelzett a kódban hagyott SECRET_KEY miatt.	A SECRET_KEY eltávolítása a kódból és biztonságos áthelyezése GitHub Secrets-be és az EB környezeti változói közé.
Biztonsági Riasztás (OIDC)	IAM trust policy hibás	feltétel javítása repo-azonosítóra, mindig ellenőrizni kell az ARN-t
Telepítési Hiba (IAM)	A deploy.yml pipeline AccessDenied hibával elszállt.	A kezdeti "lusta" admin jogok helyett iteratív hibakeresés kezdődött (1. melléklet 78-82. lépés. A pipeline logjai alapján egyesével lettek hozzáadva a minimálisan szükséges jogok, megvalósítva a "Least Privilege" elvet

3. táblázat Kiemelt problémák és megoldási stratégiák összefoglalása

(Forrás: saját szerkesztés)

5.9. Felhasználói dokumentáció és verziókezelés

A dokumentáció két helyen él: a repozitórium README.md-je a rövid, angol nyelvű telepítési összefoglalóval és pipeline-használattal, illetve az 1. melléklet, amely lépésről lépésre végigvezet a gyakorlati megvalósításon. A verziókezelést a Git/GitHub adta; minden mérföldkő (CI integráció, EB deploy, OIDC beállítás stb.) külön commitot kapott. A kiadások GitHub-tagekkel is jelölhetők a könnyebb visszakövetéshez. A telepített alkalmazás elérhető az EB-URL-en:

http://mysite-env-2.eba-thgvigmp.eu-north-1.elasticbeanstalk.com/

A fenti megoldás közben sokat finomodott a pipeline: a YAML-ok modularizálása és a minőségbiztosítási lépések (lint, security scan, tesztek) mérhetően javították a kódminőséget. Az Elastic Beanstalk elsőre „fekete doboznak” tűnt, de az .ebextensions és a Procfile révén pontos kontrollt kaptam. A legnagyobb kihívást az IAM/OIDC finomhangolása volt; több iteráció után állt össze a minimális, mégis működő jogosultságkészlet. Összességében a projekt igazolta, hogy egyszemélyes csapatként is felépíthető professzionális, megbízható CI/CD-infrastruktúra nyílt forrású és menedzselt eszközökre támaszkodva.

6. Elemzés és értékelés

6.1. CI/CD teljesítménymutatók

A szoftver-szállítási folyamat hatékonyságát a szakirodalom négy ún. DORA-mutatóval szokta jellemezni: két mutató a sebességet (az átvezetési időt és a kiadások gyakoriságát), kettő a stabilitást (a hibás kiadások arányát és a helyreállítás idejét) írja le [30].

A lényeg, hogy a jó csapatok nem sebesség-megbízhatóság dilemmaként tekintenek ezekre, hanem mindkettőt egyszerre javítják. Kezdetben általános feltételezés volt, hogy a gyorsaság és a megbízhatóság egymás rovására megy - a DORA-mutatók azonban rávilágítottak, hogy ezek akár együtt is javíthatók.

DORA-mutatók	**Projektben mért érték**	**Értékelés (a szöveg alapján)**
Átvezetési idő (Lead Time)	~4,5 perc (CI + Deploy)	a teljesen automatizált pipeline miatt a változások perceken belül éles környezetbe kerülnek.
Telepítési gyakoriság (Deployment Frequency)	naponta többször	a rendszer naponta többször képes éles deployra, emberi beavatkozás nélkül.
Hibás kiadások aránya (CFR)	(nem volt hibás kiadás, tehát) <5%	statikus és biztonsági szkennerek, valamint a tesztelési lépések már korán kiszűrik a hibákat.
Helyreállítási idő (MTTR)	~3,5 perc (automatikus rollback)	AWS Elastic Beanstalk azonnal felismeri és visszaállítja a hibás verziókat

4. táblázat A projekt DORA-mutatóinak értékelése (Forrás: saját szerkesztés)

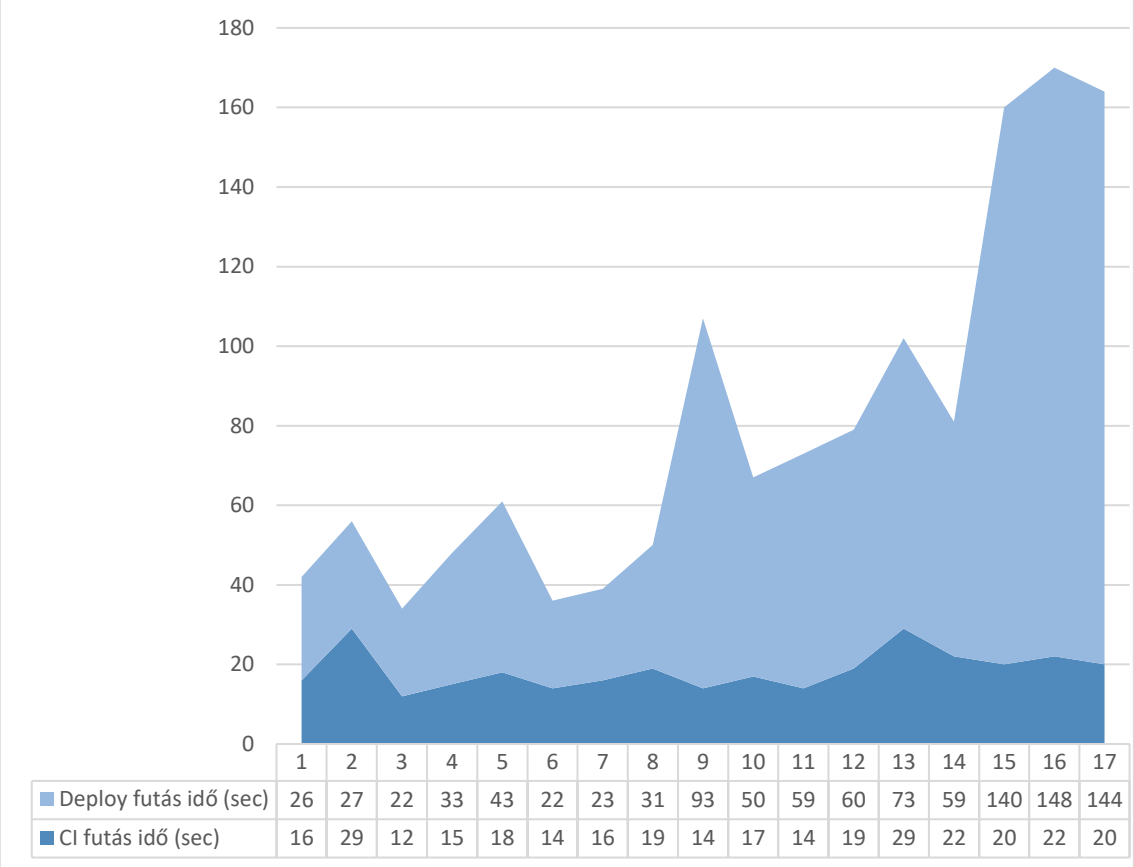
A dolgozathoz készített referencia-implementációban a CI-láncot a GitHub Actions futtatja: egy Python 3.11 környezet indul, települnek a függőségek, majd lefut a flake8, a Bandit és a pytest. Ez a hármas rövid, de átfogó ellenőrzést ad: a stílus- és biztonsági hibákat már a fejlesztés elején kiszűri, a tesztek pedig visszaigazolják a funkcionalitást. A CI futások ideje stabilan 14-29 másodperc között mozog. A commitokat automatikusan építi és teszteli a rendszer, így a változtatások átfutási ideje maximum 5 perc, ami egy manuális folyamathoz képest óriási gyorsulás. A beépített statikus és biztonsági szkennerek miatt a problémák többsége még azelőtt kiderül, hogy bekerülne a fő ágba. A DORA-mutatók rendszeres figyelése azt is megmutatta, hol érdemes tovább finomítani a folyamatot, ahogy a 4. táblázat is mutatja. Fontos megjegyezni, hogy projektünk a mért értékek és DORA-mutatók alapján messze az iparági átlag és az "Elit" kategória felett

teljesít, a legfrissebb DORA benchmark tesztek szerint is a maximális 10-es pontszámot érve el, ezt és a pipeline helyreállítási idejét a 3. melléklet 2. részében demonstrálom. [31]

6.1.1. Build idők, tesztek lefutási ideje, sikerességi arány

A ci.yml workflow futási ideje átlagosan harminc másodperc alatt van. A pip-függőségek telepítése után lefut a flake8, a Bandit és a pytest; a lintelés és a biztonsági vizsgálat együttesen tíz-tizenöt másodpercet vesz igénybe. A sikerességi arány gyakorlatilag 100 %. A hibák többségét még a feature-ágakon elcsíptük, így a main ágra alig került hibás commit. Így a change lead time néhány perces tartományban marad, ami a DORA „Elite” szintjéhez hasonlítható. Persze nagyobb, monolit rendszereknél ez már más nagyságrend — ezért mindig a projekt méretéhez viszonyítok.

6.1.2. Deployment idők és elérhetőségi mutatók



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Deploy futás idő (sec)	26	27	22	33	43	22	23	31	93	50	59	60	73	59	140	148	144
CI futás idő (sec)	16	29	12	15	18	14	16	19	14	17	14	19	29	22	20	22	20

16. ábra CI és Deploy futásidők (Forrás: saját szerkesztés)

A Deploy futások időtartama a mérések szerint 26-148 másodperc között alakult. A korai futások 20-60 s tartományban mozogtak, a minőségi kapu (quality gate) és a post-deploy hook (migrate + collectstatic) bevezetése után pedig 1-2,5 percre nőtt a teljes idő

(az utolsó három futás 140+ s). Ez nem teljesítményromlás, hanem tudatos késleltetés: a Deploy csak sikeres CI után indul, majd a Beanstalk Ready + Green/Ok állapotáig és a hook-lépésekig vár. A pipeline futási idejét részletesen a 3. mellékletben szereplő mérési táblázat mutatja be, amely a GitHub Actions workflow-futásokból származó időadatokat foglalja össze és a 16. ábrán öntjük az adatokat grafikonos formába.

A Beanstalk egészségi jelzői („Ready” és „Green/Ok”) jelzik, mikor áll újra szolgálatra kész állapotba. A helyreállítási idő így jellemzően csak néhány perc; a Mean Time to Detect (MTTD) is rövid, mert a hibák pillanatok alatt megjelennek a naplókban. A kialakított infrastruktúra működése alapján megállapítható, amíg a Beanstalk új verziója „zöldre” vált - a csomagolás és verzióképzés automatizálása viszont jelentősen lerövidítette a kiadás teljes idejét.

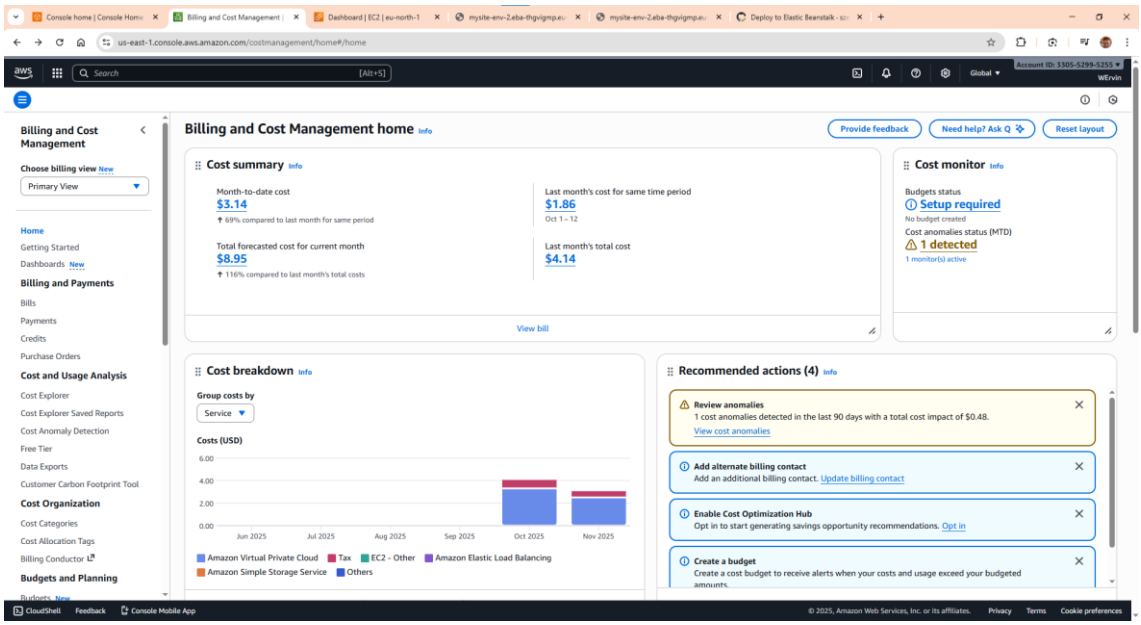
A pipeline futtatása során világossá vált, hogy a folyamatos integráció és bevezetés nem csak technológiai, hanem kulturális kérdés is. A GitHub Actions által kínált statikus kódelemzés és biztonsági vizsgálatok segítettek abban, hogy a hibákat korán észrevegyük, így a deploy-ok ritkán fulladtak kudarcba. A telepítési időket jelentősen csökkentette az automatizált zippelés, verzióképzés és környezetfrissítés; a legnagyobb várakozást a Beanstalk környezet „zöldre” váltása okozta. Ebből levonhatjuk a tanulságot: automatizálás nélkül sokkal lassabb lenne az egész, ezért is törekedtem arra, hogy a deploy ágat elsődleges prioritásként megvalósítsam. De ez a biztonság rovására ment és utólag kellett beállítanom a legkisebb hozzáférést mindenhova. Megtanultam, hogy egy éles rendszer esetén a biztonságra kell a legnagyobb hangsúlyt fektetni.

6.2. Költség- és erőforrás elemzés

6.2.1. AWS szolgáltatások és pipeline futtatások költségei

Az AWS árazása használatarányos: pay-as-you-go modellben azt fizetjük ki, amit ténylegesen igénybe veszünk. Fejlesztési fázisban ez különösen kedvező: a GitHub Actions havi ingyenes keretét (2000 perc) nem merítettük ki, a Beanstalk kis példánnyal az AWS Free Tier része volt, az S3 és a CloudWatch logok pedig filléres tételt jelentettek. Ez a 'Free Tier' modell tökéletesen működött a projekt kezdeti, 'single instance'-re épülő fázisában. A 17-es ábrán látható valós költségek (havi kb. 8-9 dollár) akkor jelentek meg, amikor a rendszert a 5.7-es fejezetben leírt terheléses teszthez horizontálisan skálázhatóvá alakítottam. Ez a magas rendelkezésre állású, 'production-ready' konfiguráció egy Elastic Load Balancer bevezetését igényelte, ami már egy alacsony, de valós 'pay-as-you-go'

díjjal jár. Ez a költség tehát nem hiba, hanem a megbízhatóság és a skálázhatóság konkrét, pénzben mért ára. Ráadásul a környezetet leállítottam, szünetben, amikor nem fejlesztettem, így csak a valódi használati idő után számláztak. Tapasztalatom szerint ez a legjobb spórolás. [12]



17. ábra Az AWS Billing and Cost Management konzol költségáttekintése (Forrás: saját szerkesztés)

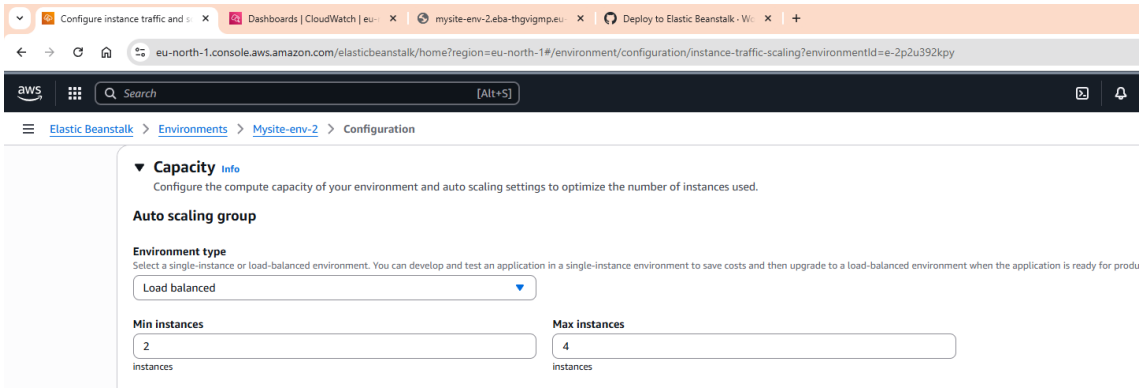
6.2.2. Optimális erőforrás-használat és költségcsökkentési lehetőségek

Az optimalizálás kulcsa a megfelelő példányméret és autoscaling beállítás: a terhelés növekedésével új példányok indulnak, csökkenéskor leállnak. Hosszabb távon a Reserved Instances vagy a Savings Plans akár 75 %-os kedvezményt adhatnak a listaárhoz képest. A CI-nél a pip cache és a párhuzamos lépések rövidítik a futásidőt, így kevesebb runner-percet használunk el. Fejlesztési fázisban a Free Tier kihasználása jelentős megtakarítást hozott, de a jövőben a reserved instance-típusú elköteleződés és egy jól beállított cache-stratégia további költségcsökkentést jelenthet. A rendszer fejlesztése és üzemeltetése közben világossá vált, hogy a várakozó ciklus miatt néha hosszabb ideig futottak az instanciák, mint kellett volna; ezen igyekeztem finomítani. Ezért a várakozó ciklus idejét jelentősen lerövidítettem az alapbeállítás: 300 másodpercéről 5 másodpercre. Ezért nem maradnak feleslegesen aktív instanciák csak pár másodpercig aktívak. Ez jelentős megtakarítást jelenthet hosszú távon.

6.3. Teljesítmény, skálázhatóság és megbízhatóság

6.3.1. Terhelési tesztek és horizontális skálázás

A terhelési tesztekkel azt lehet modellezni, hogy az alkalmazás hogyan reagál a várható felhasználói forgalomra. A Beanstalk Autoscaling csoportjai (lásd a 18. ábrát) rugalmasan indítanak új példányokat, ha a CPU-terhelés magas, és állítják le őket, ha visszaesik a terhelés. A horizontális skálázás (X-tengelyű skálázás) előnye, hogy nem egy nagy gépet vásárolunk, hanem több kisebbet futtatunk párhuzamosan, így elkerülhető az egyedüli hibapont (Single Point of Failure) és növelhető a rendelkezésre állás. The Art of Scalability című könyv szerint ez a megközelítés a hibatűrés és a skálázhatóság kulcsa, mivel a terhelést több, redundáns komponens között osztja el A mérések során 100-200 párhuzamos kérés mellett is stabil maradhat a válaszidő, amit a CloudWatch és a Beanstalk metrikái mutatták, hogy hol kell/lehet még finomítani. Konténeres alternatívák? ECS, Fargate, Kubernetes további rugalmasságot nyújtanak: a Fargate automatikusan kiosztja az erőforrást, a Kubernetes pedig hordozhatóbb, de komplexebb megoldást ad. [32]

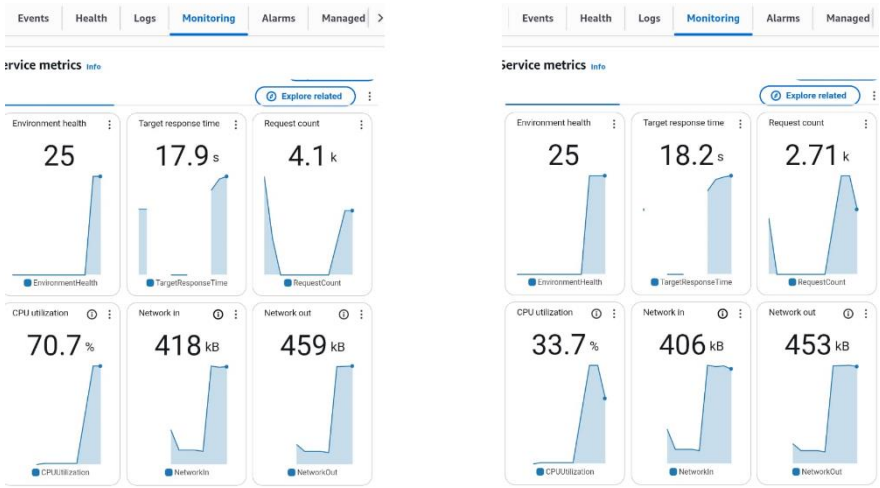


18. ábra Az Elastic Beanstalk skálázási és magas rendelkezésre állási beállításai (Forrás: saját szerkesztés)

6.3.2. High availability és fault tolerance vizsgálata

A magas rendelkezésre állást úgy érhetjük el, hogy a szolgáltatást több Availability Zone-ben futtatjuk. Availability Zone alapbiztosítás - ha az egyik kiesik, a másik viszi tovább. Fault tolerance esetén az adatok replikálódnak, és automatikus failover biztosítja, hogy a rendszer komponens kiesés esetén is működjön. Több régióra is lehet terjeszkedni (multi-region), ahol a Route 53 ellenőrzi az egészséget és irányítja a forgalmat;

dolgozatfeladatában egy régió, több AZ-os üzemmód futott, ami jó kompromisszumnak bizonyult. Multi-region konfigurációval még nagyobb rendelkezésre állást lehetne elérni, de ennek plusz költség- és komplexitásigénye van, ezért úgy döntöttem: egy régió elég lesz. [33]



19. ábra Az Elastic Beanstalk Monitoring felülete a terheléses teszt alatt (Forrás: saját szerkesztés)

Az előző pontban említett terhelési tesztek során kiderült, hogy az egyszerű "Hello World" alkalmazás minimális számítási igénye nem generált elegendő CPU-terhelést a horizontális skálázás (autoscaling) teszteléséhez. A probléma megoldására és az infrastruktúra valós tesztelésére ezért egy dedikált, /cpu-test/:

http://mysite-env-2.eba-thgvigmp.eu-north-1.elasticbeanstalk.com/hello/cpu-test/

nevű végpontot implementáltam a Django alkalmazásba. Ez a végpont egy szándékosan processzorigényes feladatot szimulált, így lehetővé téve a kontrollált stressztesztet és az autoscaling-mechanizmus valós idejű igazolását. A 19. ábra pontosan ennek a sikeres tesztnek az eredményét mutatja.

A nézet mesterségesen leterheli a szerver CPU-erőforrásait. Locusttal stressztesztet hajtottam végre ezen az erőforrás igényes végponton. A hirtelen megugró terhelés hatására az Elastic Beanstalk két további t3.micro példányt indított el a meglévők mellé. Ezzel a futó példányok száma ideiglenesen 2-ről 4-re nőtt. Természetesen a terhelési csúcs elején a szerver válaszideje megnőtt, és rövid ideig kis hibaarány is tapasztalható volt, de amint az új instance-ok forgalomba álltak, a rendszer teljesítménye

stabilizálódott. A teszt lefutása után az auto-scaling mechanizmus automatikusan leállította a két extra példányt, visszaállítva a környezetet az eredeti kapacitásra. Ez a kísérlet igazolta, hogy a kialakított infrastruktúra képes a terhelés növekedésére automatikus horizontális skálázódással reagálni, a terhelés megszűntével pedig önmaga is visszaáll az alapállapotába (önhelyreállító módon), ezzel biztosítva a magas rendelkezésre állást.

6.4. Biztonsági értékelés

6.4.1. IAM szerepkörök és OIDC beállításainak felülvizsgálata

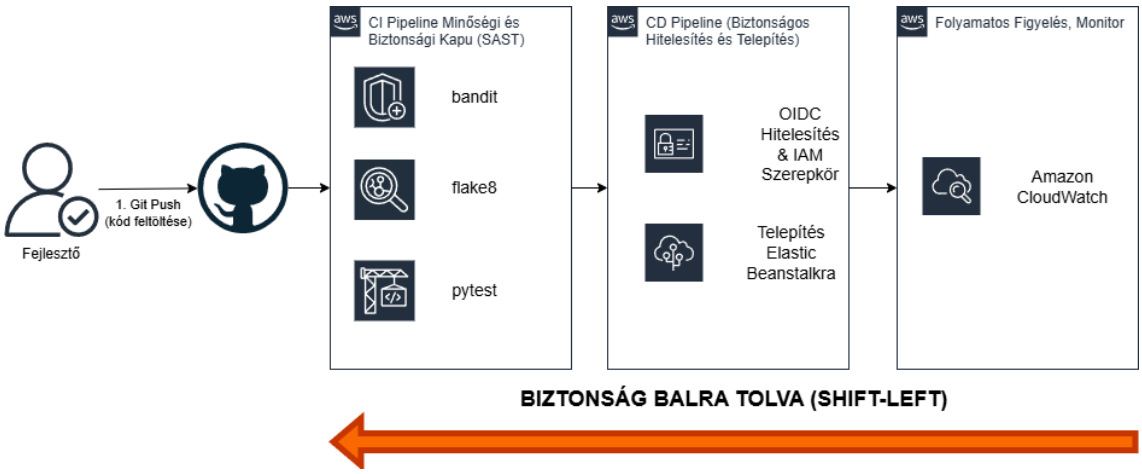
A GitHub Actions és az AWS közötti OIDC-hitelesítés rövid élettartamú tokenekkel oldja meg a hitelesítést, így nincs szükség statikus AWS-kulcsokra a repóban. A kialakított IAM-szerepkör csak az Elastic Beanstalk verziók létrehozásához, az S3-műveletekhez és a környezet frissítéséhez kapott engedélyt - ezzel teljesült a legkisebb jogosultság elve. Ezt nem elsőre találtam el — az első verzióm azonnal hibát dobott, ami jól mutatja, milyen érzékeny a jogosultság-beállítás. Az AWS a fizikai és virtualizációs réteget védi, a mi felelősségünk pedig az S3 vödör (bucket) és az IAM-szabályok konfigurálása a feladat. A GitHub Secrets és az EB-környezeti változók megfelelő használatával a titkok nem kerülnek a forráskódba. Gyakorlati tanulságként: a szerepkör első policyja túl szűk volt, a CloudTrail logok alapján pontosítottam, így sikerült minimalista, mégis működő engedélykészletet beállítani. Ez volt az a pont, ahol először éreztem, hogy tényleg értem, mit csinál az IAM a háttérben. [25] [34]

6.4.2. Sebezhetőségi vizsgálatok és compliance

A biztonságos fejlesztéshez több szinten futnak sebezhetőségi vizsgálatok. A statikus tesztek (SAST), mint a Bandit, a forráskódot ellenőrzik, a Flake8 a kódminőségi hibákat azonosítja, míg a futó alkalmazás terhelését és stabilitását a Locust tesztekkel vizsgáltuk. Ezek a vizsgálatok már a kód commitjától kezdve végigkísérik a pipeline-t; a „shift left” elv szerint minél hamarabb felfedezzük a hibát, annál olcsóbb javítani. Shift left, nézzük a 20. ábrát. Korán fogjuk el a hibát - olcsóbb, gyorsabb.

A compliance (PCI DSS v4.0, ISO/IEC 27001) megköveteli az automatizált biztonsági teszteket, így a bandit és a flake8 nemcsak a minőséget, hanem a megfelelőséget is szolgálja. Ezek a szkennerek néha meghosszabbítják a build időt -

különösen, ha sok a dependency, de cserébe még fejlesztés közben kiszűrik a kritikus hibákat. [35]



20. ábra A "Shift-Left" biztonsági modell a projekt gyakorlatában
(Forrás: saját szerkesztés)

6.5. Összehasonlítás alternatív megoldásokkal

6.5.1. Hagyományos telepítés vs. automatizált pipeline

A kézi *build-test-deploy* folyamatok idejétmúlt, hibalehetőségekkel teli módszerek. Minden egyes kézi lépés emberi hibát, felesleges időt és erőforrás-pazarlást rejt magában. Az automatizálás épp ezt a láncot töri meg: a rendszer önállóan készíti elő, teszteli és telepíti a kódot. A csapatnak már nem a gombnyomás a dolga, hanem az, hogy értéket teremtsen. [2]

Megéri? A különbség nemcsak kényelmi kérdés, hanem gazdasági is. Számoljunk: egy kézi telepítés átlagosan 35 percet igényel, és ha naponta mindössze ötször fordul elő, az máris 725 munkaóra évente - vagyis nagyjából 7,25 millió forint fejlesztői munkaidő. Ezzel szemben - ahogy azt a 6.2.1-es fejezet és a 17-es ábra részletesen bemutatta - a 'production-ready', horizontálisan skálázott rendszer (Elastic Load Balancerrel) valós, mért éves díja is mindössze kb. 30 000 Ft (kb. havi 9 dollár) volt. Ez a kézi telepítés költségének kevesebb mint 0,6 százaléka! Még ha az AWS fizetős (nem Free Tier) környezetét nézzük is, az eredmény egyértelmű: az automatizálás évente ~ 7,2 millió Ft (~ $21,600) megtakarítást hoz, miközben a hibák száma gyakorlatilag nullára csökken.

A konklúzió tehát világos: a pipeline egyszeri kiépítése bőven megtérül, hiszen már az első hónapokban visszahozza az árát. A fejlesztők ideje felszabadul, hibák aránya jelentősen csökken, a kiadások pedig kiszámíthatóbbá válnak és a csapat végre arra koncentrálhat, ami igazán számít: az innovációra.

Ha AWS-be kell egyébként is deployolni, akkor az ár növekedés (ha van egyáltalán) minimális. A valódi összehasonlítás még ennél is drámaibb, hiszen az automatizált megoldás az emberi hibák kiküszöbölésével nemcsak anyagi, hanem minőségi előnyt is biztosít.

	Kézi telepítés	**Automatizált telepítés**
Telepítés idő	30-40 perc	<5 perc
Hiba kockázat CFR	Magas	Elenyésző <5%
Rollback	Manuális	Automatikus
Monitoring	Kézi	CloudWatch + Logs
Biztonság	Jelszavas kulcsok	OIDC tokenek
Első beállítás ráfordítás	szinte nincs	közepes
Becsült megtérülés	-	15-20 kézi telepítés

5. táblázat Hagyományos és automatizált telepítés összehasonlítása

(Forrás: saját szerkesztés)

6.5.2. Más CI/CD eszközök gyakorlati tapasztalatai

Jenkins: elképesztően rugalmas, moduláris pluginrendszerrel, de üzemeltetni és karbantartani kell. Tapasztalatból mondom, hogy egy nagyobb projectnél a Jenkinsnél egy plugin-frissítés konkrétan fél napba telik. GitLab CI/CD: egy platformon kapjuk a kódtárolást, a konténer-registryt és a CI/CD-t, de a self-managed Runner saját erőforrást igényel. AWS CodePipeline/CodeBuild: mélyen integrálódik az AWS-szolgáltatásokkal, a pipeline-szerkesztés viszont vizuális vagy JSON-alapú, ami más jellegű tanulást kér. Ebben a projektben a GitHub Actions bizonyult a legjobb kompromisszumnak: a forráskód amúgy is (az általam használt) GitHubon van, a YAML szintaxis deklaratív, a Marketplace actionök bőségesek, és a workflow-k újrafelhasználhatók. A futtatók efemerek, tehát minden job után eltűnnek, ez egyrészt biztonságosabb, másrészt költséghatékony. [23]

6.6. Korlátok és kockázatok

6.6.1. Vendor lock-in és szolgáltatófüggőség

A vendor lock-in azt jelenti, hogy egyetlen szolgáltatóhoz kötődünk - a váltás pedig időben és pénzben is drága lehet. Ezért sok cég több felhőt használ egyszerre: 86 %

legalább két szolgáltatót tart fent, hogy jobb alkupozícióban legyen és csökkentse a kockázatot [36]. A multi-cloud azonban új bonyodalmakat hozhat (eltérő API-k, adatmozgási költségek, ritka szakértelem). Ha például most GCP-re kellene átvinnem a rendszert, a legtöbb időt az IAM-szabályok újraírása vinné el. A konténerizáció (Docker, Kubernetes) és a vendor-semleges IaC-megoldások (Terraform) javíthatják a hordozhatóságot, ez persze elsőre jó alternatíva lehet, de a gyakorlatban már két felhő között is nehéz az összehangolás. A pipeline-unk jelenleg erősen AWS-re és GitHubra épül; hatékony így, de hosszabb távon célszerű vendor-független elemeket bevezetni. [32]

6.6.2. Jogszabályi és adatvédelmi megfontolások

A GDPR és más adatvédelmi szabályok előírják az átlátható adatkezelést, az adatlakhely tiszteletben tartását és a személyes adatok védelmét. Multi-cloud környezetben különösen figyelni kell, mely régiókban tároljuk az adatokat, és hogy egységes jogosultsági szabályokat alkalmazzunk. A compliance keretrendszerek (PCI DSS, ISO/IEC 27001) az automatikus sebezhetőség-vizsgálatokon túl a hozzáférések naplózását és a titokkezelést is megkövetelik. A pipeline-ban a GitHub Secrets és az IAM-szerepkörök megfelelő használatával el tudtam érni, hogy az érzékeny információk ne kerüljenek napvilágra, miközben a CloudWatch logok biztosították az auditálhatóságot. A jogi megfelelés nem egyszeri feladat: folyamatosan figyelni kell a szabályok változásait, rendszeresen felülvizsgálni a hozzáféréseket és frissíteni a biztonsági eszközöket. [37]

A GDPR-kompatibilitásra eleinte legyintettem, amíg egy tesztadat véletlenül logba nem került - ezután vettem komolyabban. Ezért is kellett az első Mysite-env-1 -et kitörölnöm és indítottam egy új Mysite-env-2 verziót.

6.7. Felhasználói visszajelzések és tanulságok

Mivel a projektben a fejlesztői, tesztelői és DevOps-mérnöki feladatokat is egy személyben láttam el, a "felhasználói visszajelzések" valójában a közvetlen fejlesztői tapasztalatot jelentették. Ebből a szempontból a tapasztalatok pozitívak: a kiadási ciklusok gyorsak, a működés pedig megbízható volt. A mysite-env-2 környezet szünet nélkül szolgálta ki a kéréseket, az üdvözlő oldal pedig egyértelmű, azonnali jelet adott arról, hogy a deployment sikeres volt.

Ugyanakkor fejlesztőként és tesztelőként is észleltem, hogy a frissítés közbeni várakozás néha hosszúnak tűnt - ezen a jövőben a blue/green vagy a canary deploy

stratégia segíthet. A pipeline-ba épített monitoring kulcsfontosságúnak bizonyult a gyors beavatkozáshoz; a logok elemzése során több apró konfigurációs hibát is hamar megtaláltam és javítottam.

Összefoglalva: a projekt igazolta, hogy még egy egyetlen fejlesztő is képes stabil CI/CD-infrastruktúrát építeni, ha a folyamatot tudatosan tervezi, a minőségbiztosítást beépíti, és a fejlesztői élményt (ami ebben az esetben egyben a felhasználói élményt is jelentette) szem előtt tartja. A legfontosabb tanulság számomra az volt, hogy rájöjjek: a technikai hatékonyság és a jó fejlesztői környezet együtt nő - gyors iterációval és fókuszált fejlesztéssel a szolgáltatás is jobb lesz.

7. Összegzés és jövőbeli lehetőségek

7.1. Eredmények összefoglalása

Az elmúlt hónapokban sikerült egy teljesen automatizált CI/CD-folyamatot összeállítanom a Django-alkalmazáshoz. Az volt a célom, hogy ne csak működjön, hanem valóban megbízható legyen! A GitHub Actions és az AWS Elastic Beanstalk együttese ma már lehetővé teszi, hogy a kódváltozásokat gyorsan és biztonságosan juttassam ki az éles környezetbe. A CI-lépés (a minőségi kapukkal együtt) stabilan fél perc alatt lefut, a 'wait-for-green' logikával ellátott CD-lépés pedig (a 16. ábra alapján) átlagosan 2-2,5 percet vesz igénybe. Így a teljes átfutási idő (Lead Time) stabilan 5 perc alatt marad, miközben a környezet állapotát végig monitorozom. A pipeline része a statikus kód- és sebezhetőségi vizsgálat, így a legtöbb hiba már a tesztfázisban napvilágra kerül. Nagy előrelépés volt az OIDC-alapú hitelesítés bevezetése is: ez rövid élettartamú, dinamikus tokenekkel oldja meg a GitHub és az AWS közötti kommunikációt. A fejlesztési és telepítési lépések deklaratív YAML-workflowokban szerepelnek, ami átláthatóvá, reprodukálhatóvá és újra felhasználhatóvá teszi a folyamatot. A teljes forráskód és a lépésről lépésre haladó telepítési útmutató a GitHubon érhető el; ezek bizonyítják, hogy akár egy fejlesztő is képes lehet professzionális CI/CD-infrastruktúrát építeni. A projekt ezzel túlnőtte a gyakorlati beadandó szerepét: most már valóban működő, napi szinten használható rendszer lett belőle.

7.2. Fő tanulságok és következtetések

A projekt során egy dolog vált teljesen világossá számomra: a DevOps nem csak technológia, hanem gondolkodásmód. A fejlesztők és az üzemeltetés folyamatos együttműködése, az automatizálás, a gyors visszacsatolás és a nyitott kommunikáció teremti meg a hatékonyságot. Még egy egyszemélyes fejlesztésben is muszáj fejlesztőként és „üzemeltetőként” is gondolkodnom - ez volt a legnagyobb szemléletváltás. A GitHub Actions rugalmas szintaxisa és a marketplace-en elérhető actionök lehetővé tették, hogy a tényleges fejlesztésre összpontosítsak, miközben a pipeline megbízhatóan működött a háttérben. Az Elastic Beanstalk PaaS-ként elrejtette az infrastruktúra komplexitását: automatikusan kezelte az EC2-példányokat, a terheléselosztást és a skálázást. Így a telepítés annyi volt, mint egy új workflow-lépés. Ugyanakkor látszik az árnyoldal is: a magas szintű kényelem erősebb kötődést jelent a szolgáltatóhoz, és bizonyos beállítások finomhangolása nehézkesebb, mint saját kezelési

réteggel. Az OIDC-integráció megmutatta, hogy a hosszú élettartamú kulcsok mellőzése nagyban csökkenti a biztonsági kockázatokat. Az OIDC-beállítás megértése volt az első pillanat, amikor igazán „összeállt a kép” a biztonság és automatizáció kapcsolatáról.

7.3. Jövőbeli fejlesztési javaslatok

7.3.1. Konténerizáció (Docker, Kubernetes) integrálása

Ha ma újrakezdeném a projektet, már biztosan konténerekben gondolkodnék. A Docker-képek biztosítják, hogy az alkalmazás minden környezetben ugyanúgy viselkedjen; a Kubernetes pedig lehetővé teszi a konténerek ütemezését, menedzselését és automatikus skálázását. Ezzel sokkal könnyebb lenne skálázni és több szolgáltató között mozogni. A mikroszolgáltatás-alapú architektúrákat, csökkentené a vendor lock-int, és megnyitná az utat a multi-cloud vagy hibrid deployok felé. Ilyen architektúra fejlesztéshez illeszkedő Kubernetes klaszterek a vendor lock-in csökkentésében is segítenek: bármelyik nagy felhőszolgáltatóval integrálhatók [38].

7.3.2. Serverless deploy lehetőségek (AWS Lambda, Fargate)

A serverless megoldásokkal már régóta szemezek - lenyűgöz, hogy mennyire eltűnik mögülük az infrastruktúra. Az AWS Lambda eseményvezérelt környezet, ahol nem kell a szerverekkel bajlódni; a Fargate pedig a konténerek serverless futtatását valósítja meg. Ezek a megoldások ideálisak lehetnek rövid futású, nagy fluktuációjú feladatokhoz, és teljesen leválasztanak az infrastruktúra kezelésétől. A hidegindítás persze még mindig bosszantó korlát, de kíváncsi vagyok, hová fejlődik ez a technológia pár éven belül. [39].

7.3.3. Mélyebb automatikus tesztelés (integration, e2e, security)

A tesztelés az egyik olyan terület, ahol érzem, hogy még rengeteget tudnék fejlődni. A jelenlegi pipeline elsősorban egységtesztekre épül. A következő lépés az integrációs és end-to-end tesztek, amelyek a komponensek közötti együttműködést vizsgálják. A biztonsági vizsgálatokat is érdemes elmélyíteni, a jelenlegi statikus (SAST) vizsgálatokat kiegészítve dinamikus (DAST) és függőségvizsgálati (SCA) eszközökkel. Nagyon érdekel, mit tud majd egy AI, ha megtanulja felismerni, mely tesztek feleslegesek - ez a jövő egyik legizgalmasabb kihívása lehet [40].

7.4. Kutatási irányok és iparági trendek

7.4.1. AI-alapú pipeline optimalizáció

A mesterséges intelligencia már most is beszivárog a DevOps-folyamatokba - például a CodeGuru javaslatai néha megdöbbentően pontosak. Az OWASP irányelvei az automatizált biztonsági tesztelést támogatják [40]. Ezzel párhuzamosan az AI-alapú prediktív elemzések segíthetnek előre jelezni a rendszerhibákat és teljesítmény-szűk keresztmetszeteket, míg az olyan AI-hátterű eszközök, mint az Amazon CodeGuru és a GitHub Copilot, már most is képesek támogatni a kódminőség javítását és a fejlesztési idő csökkentését [41]. Azonban a vállalatok mindössze egy kis része érzi magát felkészültnek az AI integrálására: egy friss felmérés szerint a cégek 84 %-a látja az AI jelentőségét, de csak 14 % tekinti magát igazán felkészültnek az alkalmazására. Ez rávilágít arra, hogy a jövőbeni kutatásnak nemcsak technológiai, hanem szervezeti és oktatási aspektusokat is figyelembe kell vennie.

Azt látom, hogy a technológia már itt van, de az emberek (és a szervezetek) még tanulják, hogyan használják jól.

7.4.2. Multi-cloud és hibrid CI/CD

A vendor lock-in enyhítésére egyre többen választják a multi-cloud stratégiát: több felhőszolgáltató párhuzamos használatát konténerizációval és platformfüggetlen IaC-eszközökkel (Terraform, Pulumi). Az olyan eszközök, mint a Spinnaker vagy az Argo CD, lehetővé teszik a többfelhős deployok összehangolását. Ezzel egyidejűleg növekszik a komplexitás, ezért józan mérlegelés szükséges, hogy egy adott projektnek valóban megéri-e. [42]

7.5. Személyes reflexiók

Az egész projekt során egyre mélyebben merültem el a DevOps kultúrában és a felhőalapú technológiákban. A tanulás nemcsak technikai szinten zajlott: mivel a projektet teljes egészében egyedül valósítottam meg, a csapatmunka helyett a fegyelmezett, módszeres munkavégzésre és az alapos dokumentálásra kellett támaszkodnom. A folyamatos visszajelzést nem kollégáktól, hanem magától az automatizált pipeline-tól és a tesztektől kaptam, ami nélkülözhetetlennek bizonyult a sikerhez. Különösen inspirált, hogy a GitHub repom és a hozzá kapcsolódó workflow-k alapján más fejlesztők is tanulhatnak a megoldásaimból.

Tudatosult bennem, hogy ez a munka nem csupán a szakdolgozat céljait szolgálja, hanem egy komoly portfólió-projekt alapja is, amely jól demonstrálja a képességeimet egy jövőbeli állásinterjú során. Ennek szellemében a jövőben szeretném bővíteni a projektet konténerizált és serverless irányokba, kipróbálni a Kubernetes adta hordozhatóságot, és kísérletezni AI alapú optimalizációval. Emellett nagy vonzerőt jelent számomra a multicloud stratégiák rugalmassága, ezért terveim között szerepel, hogy a pipeline-t Azure App Service vagy Google App Engine környezetben is kipróbáljam.

A gyakorlati tapasztalatok mellé a formális tudás megszerzése is célom. A munkaerőpiaci értékem növelése érdekében tervezem letenni az AWS Associate szintű minősítéseket. Mivel ez a szakdolgozat egy klasszikus DevOps projekt volt, a megszerzett tudást két irányban is relevánsnak érzem. A jövőben célom az AWS Certified Developer - Associate (DVA) megszerzése, valamint az AWS Certified SysOps Administrator - Associate (SOA) letétele.

Összességében a projekt megerősítette, hogy az örömteli, alkotómunka és a tanulás kéz a kézben jár: a technológia iránti lelkesedésem tovább nőtt, és készen állok arra, hogy újabb szintre emeljem az alkalmazásomat.

Irodalomjegyzék

[1] Amazon, „What is DevOps?,” 2025. [Online]. Available: https://aws.amazon.com/devops/what-is-devops/. [Hozzáférés dátuma: 25 10 2025].

[2] Red Hat, „What is CI/CD?,” Red Hat, 2024. [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd. [Hozzáférés dátuma: 25 10 2025].

[3] Django, „Django documentation,” 2025. [Online]. Available: https://docs.djangoproject.com/. [Hozzáférés dátuma: 25 10 2025].

[4] Amazon, „CodePipeline concepts,” AWS Documentation, 2025. [Online]. Available: https://docs.aws.amazon.com/codepipeline/latest/userguide/concepts.html. [Hozzáférés dátuma: 25 10 2025].

[5] ISO/IEC/IEEE, „ISO/IEC/IEEE 12207:2017 - Systems and software engineering — Software life cycle processes,” 2017. [Online]. Available: https://www.iso.org/standard/63712.html. [Hozzáférés dátuma: 25 10 2025].

[6] IBM, „What is the software development life cycle (SDLC)?,” 2025. [Online]. Available: https://www.ibm.com/think/topics/sdlc. [Hozzáférés dátuma: 25 10 2025].

[7] C. R. China és M. Goodwin, „What is continuous integration?,” IBM, 2023. [Online]. Available: https://www.ibm.com/think/topics/continuous-integration. [Hozzáférés dátuma: 25 10 2025].

[8] C. R. China, „Three pillars of observability: Logs, metrics and traces,” IBM, 2023. [Online]. Available: https://www.ibm.com/think/insights/observability-pillars. [Hozzáférés dátuma: 25 10 2025].

[9] P. Mell és T. Grance, „The NIST Definition of Cloud Computing,” National Institute of Standards and Technology, 2011. [Online]. Available:

https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf. [Hozzáférés dátuma: 25 10 2025].

[10] C. R. China és M. Gooodwin, „IaaS, PaaS, SaaS: What's the difference?,” IBM, 2024. [Online]. Available: https://www.ibm.com/think/topics/iaas-paas-saas. [Hozzáférés dátuma: 25 10 2025].

[11] IBM, „What is FaaS?,” 2025. [Online]. Available: https://www.ibm.com/think/topics/faas. [Hozzáférés dátuma: 25 10 2025].

[12] C. Slingerland, „Choosing the best AWS pricing model: a complete look at AWS pricing,” CloudZero, 2024. [Online]. Available: https://www.cloudzero.com/blog/aws-pricing-model/. [Hozzáférés dátuma: 25 10 2025].

[13] Serverless, „AWS Lambda The Ultimate Guide,” 2024. [Online]. Available: https://www.serverless.com/aws-lambda. [Hozzáférés dátuma: 25 10 2025].

[14] AWS, „AWS Elastic Beanstalk Documentation,” Amazon Docs, 2025. [Online]. Available: https://docs.aws.amazon.com/elastic-beanstalk/. [Hozzáférés dátuma: 25 10 2025].

[15] P. Oroszvári, AWS könyv, https://cheppers.hu/aws-konyv-magyarul, ISBN: 978-615-02-3441-0: Cheppers Global Zrt., 2025.

[16] Amazon, „Savings Plans,” 2025. [Online]. Available: https://aws.amazon.com/savingsplans/. [Hozzáférés dátuma: 25 10 2025].

[17] V. Krishnakumar, „Everything You Need to Know About AWS Elastic Beanstalk,” CloudOptimo, 2025. [Online]. Available: https://www.cloudoptimo.com/blog/everything-you-need-to-know-about-aws-elastic-beanstalk/. [Hozzáférés dátuma: 25 10 2025].

[18] AWS Whitepaper, „Blue/Green Deployments on AWS,” 2025. [Online]. Available: https://docs.aws.amazon.com/pdfs/whitepapers/latest/blue-green-deployments/blue-green-deployments.pdf. [Hozzáférés dátuma: 25 10 2025].

[19] Microsoft, „App Service overview,” 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/app-service/overview. [Hozzáférés dátuma: 25 10 2025].

[20] Google, „App Engine,” 2025. [Online]. Available: https://cloud.google.com/appengine. [Hozzáférés dátuma: 25 10 2025].

[21] S. Nayak, „ECS, EKS, and Fargate: A Deep Dive into Container Orchestration,” CloudOptimo, 2024. [Online]. Available: https://www.cloudoptimo.com/blog/ecs-eks-and-fargate-a-deep-dive-into-container-orchestration/. [Hozzáférés dátuma: 25 10 2025].

[22] GitHub, „Understanding GitHub Actions,” GitHub Docs, 2025. [Online]. Available: https://docs.github.com/en/actions/get-started/understand-github-actions. [Hozzáférés dátuma: 25 10 2025].

[23] GitHub, „Using GitHub-hosted runners,” GitHub Docs, 2025. [Online]. Available: https://docs.github.com/en/actions/how-tos/manage-runners/github-hosted-runners/use-github-hosted-runners. [Hozzáférés dátuma: 25 10 2025].

[24] GitHub, „Reusing workflow configurations,” GitHub Docs, 2025. [Online]. Available: https://docs.github.com/en/actions/concepts/workflows-and-actions/reusing-workflow-configurations. [Hozzáférés dátuma: 25 10 2025].

[25] GitHub, „Configuring OpenID Connect in Amazon Web Services,” GitHub Docs, 2025. [Online]. Available: https://docs.github.com/en/actions/how-tos/secure-your-work/security-harden-deployments/oidc-in-aws. [Hozzáférés dátuma: 25 10 2025].

[26] Jenkins, „Architecting for Scale – Jenkins,” Jenkins Documentation, 2024. [Online]. Available: https://www.jenkins.io/doc/book/scaling/architecting-for-scale/. [Hozzáférés dátuma: 25 10 2025].

[27] GitLab, „GitLab Runner,” 2025. [Online]. Available: https://docs.gitlab.com/runner/. [Hozzáférés dátuma: 25 10 2025].

[28] Codefresh, „AWS CodeBuild: The Basics and a Quick Tutorial,” Codefresh, 2025. [Online]. Available: https://codefresh.io/learn/devops-tools/aws-codebuild-the-basics-and-a-quick-tutorial/. [Hozzáférés dátuma: 25 10 2025].

[29] AWS, „Amazon CloudWatch tutorials,” Amazon Docs, 2025. [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-tutorials.html. [Hozzáférés dátuma: 25 10 2025].

[30] N. Harvey, „DORA’s software delivery metrics: the four keys,” DORA, 2025. [Online]. Available: https://dora.dev/guides/dora-metrics-four-keys/. [Hozzáférés dátuma: 25 10 2025].

[31] N. Forsgren, J. Humble és G. Kim, Accelerate: The Science of Lean Software and Devops - Building and Scaling High Performing Technology Organizations, IT Revolution Press, 2018.

[32] M. L. Abbott és M. T. Fisher, The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Sebastopol: O'Reilly Media, 2015.

[33] AWS, „Fault tolerance and fault isolation,” Amazon Docs, 2023. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/availability-and-beyond-improving-resilience/fault-tolerance-and-fault-isolation.html. [Hozzáférés dátuma: 25 10 2025].

[34] AWS, „Security best practices in IAM,” Amazon Docs, 2025. [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html. [Hozzáférés dátuma: 25 10 2025].

[35] B. K. Kaithe, „Shift Left Security: A Paradigm Shift in Software Development Security Integration,” *European Journal of Computer Science and Information Technology,* 13. kötet, 24. szám, pp. 96-102, 2025.

[36] Flexera, „2025 State of the Cloud Report,” 2025. [Online]. Available: https://info.flexera.com/CM-REPORT-State-of-the-Cloud. [Hozzáférés dátuma: 25 10 2025].

[37] European Data Protection Board, „Guidelines 05/2021 on the Interplay between the application of Article 3 and the provisions on international transfers as per Chapter V of the GDPR,” 2023. [Online]. Available: https://www.edpb.europa.eu/our-work-tools/our-documents/guidelines/guidelines-052021-interplay-between-application-article-3_en. [Hozzáférés dátuma: 25 10 2025].

[38] IBM, „Top 7 benefits of Kubernetes,” 2025. [Online]. Available: https://www.ibm.com/think/insights/kubernetes-benefits. [Hozzáférés dátuma: 25 10 2025].

[39] C. Bicknell, „An overview of Amazon EC2 vs. AWS Lambda,” TechTarget, 2025. [Online]. Available: https://www.techtarget.com/searchcloudcomputing/tip/An-overview-of-Amazon-EC2-vs-AWS-Lambda. [Hozzáférés dátuma: 25 10 2025].

[40] OWASP Foundation, „DevSecOps Guideline v0.2: Software Composition Analysis (SCA) / SAST / DAST,” 2023. [Online]. Available: https://owasp.org/www-project-devsecops-guideline/. [Hozzáférés dátuma: 25 10 2025].

[41] GitHub, „GitHub Copilot Documentation,” 2025. [Online]. Available: https://docs.github.com/en/copilot. [Hozzáférés dátuma: 25 10 2025].

[42] Canadian Forum for Digital Infrastructure Resilience (CFDIR), „Multi-Cloud Adoption,” 2024. [Online]. Available: https://ised-isde.canada.ca/site/spectrum-management-telecommunications/sites/default/files/documents/CFDIR-Multi-Cloud-Adoption.pdf. [Hozzáférés dátuma: 25 10 2025].

[43] J. Cohen, „Understanding the shared responsibility model in cloud security,” Upwind, 2024. [Online]. Available: https://www.upwind.io/glossary/what-is-the-shared-responsibility-model. [Hozzáférés dátuma: 25 10 2025].

Ábrajegyzék

Táblázatjegyzék

Mellékletek jegyzéke

1. melléklet:

Gyakorlati lépések: a rendszer reprodukálása

Projekt teljes forráskódja elérhető a GitHubon: https://github.com/szelese/ci-cd-gha-aws

Ez a melléklet lépésről lépésre vezeti végig az olvasót a rendszer teljes reprodukálhatóságának biztosítása érdekében. A megoldás a GitHub Actions automatizálási lehetőségeire, valamint az AWS Elastic Beanstalk szolgáltatására épül. A fejlesztés során a Python 3.11 és a Django 4.2 LTS verzió került felhasználásra, a biztonságos hitelesítést pedig az AWS IAM OpenID Connect (OIDC) integráció biztosítja. A projekt célja egy olyan működő prototípus létrehozása volt, amely önállóan képes a kód módosításától a felhőben történő éles telepítésig minden lépést automatikusan végrehajtani.

0. lépés - Előkészületek

A projekt megkezdése előtt szükséges a fejlesztéshez és a felhőalapú telepítéshez használt fiókok és eszközök előkészítése.

Első lépésként létre kell hozni egy GitHub-fiókot a forráskód verziókezeléséhez és a GitHub Actions CI/CD folyamatainak futtatásához, valamint egy AWS (Amazon Web Services) fiókot, amely a felhőinfrastruktúrát biztosítja az alkalmazás hosztolásához.

Javaslom a GitBash és ZIP parancs (MSYS2) feltelepítését.

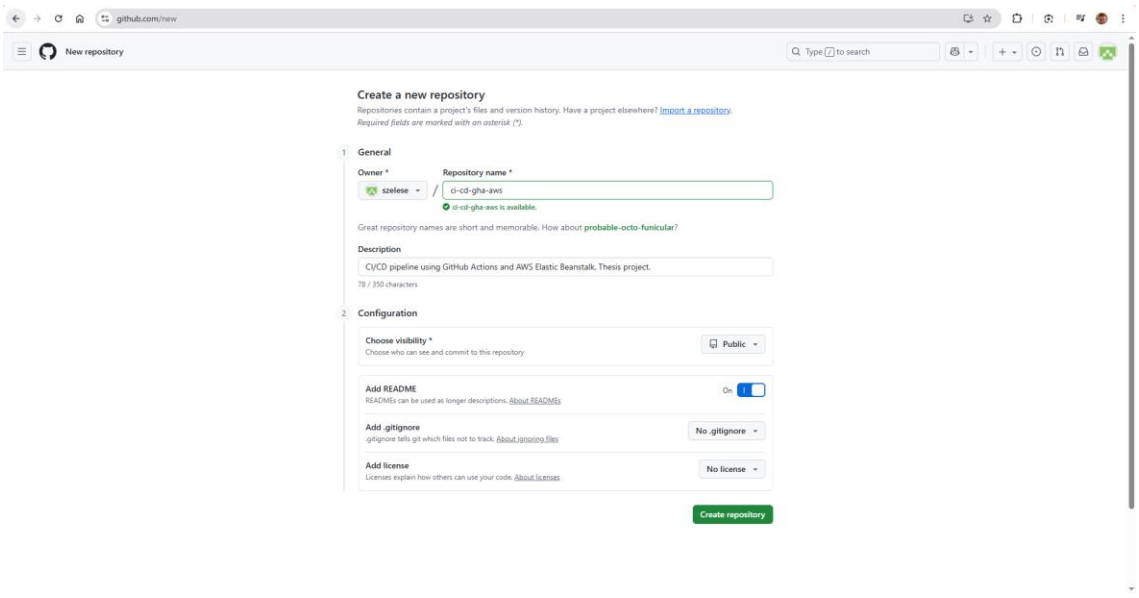
Opcionális:

A fejlesztéshez Python 3.11 futtatókörnyezetet használtam, helyben Visual Studio Code segítségével kezeltem, azonban a projekt kompatibilis bármely más fejlesztői környezettel (pl. PyCharm, IDLE Shell).

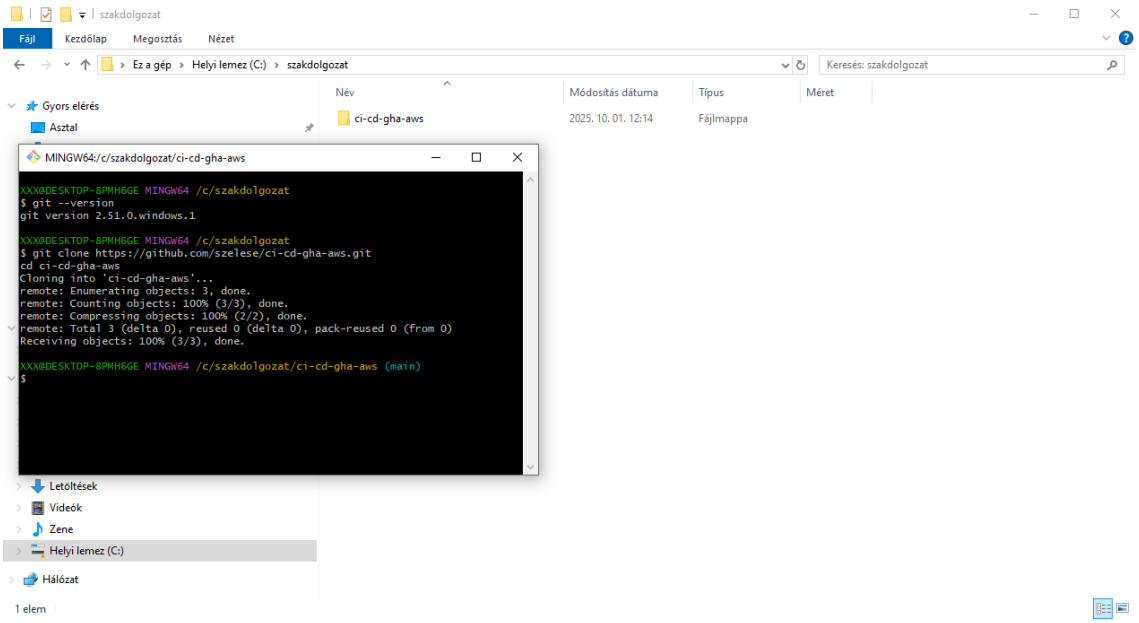
Nagyon egyszerű szövegszerkesztővel is meg lehet oldani, de figyeljünk oda, mert hibás lehet a sortörés vagy end of file jelzése.

Az alkalmazás fejlesztése és tesztelése Windows 10 Enterprise operációs rendszeren történt, de természetesen bármilyen operációs rendszeren futtatható.

1. lépés: új repo: ci-cd-gha-aws létrehozása

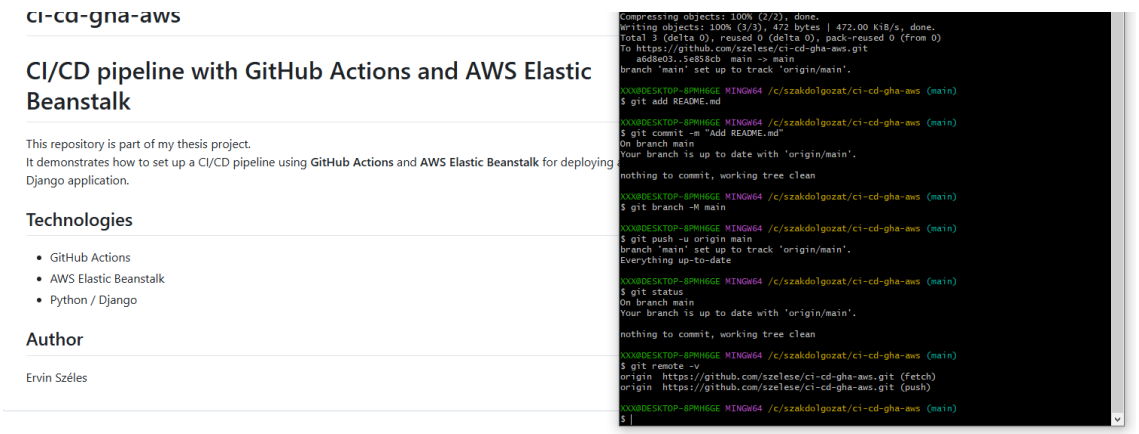


2. lépés: GitBash ellenőrzése és repo klónozása saját gépre

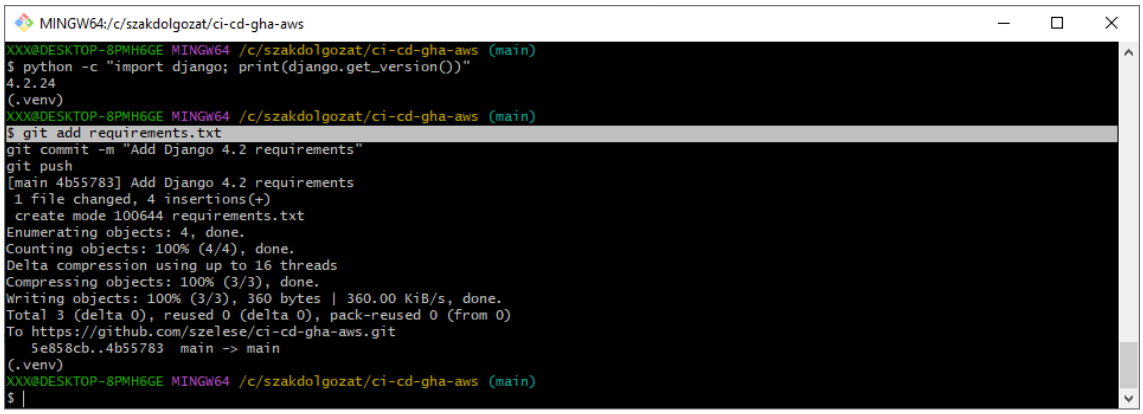



```
git - -version
git clone https://github.com/szelese/ci-cd-gha-aws/
```

3.lépés: README.md és requirements.txt módosítás, commit, push és ellenőrzés

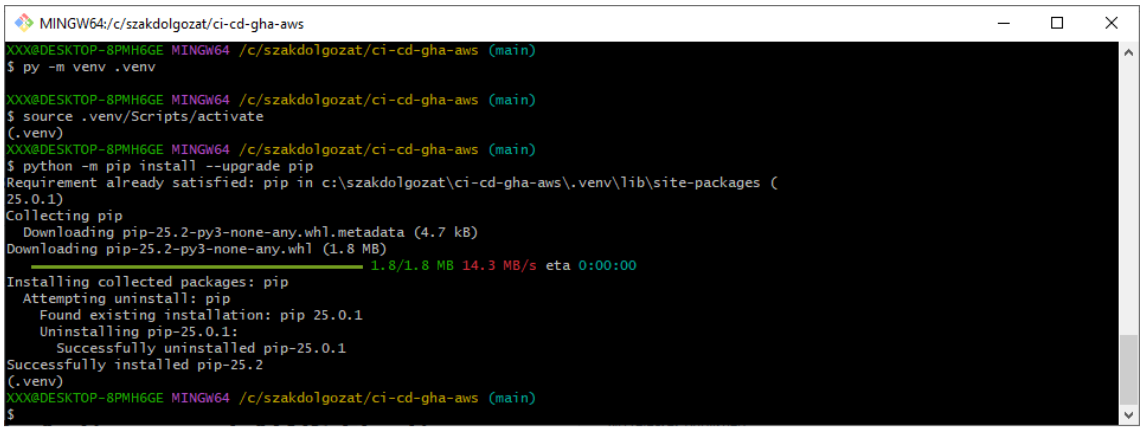



```
git add README.md
git commit -m "Add README.md"
git branch -M main
git push -u origin main
git status
git remote -v
```



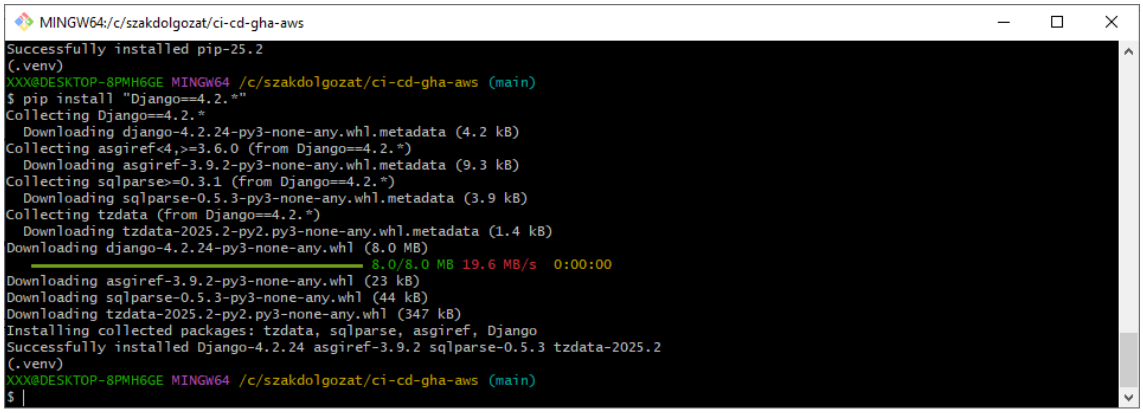

```
git add requirements.txt
git commit -m „Add Django 4.2 requirements”
```

4.lépés: virtuális környezet létrehozása, aktiválás és pip frissítés



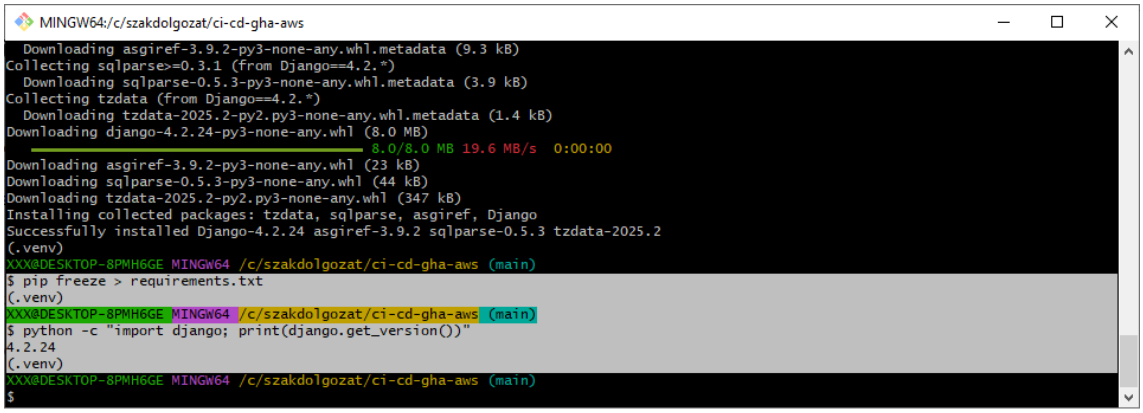

```
py -m venv .venv
source .venv/Scripts/activate
# (.venv) ha a prompt elején megjeleik, akkor azt jelenti, hogy aktív
python -m pip install --upgrade pip
```

5.lépés: Django telepítése (LTS 4.2)



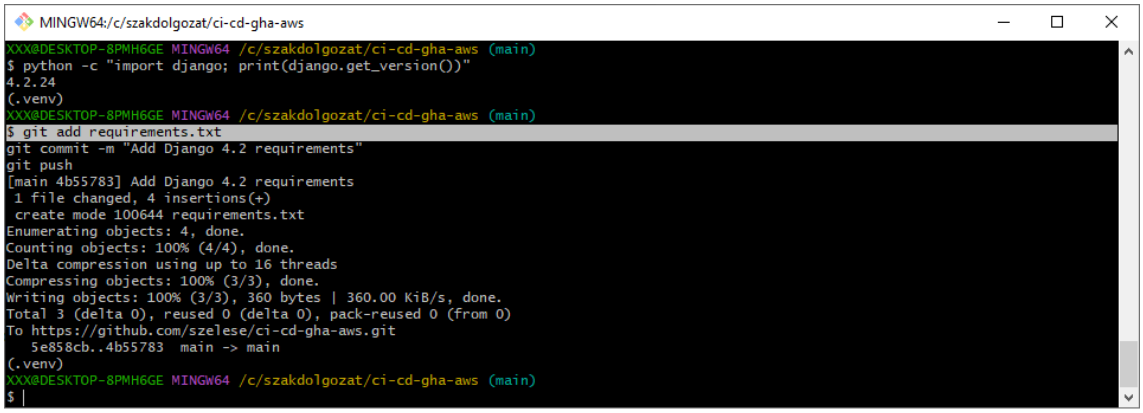

```
pip install "Django==4.2.*"
```

6.lépés: követelmények fájl kiírása és gyors ellenőrzés



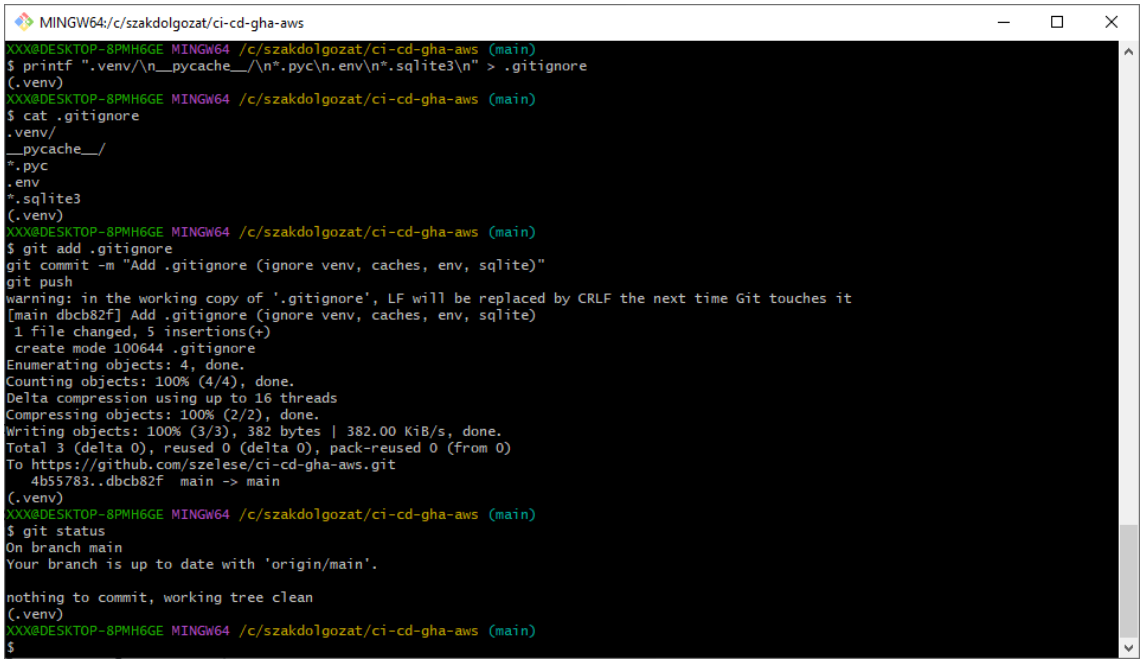

```
pip freeze > requirements.txt
python -c "import django; print(django.get_version())"
# várható és helyes ellenőrzési érték: 4.2.*
```

7. lépés: commit&push




```
git add requirements.txt
git commit -m "Add Django 4.2 requirements"
git push
```

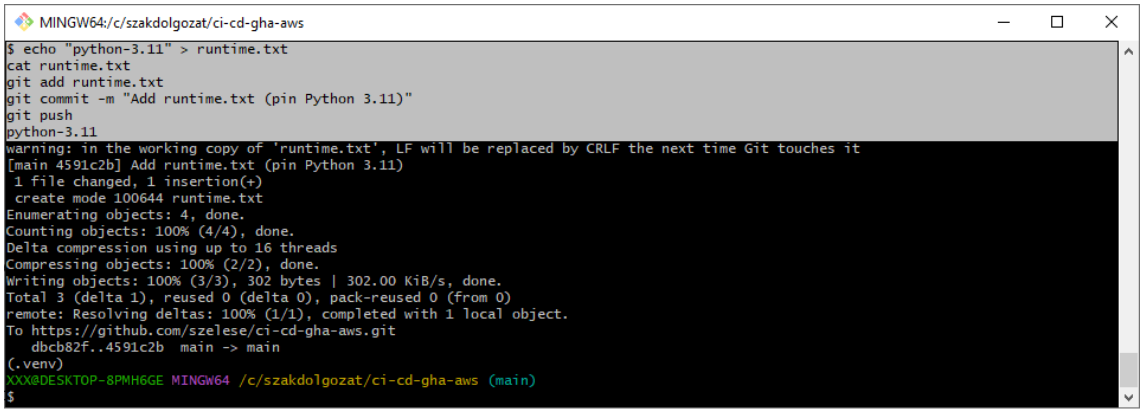
8. lépés: gitignore létrehozása, ellenőrzés



A .venv gép- és OS-függő, hatalmas lehet, és CI-ben/új gépen requirements.txt alapján épül újra. Így marad kicsi és tiszta a repó.

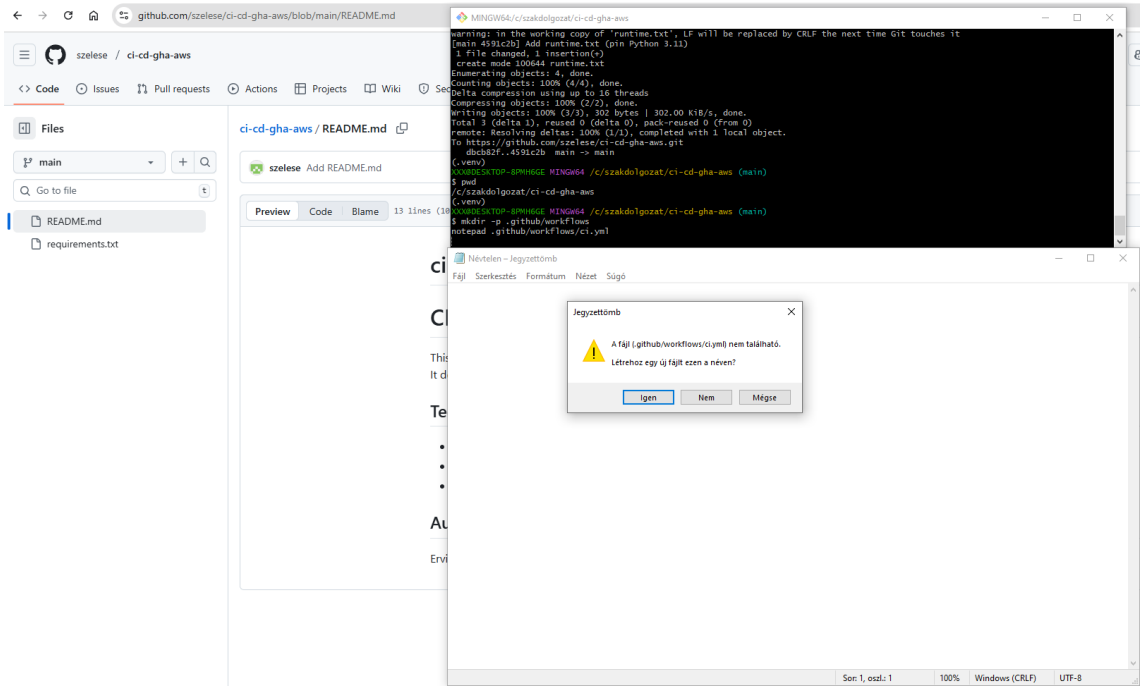
```
printf ".venv/\n__pycache__/\n*.pyc\n.env\n*.sqlite3\n" > .gitignore
cat .gitignore
git status
```

9. lépés: runtime kimenet létrehozása a pontos tartalommal, ellenőrzés, majd commit&push



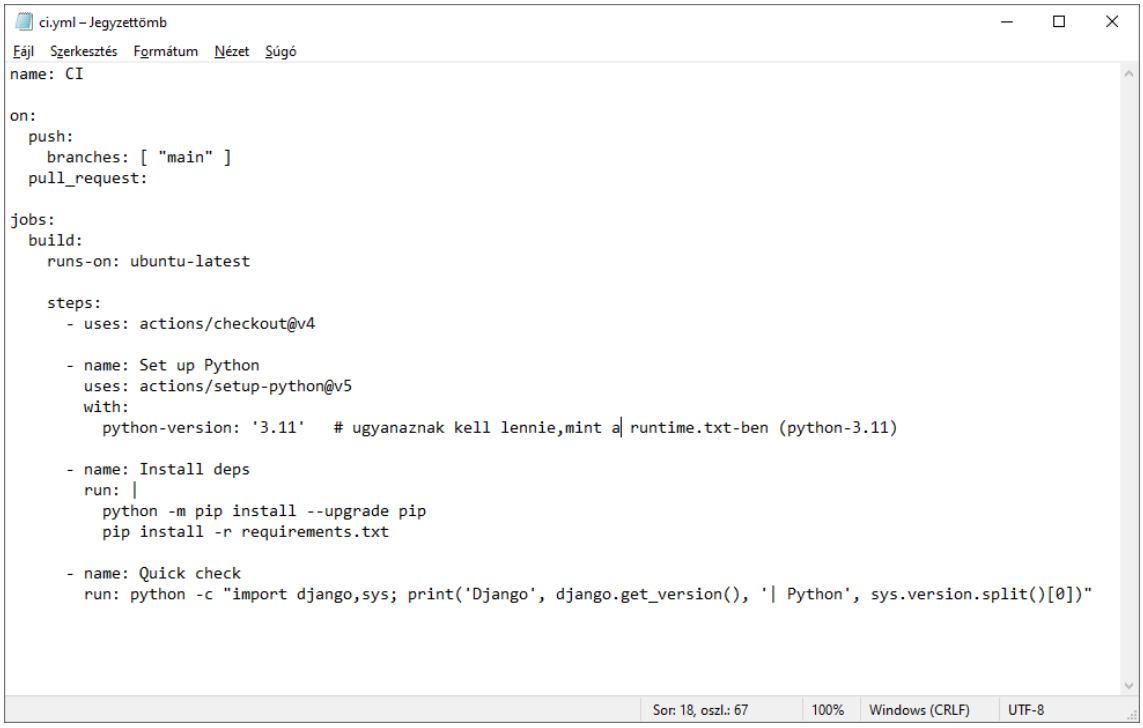

```
$ echo "python-3.11" > runtime.txt
cat runtime.txt
git add runtime.txt
git commit -m "Add runtime.txt (pin Python 3.11)"
git push
python-3.11
```

10. lépés: Workflow mappa és ci.yml létrehozása



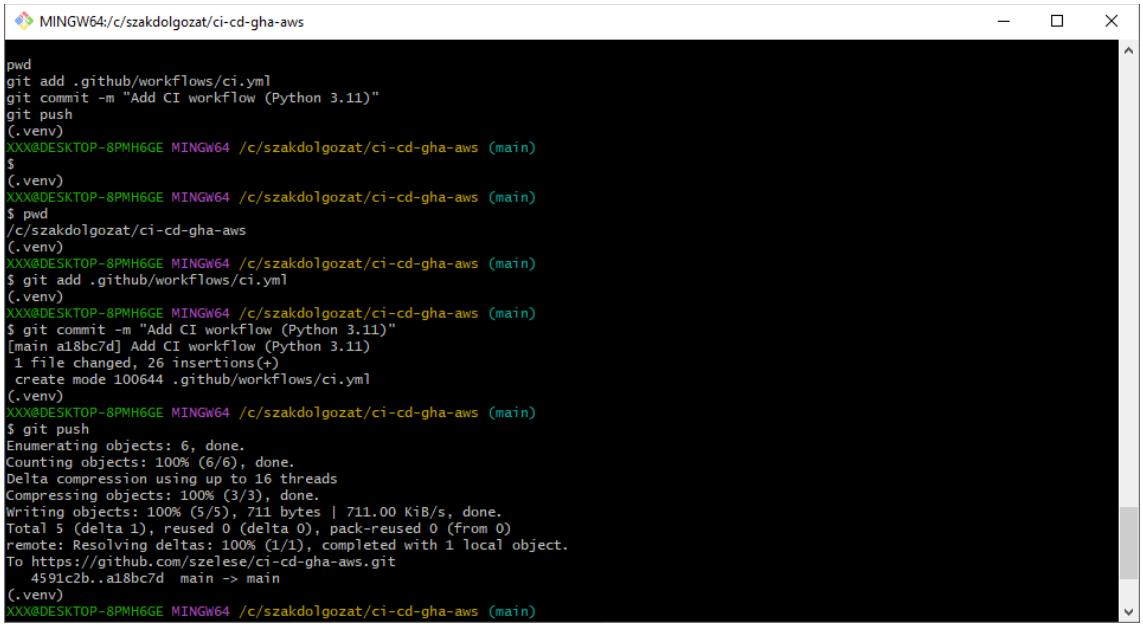

```
$ mkdir -p .github/workflows
notepad .github/workflows/ci.yml
```

11. lépés: a ci.yml file tartalma



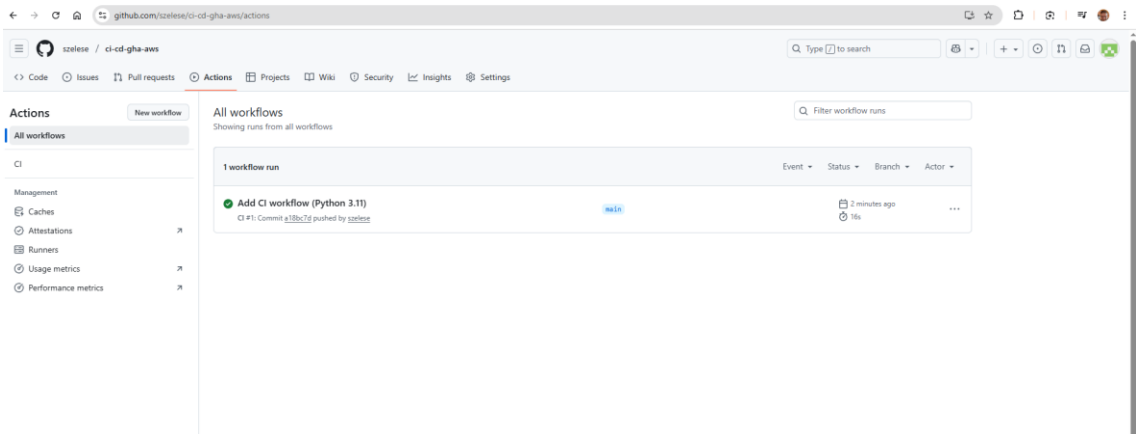
A runtime.txt biztosítja, hogy az EB pont ugyanazzal a Python-verzióval futtassa a projektet, mint amit használunk.

12. lépés: commit&push



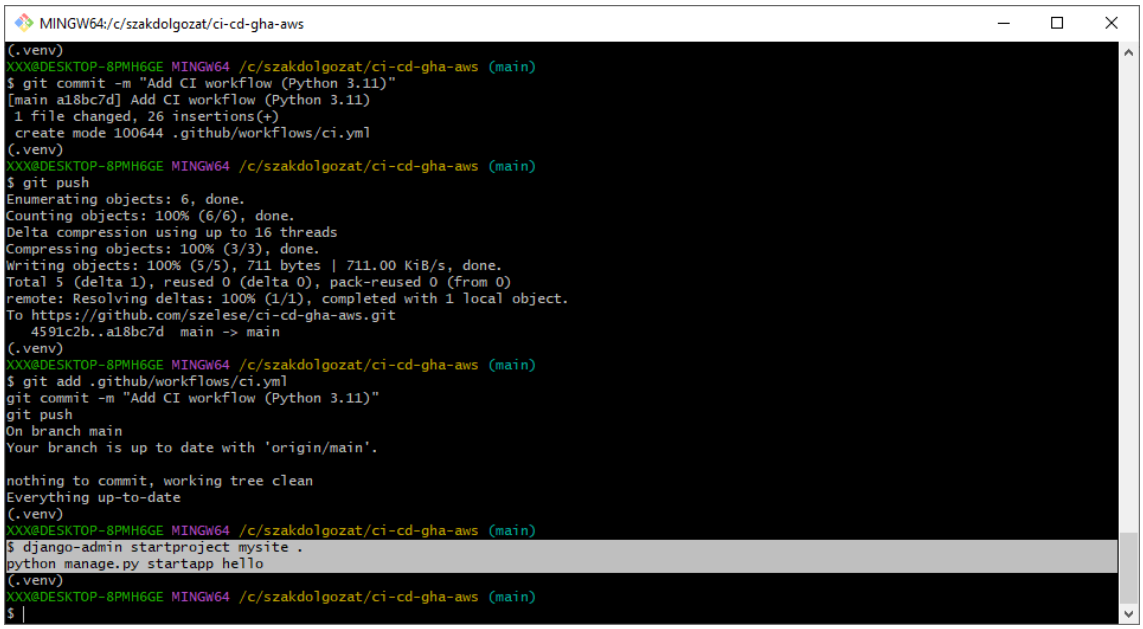

```
commit és push ci.yml -t
git add .github/workflows/ci.yml
git commit -m „Add CI workflow (Python 3.11)”
```

13. lépés: GitHub ellenőrzés



A GitHub repóban az Actions fül alatt megjelenik a „CI” futás. Zöld pipa = minden rendben.

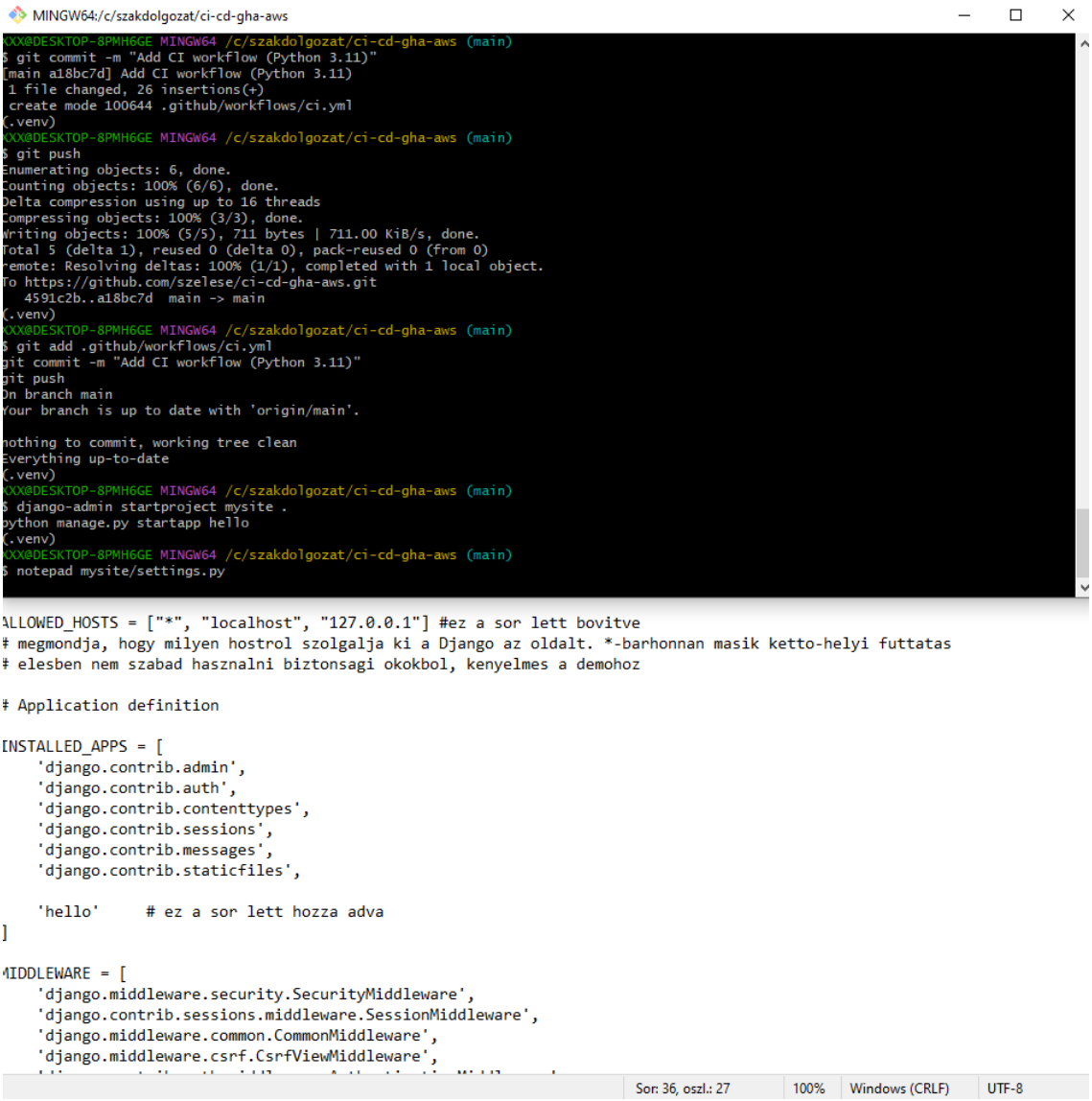
14. lépés: project és app létrehozása



egy mysite nevű project és hello nevű alkalmazás létrehozása:

```
django-admin startproject mysite .
python manage.py startapp hello
```

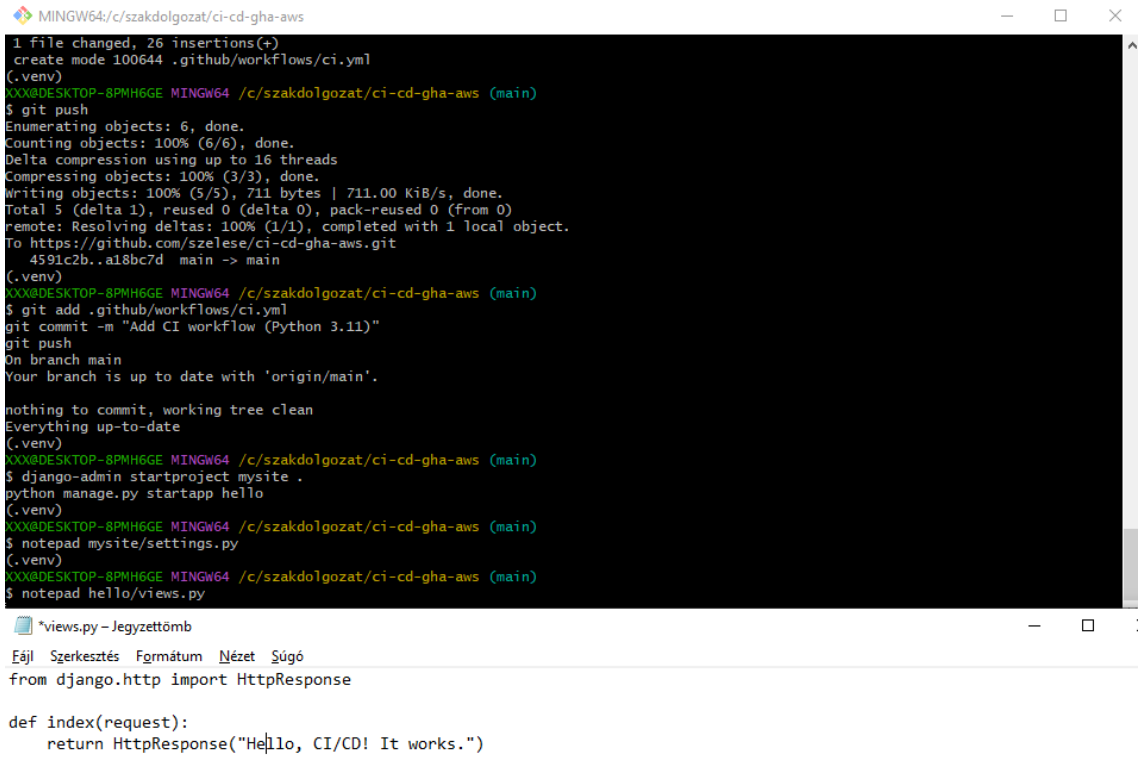
15. lépés: app regisztrálása és a demóhoz engedése




```
notepad mysite/settings.py
INSTALLED_APPS = [
    # ...
    'hello',
]
és
ALLOWED_HOSTS = ["*", "localhost", "127.0.0.1"]
```

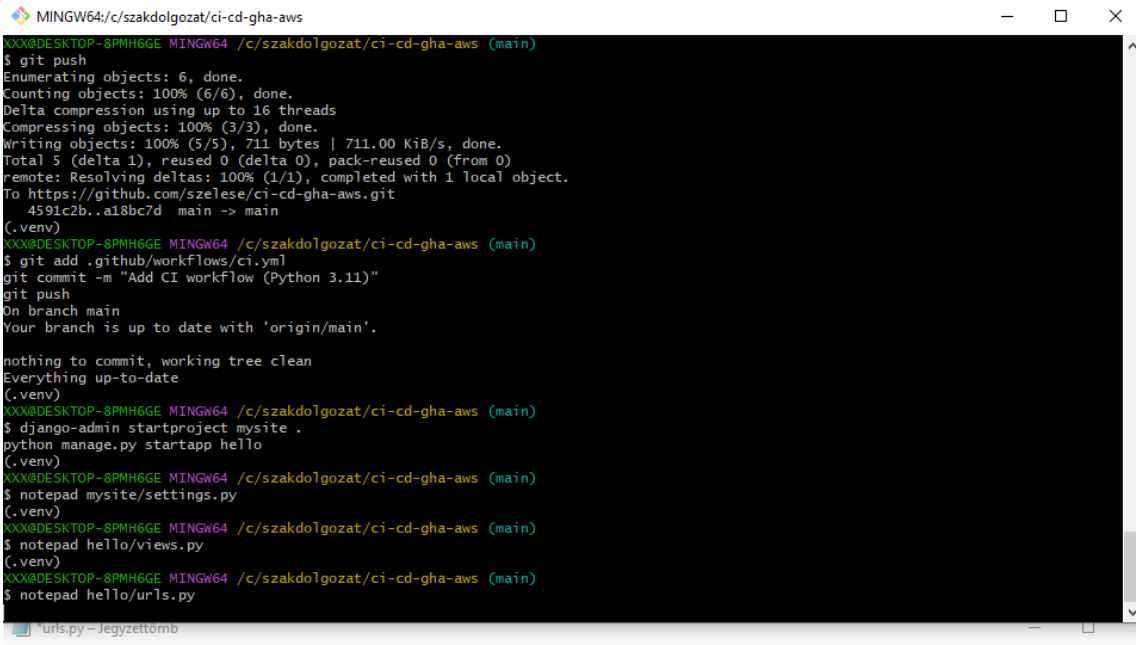
Figyelem: Az ALLOWED_HOSTS = ['*'] beállítás csak a kezdeti teszteléshez használható, éles környezetben biztonsági okokból kötelező a konkrét domain(ek) megadása!!!

16. lépés: minimál nézet írása




```
notepad hello/views.py # views.py megnyitása után:
from django.http import HttpResponse
def index(request):
    return HttpResponse("Hello, CI/CD! It works.")
```

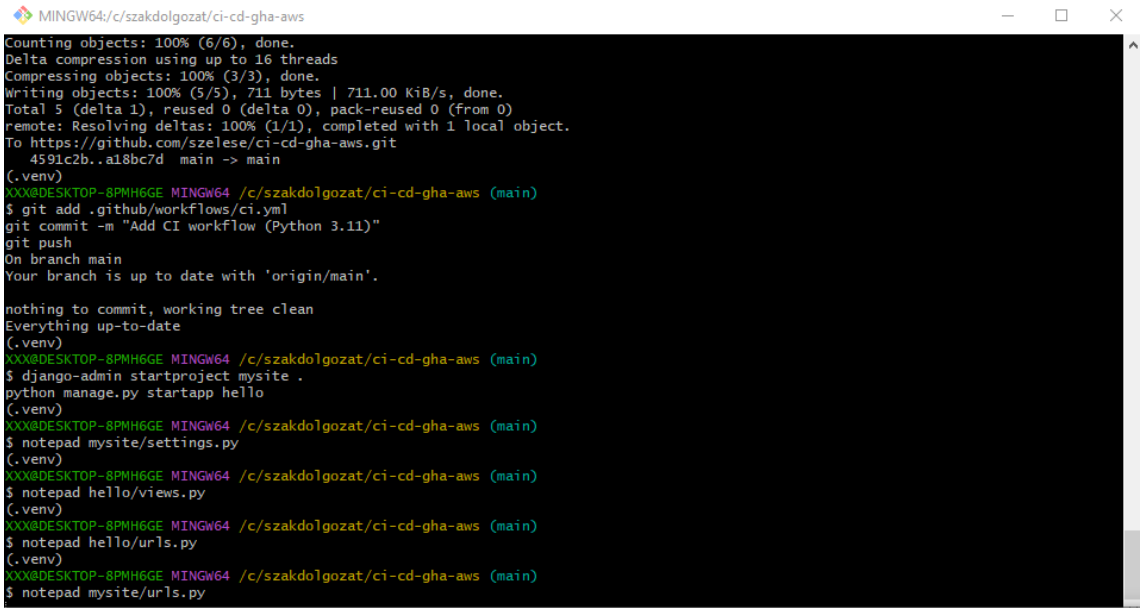
17. lépés: URL bekötése a hello alkalmazásba

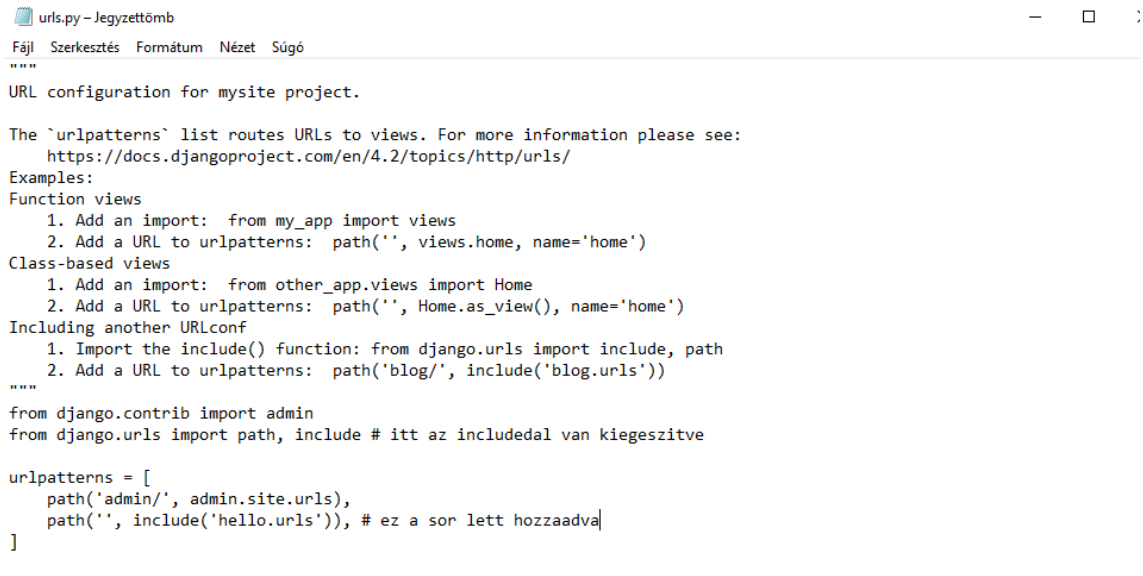


notepad hello/urls.py # létrehozás és tartalom:

```
from django.urls import path
from . import views
urlpatterns = [    path("", views.index, name="index"),]
```

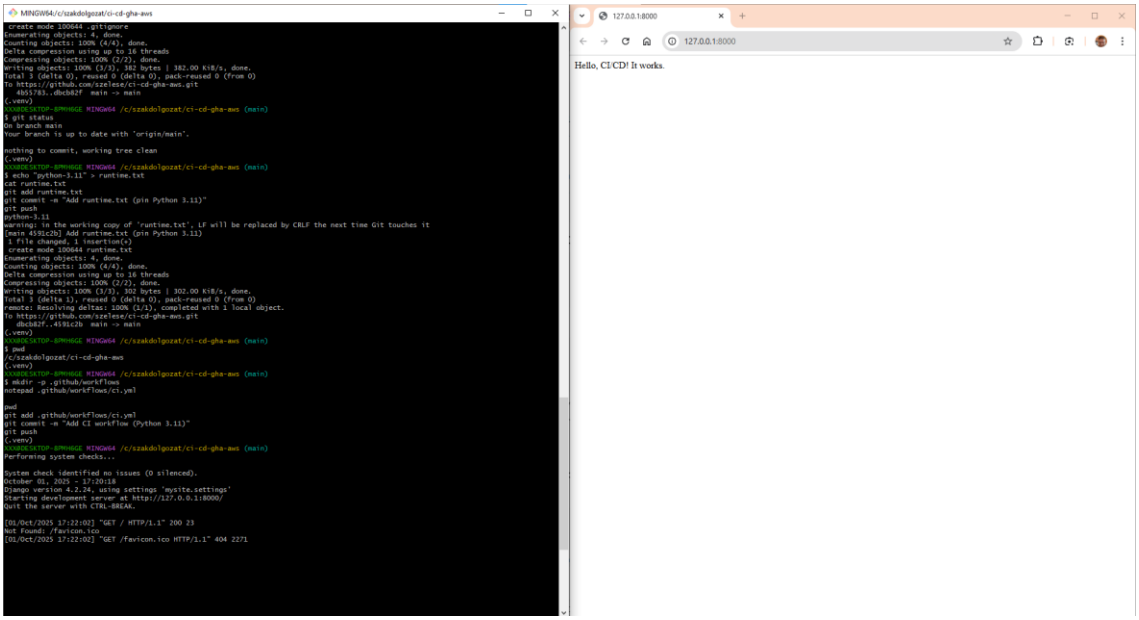
18. lépés: Az app URL-jeit a projekt fő URL fájljába bekötjük (mysite/urls.py)





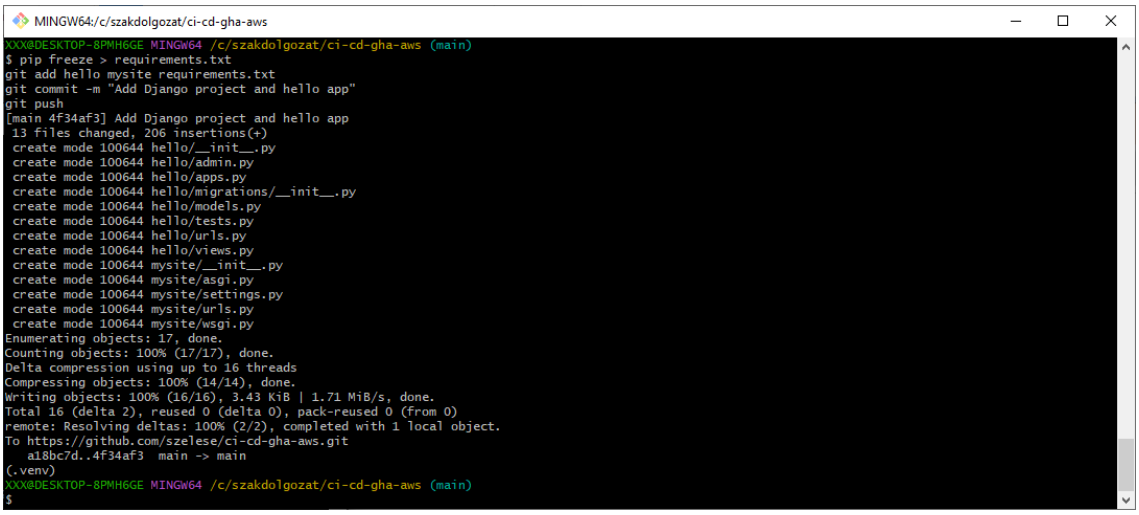

```
notepad mysite/urls.py # megnyitas és modositas:
from django.contrib import admin
from django.urls import path, include
urlpatterns = [    path('admin/', admin.site.urls),
    path('', include('hello.urls')),]
```

19. lépés: adatbázis inicializálás és helyi futtatás teszt



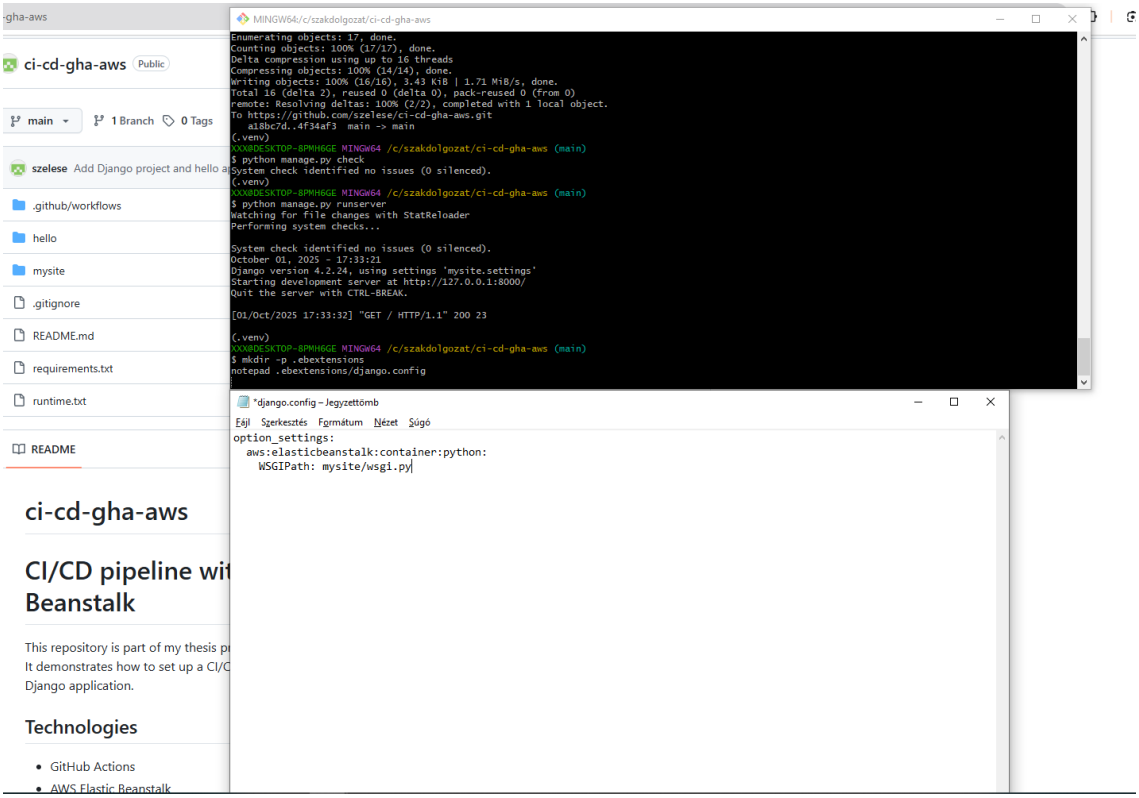

```
python manage.py migrate # db inicializálása
python manage.py runserver
# utána a böngészőben tesztelés : http://127.0.0.1:8000/
```

20. lépés: követelmények frissítése és push



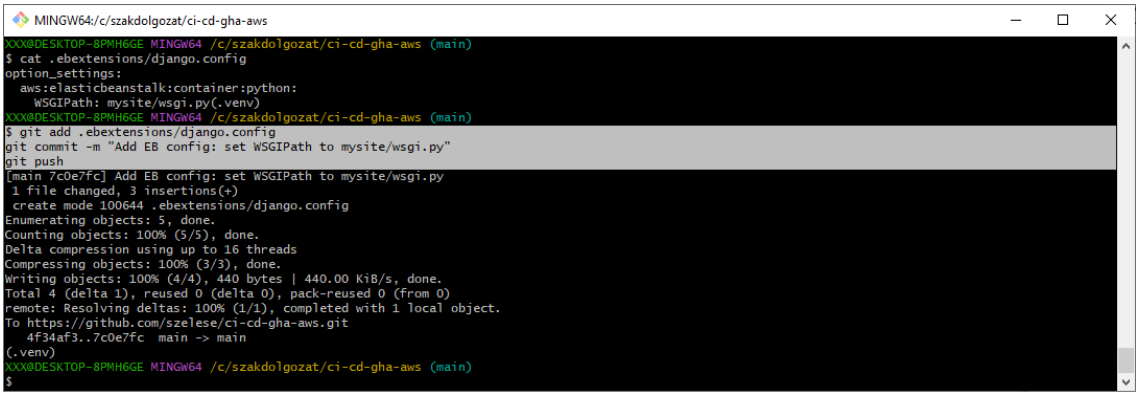

```
pip freeze > requirements.txt
git add hello mysite requirements.txt
git commit -m "Add Django project and hello app"
git push
```

21. lépés: Elastic Beanstalk-hoz szükséges beállítás, mappa és config file létrehozás




```
mkdir -p .ebextensions
notepad .ebextensions/django.config
django.config file tartalma: # szóközök!!!
option_settings:
  aws:elasticbeanstalk:container:python:
    WSGIPath: mysite/wsgi.py
```

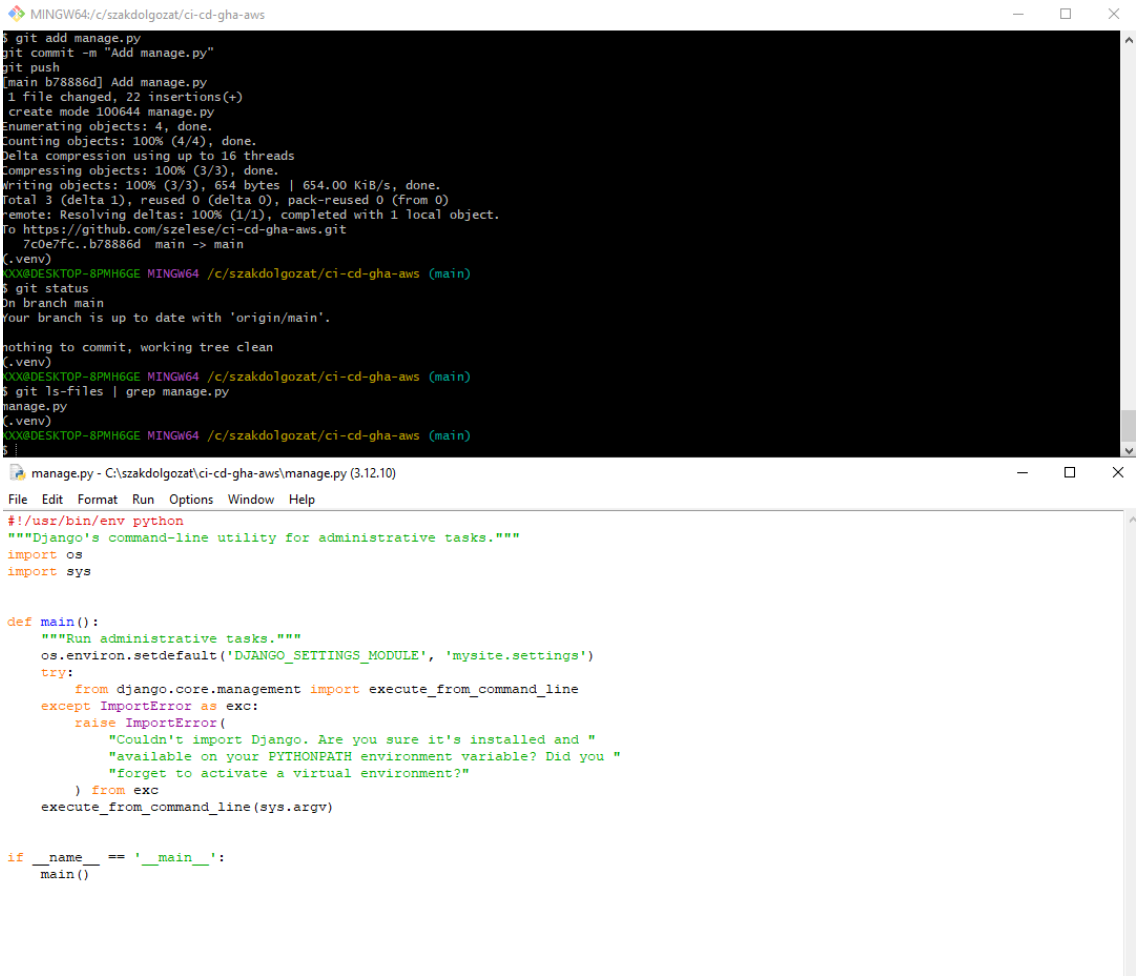
22. lépés: ellenőrzés, commit&push



commit&push után még egy ellenőrzés a GitHubon, hogy létrejött a file ezzel a tartalommal

```
cat .ebextensions/django.config
git add .ebextensions/django.config
git commit -m "Add EB config: set WSGIPath to mysite/wsgi.py"
git push
```

23. lépés: manage.py beállítása



Elastic Beanstalk futtatásához elvileg a manage.py nem kötelező (a WSGI-t használja), de helyi fejlesztéshez, migrációkhoz, collectstatic-hez és későbbi automatizmusokhoz nagy szükség lesz.

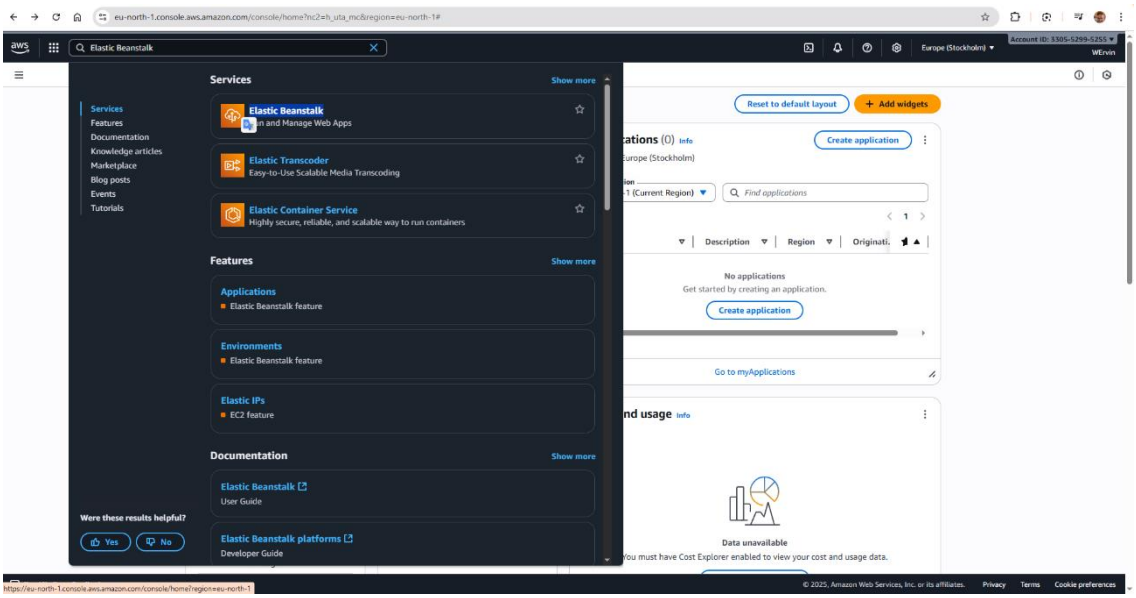
```
notepad manage.py
git add manage.py
git commit -m "Add manage.py"
git push
```

24. lépés: deploy.zip létrehozása

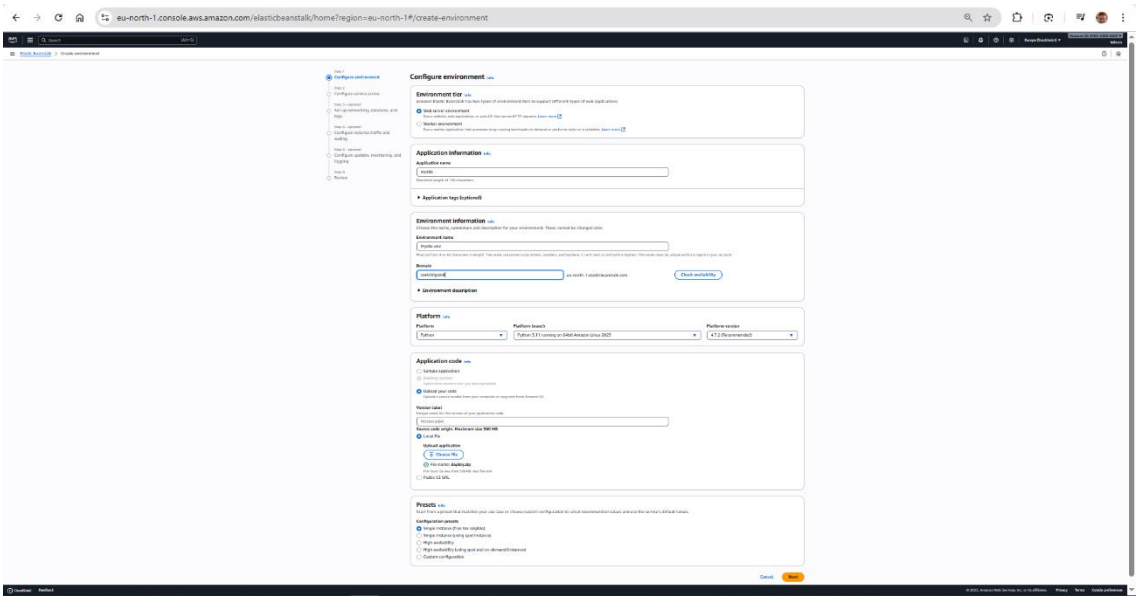

```
zip -r deploy.zip . -x ".git/*" ".venv/*" "__pycache__/*" "*.pyc"
```

MSYS2-ből telepítettem a zip-et, hogy elegánsan mindent parancssorból megtudjak oldani, de simán intézőből vagy egérrel is be lehet csomagolni .git és .venv nélkül

25. lépés: AWS search / Elastic Beanstalk/Create Application



26. lépés: AWS környezetbeállítás



Environment tier: Web Server Environment # alapbeállítás

Application information: mysite

environment information:

environment name: Mysite.env

Domain: szakdolgozat # de tetszőleges jó

Platform: Python

platform branch: Python 3.11 # runtime.txt -ben ami van

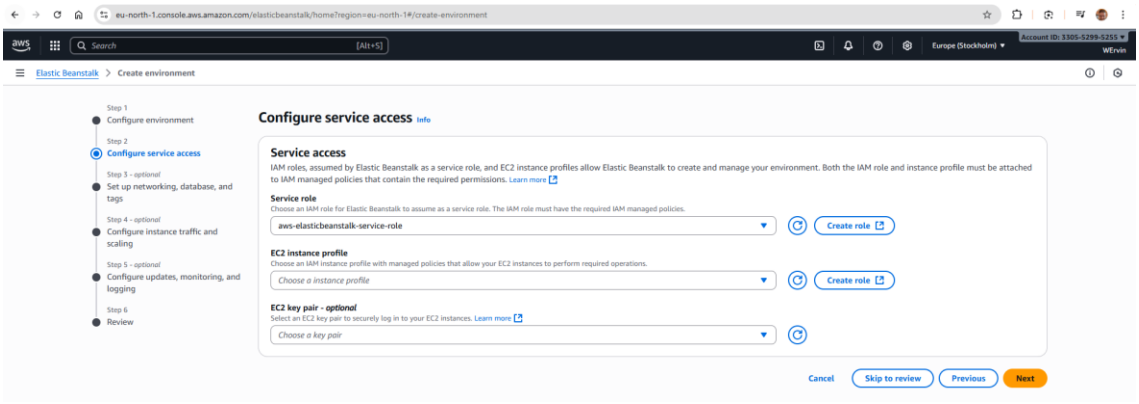
Version label: mysite-v1 # tetszőleges

Application code: Upload your code ->local file-> deploy.zip # amit a 24. lépésnél lett generálva

Presets:

Single instance (free tier eligible) # demóhoz tökéletes

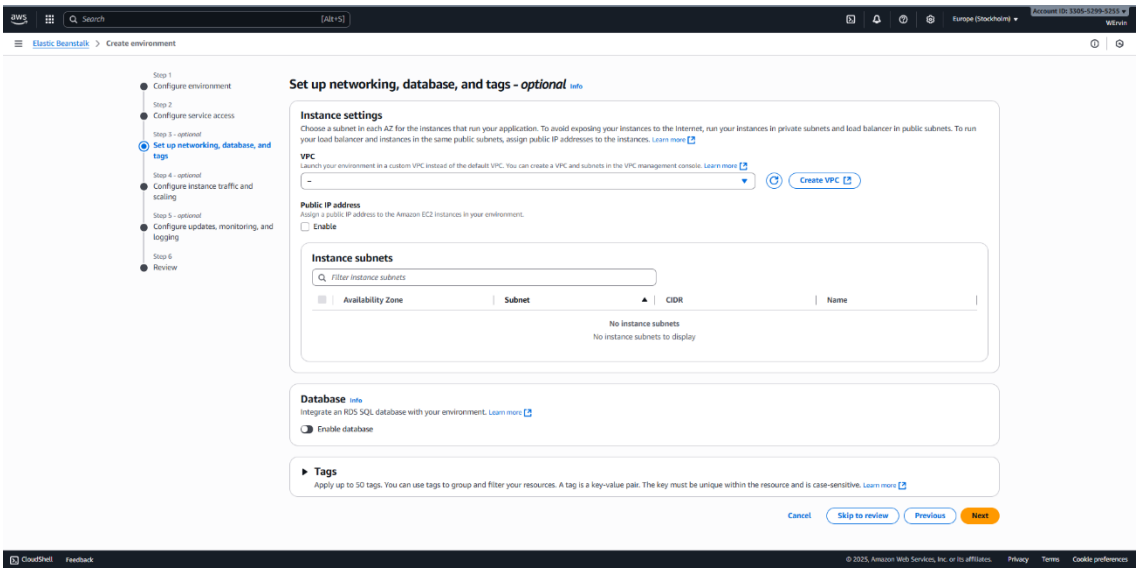
27. lépés: step2 configure service access



create role-nál az automatikusan kitöltötteket jóvá kell csak hagyni és ki kell jelölni a megadott role-okat

Az EC2 key pair (optional) mezőt hagyhatjuk üresen. # SSH beállításokat elég akkor beállítani amikor konzolból szeretnénk vezérelni a szervert.

28. lépés: step3 - optional # csak a teljesség kedvéért



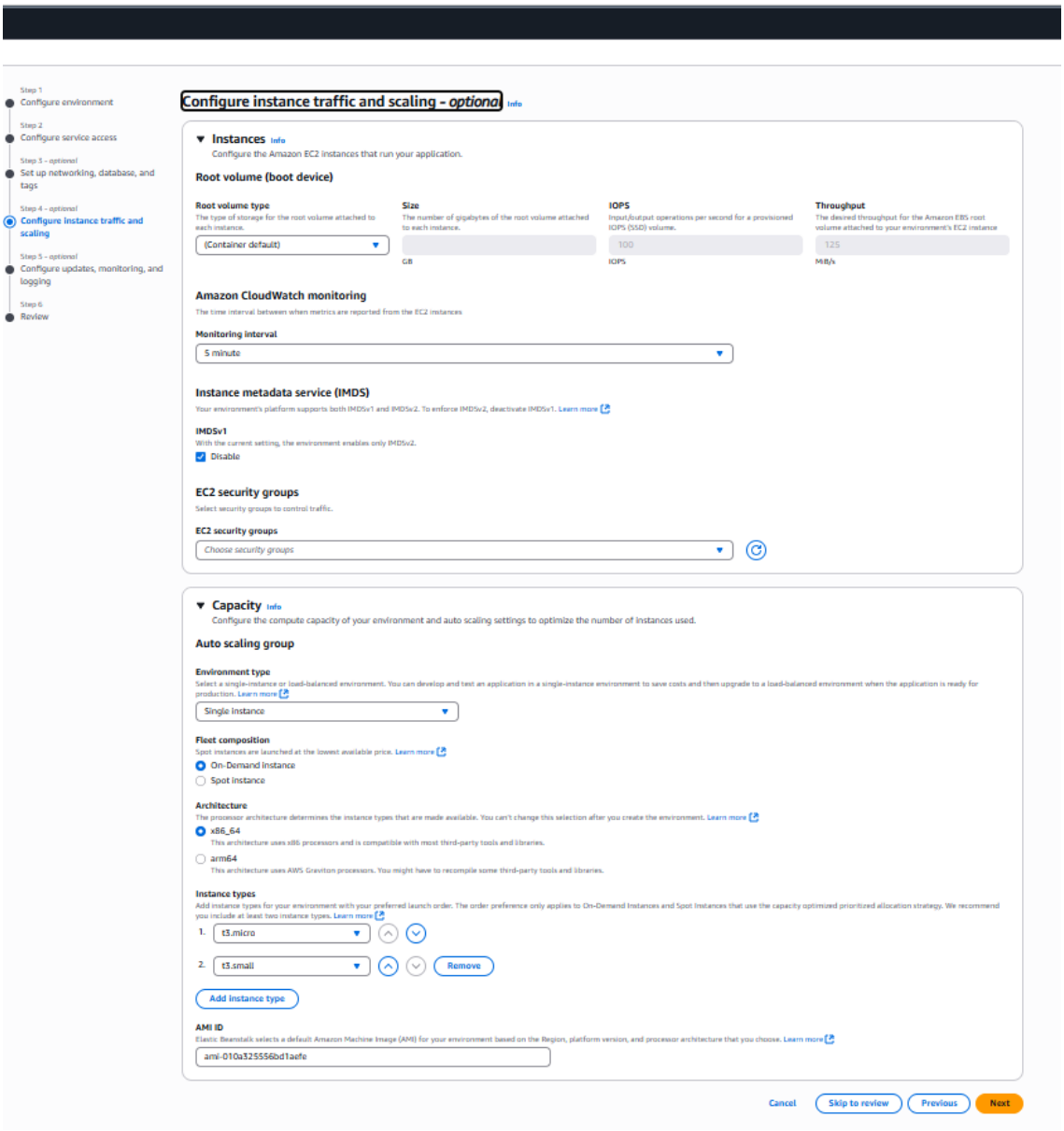
VPC: hagyd a - (default) értéken. Ne hozzunk létre új VPC-t, ne válasszunk subnetet.

Public IP address: maradhat kikapcsolva - a Single instance presetnél az EB alapból publikussá teszi az instance-ot. (Ha később mégsem lenne elérhető kívülről, utólag bekapcsolható: Environment → Configuration → Instances → Public IP: Enable.)

Database: ne kapcsoljuk be (nem használunk most RDS-t). (A rendszer később bővíthető DB-vel.)

Tags: opcionális (nyugodtan hagyjuk alapbeállításon, de természetesen később haladó beállításnál finomíthatunk rajta)

29. lépés: a demóhoz tökéletesek az alapbeállítások is



Root volume: Container default (-on hagyhatjuk).

Monitoring interval: 5 minutes. IMDSv1: Disable (így csak IMDSv2 engedélyezett).

EC2 security groups: hagyd a default EB csoportot.

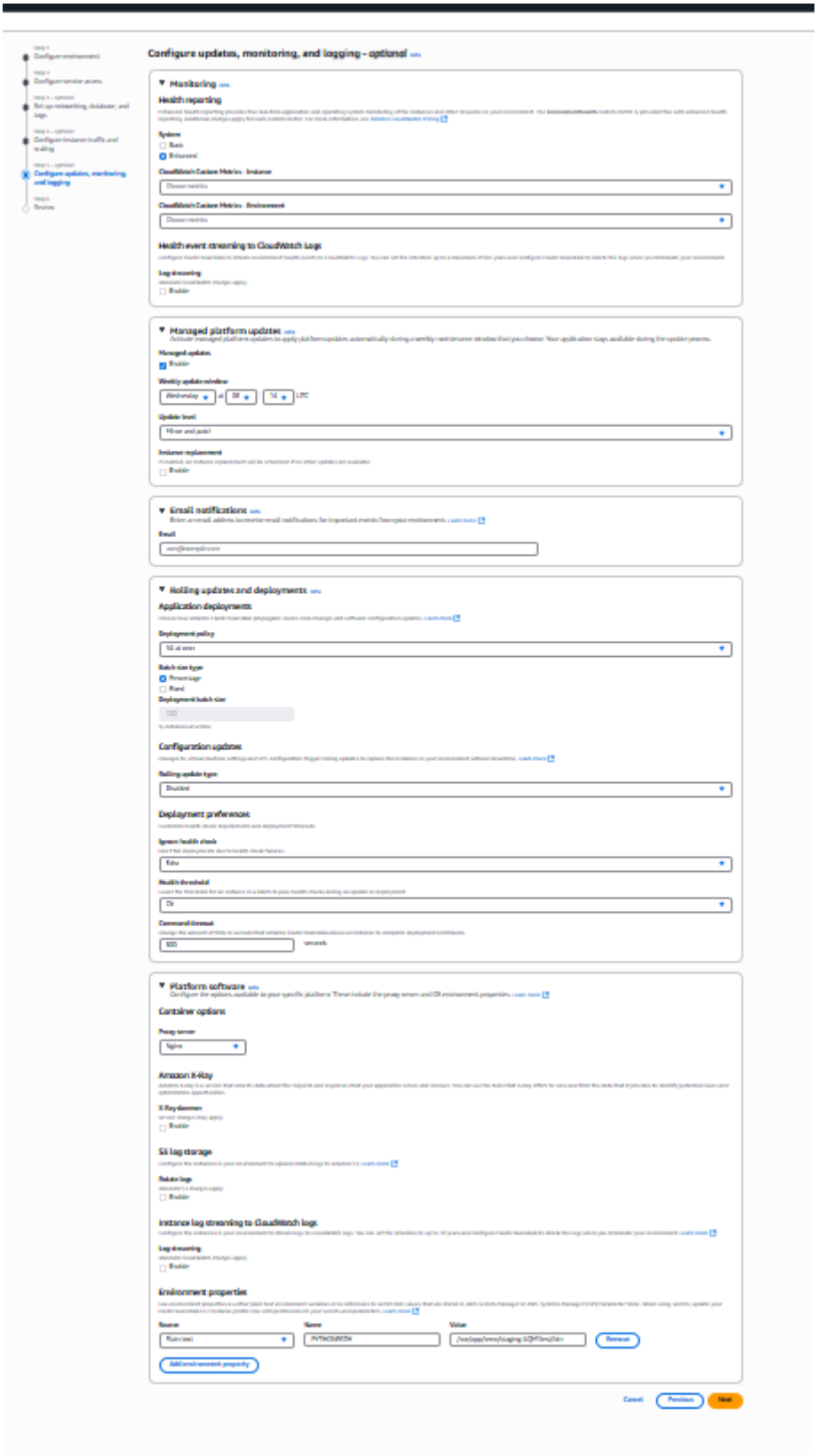
Capacity: Environment type: Single instance

Fleet composition: On-Demand Architecture: x86_64

Instance types: 1) t3.micro 2) (opció) t3.small

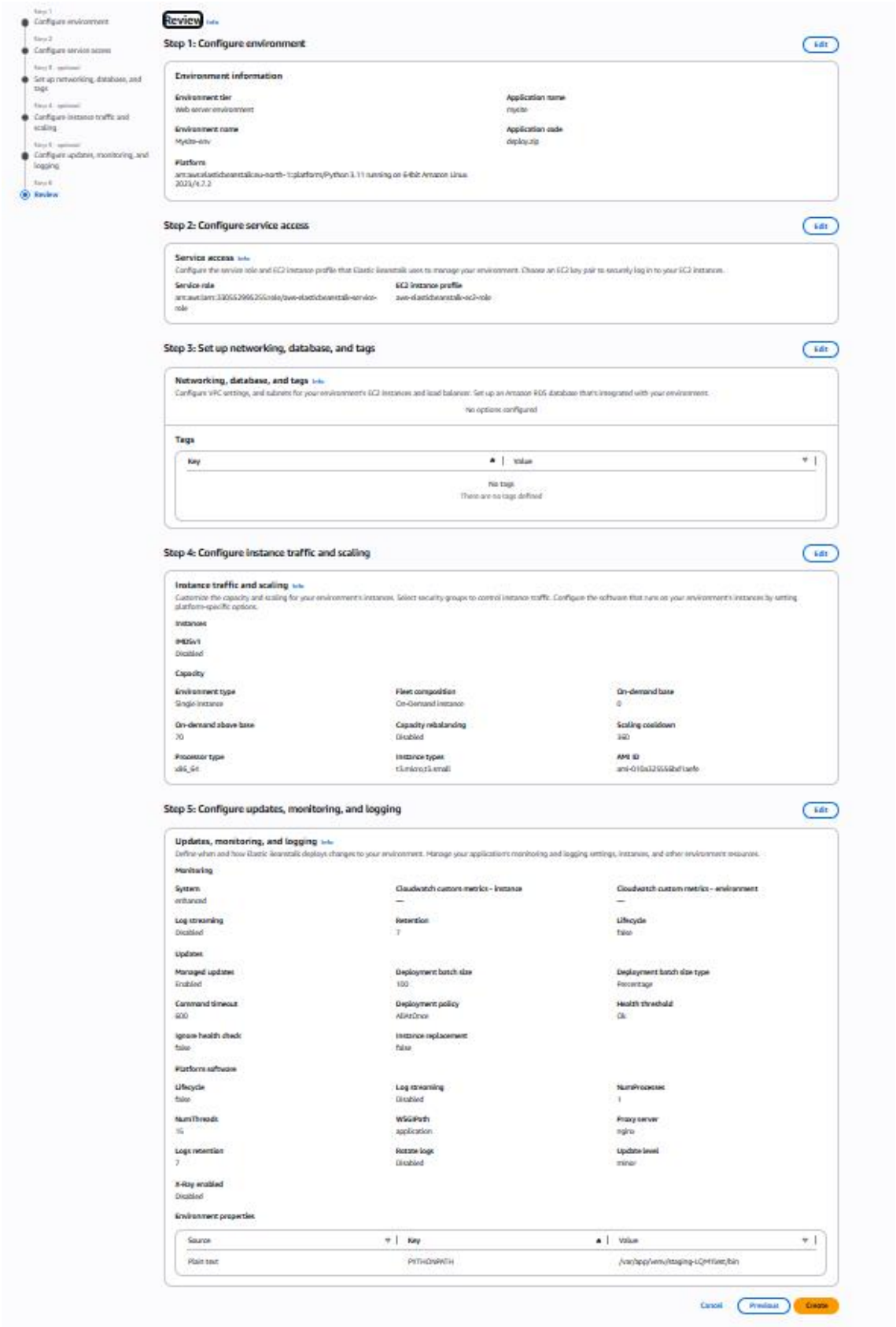
AMI ID: hagyhatjuk az alapértelmezett értéken.

30. lépés: EB Step 5 - Updates/Monitoring/Logging

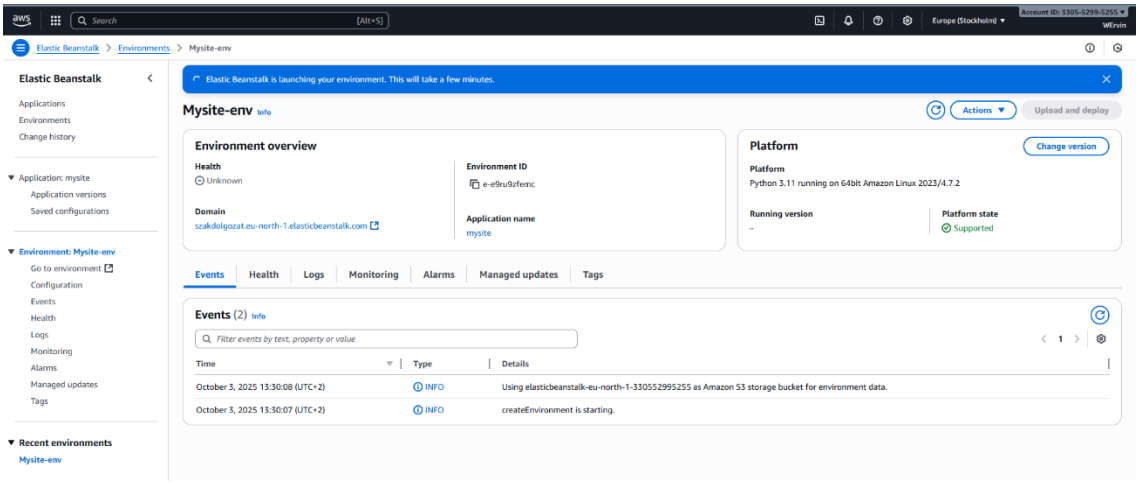


itt is szintén tökéletesek az alapbeállítások # 28, 29 és 30. lépés a teljesség igénye miatt kerül

31. lépés: teljes beállítás nézet review és create

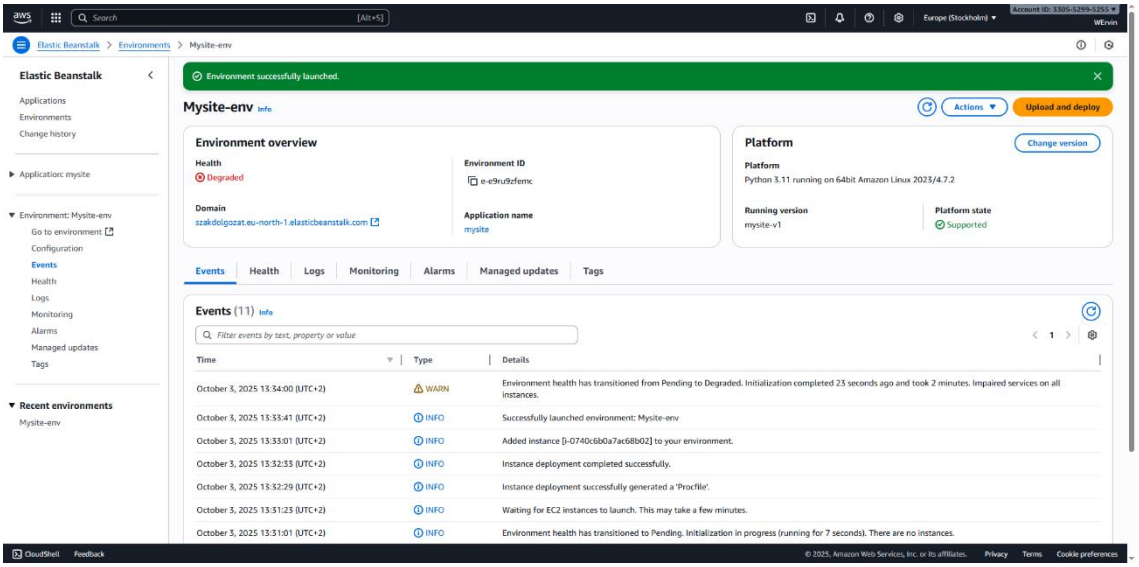


32. lépés: EB környezet indítás és ellenőrzés



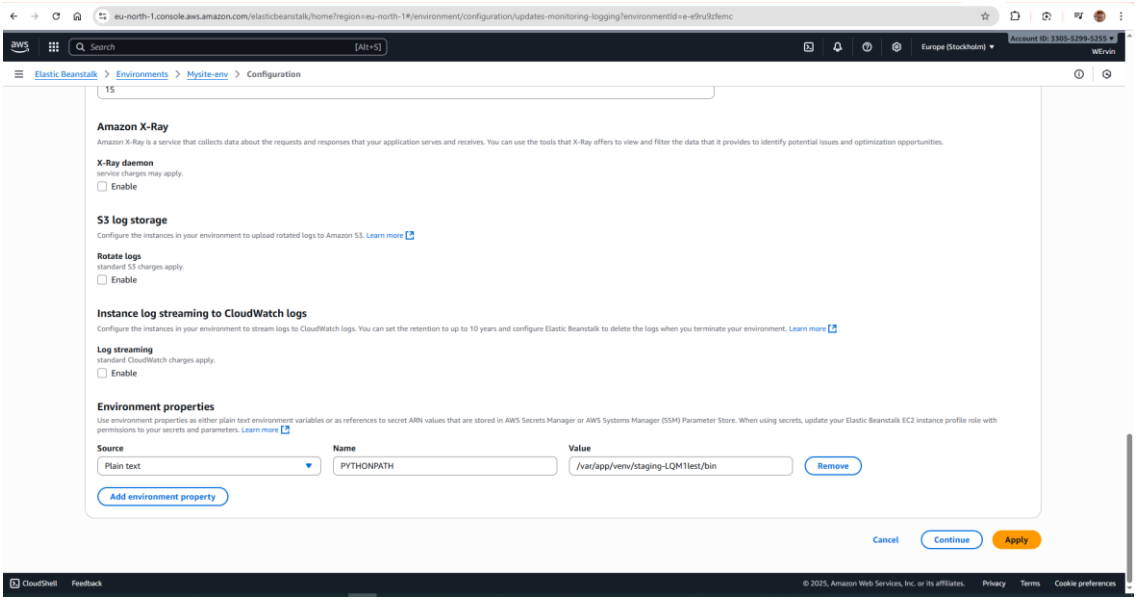
A Health fül zöld „OK” állapotánál a Domain link azonnal elérhető.

33. lépés: Health ellenőrzés



health degraded jelent meg: Ez szinte mindig azt jelenti, hogy az EB health check kérése 200 helyett 400/404/5xx-et kap a / URL-re.

34. lépés: Environment properties hozzáadása



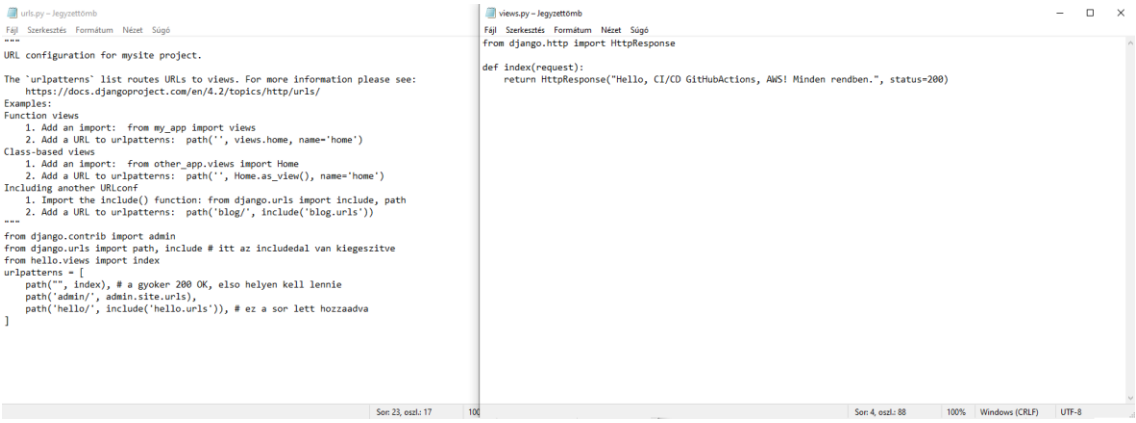
Elastic Beanstalk > Environments > Mysite_env > Configuration > (legalul) Environment properties > Add

Key: DJANGO_SETTINGS_MODULE → Value: mysite.settings

(opcionális) Key: DEBUG → Value: False

(opcionális) Key: SECRET_KEY → Value: A12345678b

35. lépés: a health checkhez lokális file módosítás



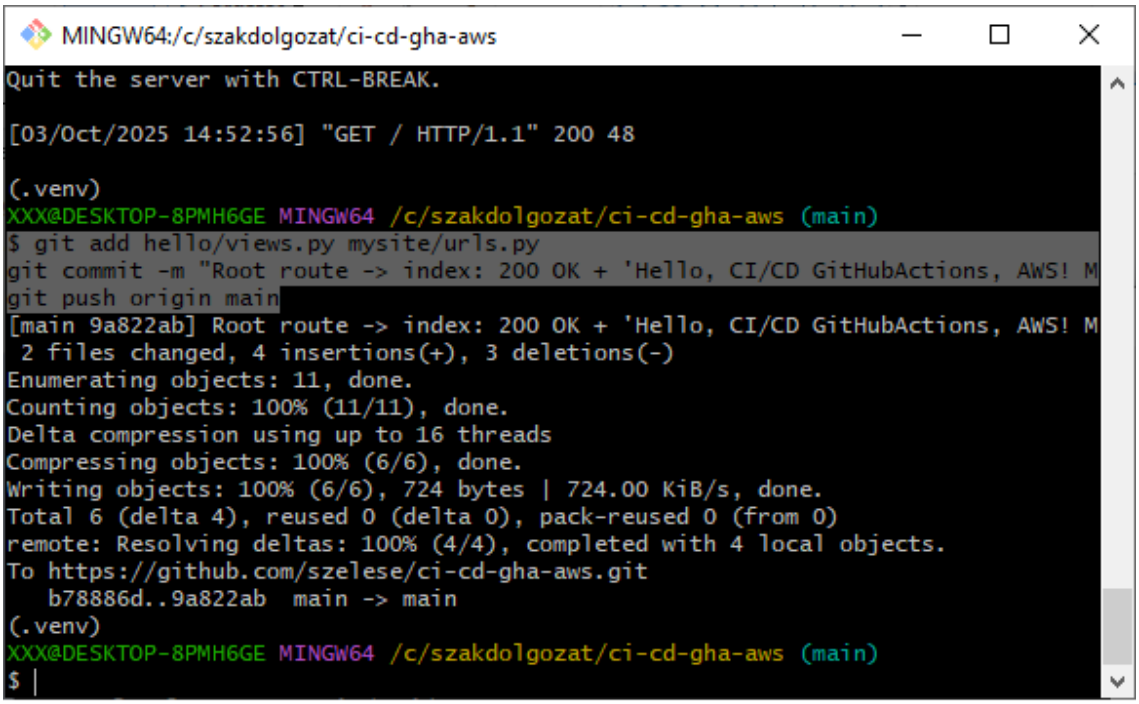
hello/views.py :

```
from django.http import HttpResponse
def index(request):
    return HttpResponse("Hello, CI/CD GitHubActions, AWS! Minden rendben.",
status=200)
```

mysite/urls.py :

```
path("", index), # a gyoker 200 OK, elso helyen kell lennie
python manage.py runserver # gyors lokális próba gitBash, ha minden rendben:
zip -r deploy.zip . \
  -x '.git/*' '.venv/*' '__pycache__/*' '*.pyc' 'deploy.zip'
```

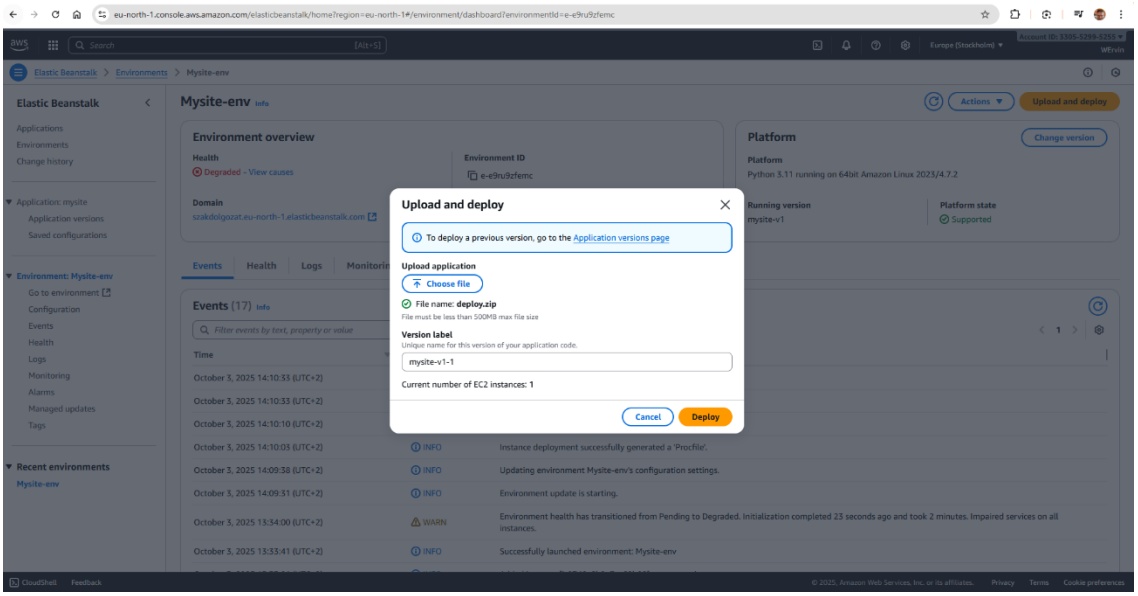
36. lépés: GitBash commit és push




```
git add hello/views.py mysite/urls.py
git commit -m "Root route -> index: 200 OK + 'Hello, CI/CD GitHubActions, AWS! Minden
rendben.'"
git push origin main
```

A sok apró commit és push nem hiba, hanem a CI/CD-folyamat lényege. Ebben a projektben minden git push parancs automatikusan elindítja a GitHub Actions CI-folyamatot (a ci.yml-t), így a rendszer minden egyes apró változtatás után azonnal teszteli magát.

37. lépés: feltöltés az Elastic Beanstalkra



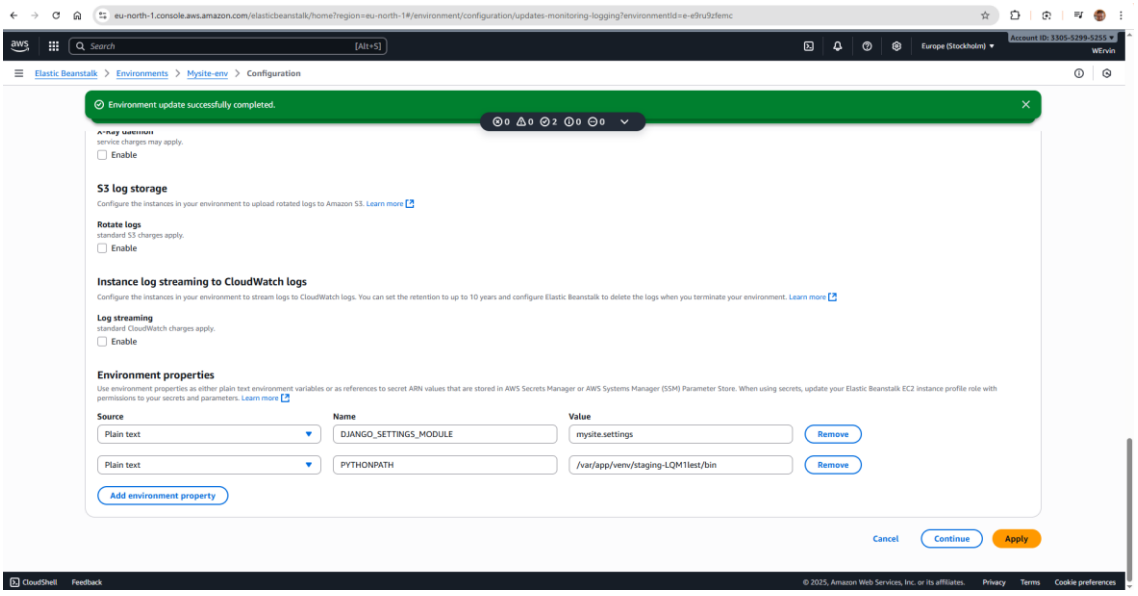
AWS Console → Elastic Beanstalk → Environments → Mysite-env

Jobb felső sarok: Upload and deploy.

Choose file → válasszuk ki a friss deploy.zip-et.

Version label: pl. mysite-v1-1 → Deploy.

38. lépés: a beállított PYTHONPATH miatt nem elérhető a gateway-e, ezért /var/app/current módosítjuk

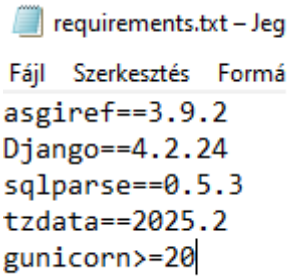


ez sem vezet jó megoldáshoz, valójában a log file elemzés miatt észrevettem, hogy ImportError: No module named 'mysite/wsgi' „Failed to find application, did you mean mysite/wsgi:application?”

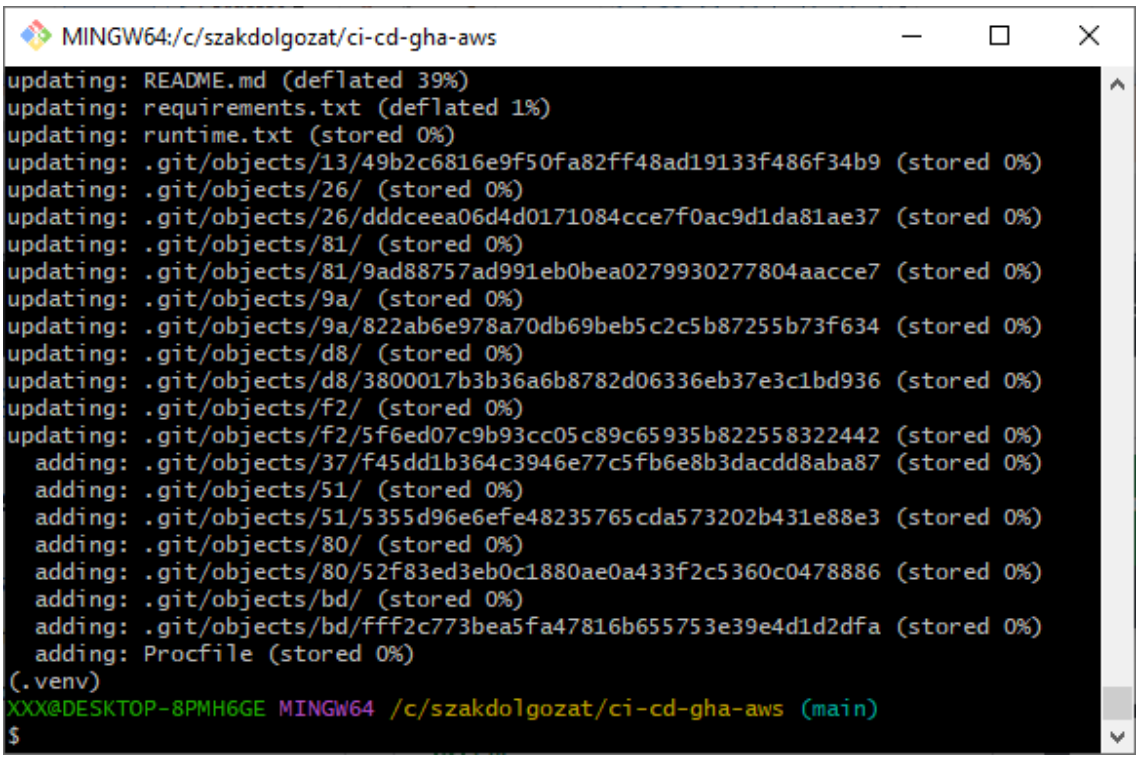
A Gunicorn modulnevet vár pontokkal, nem perjelekkel. Tehát a helyes belépési pont mysite.wsgi:application. Jelenleg az EB úgy indítja, mintha mysite/wsgi lenne a modul — ezért nem találja, és 502 jön.

persze ezek a lépések kihagyhatóak, de véleményem szerint jól szemléltette az egyszerű hibakezelést és tesztelést

39. lépés: a requirements.txt file kiegészítése a gunicorn>=20

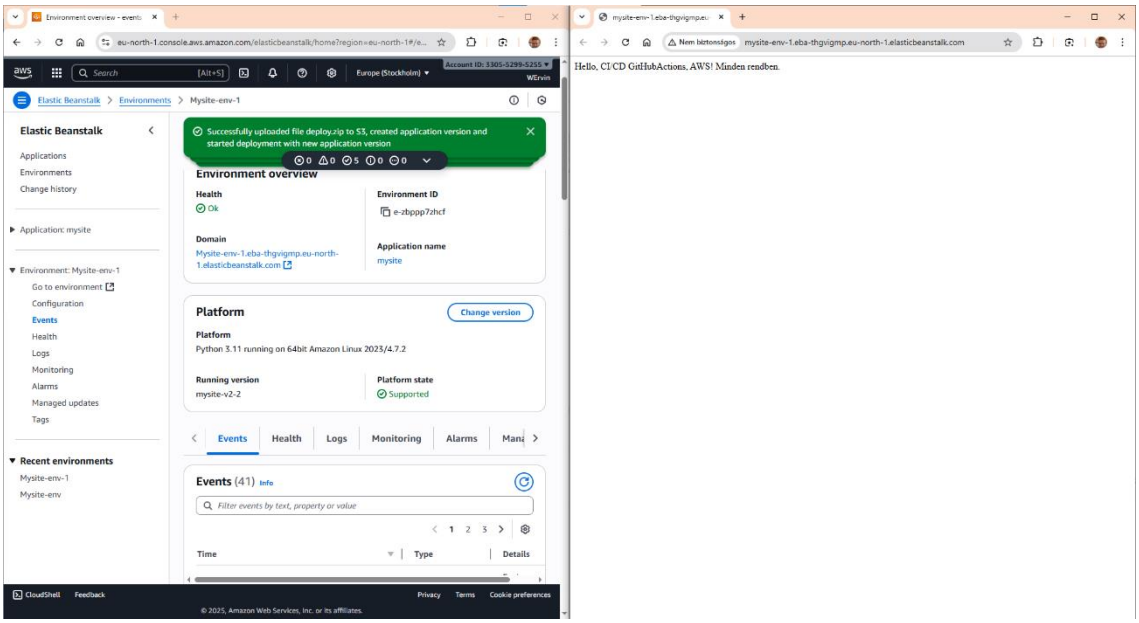


40. lépés: GitBashben a procfile létrehozása, majd commit és push



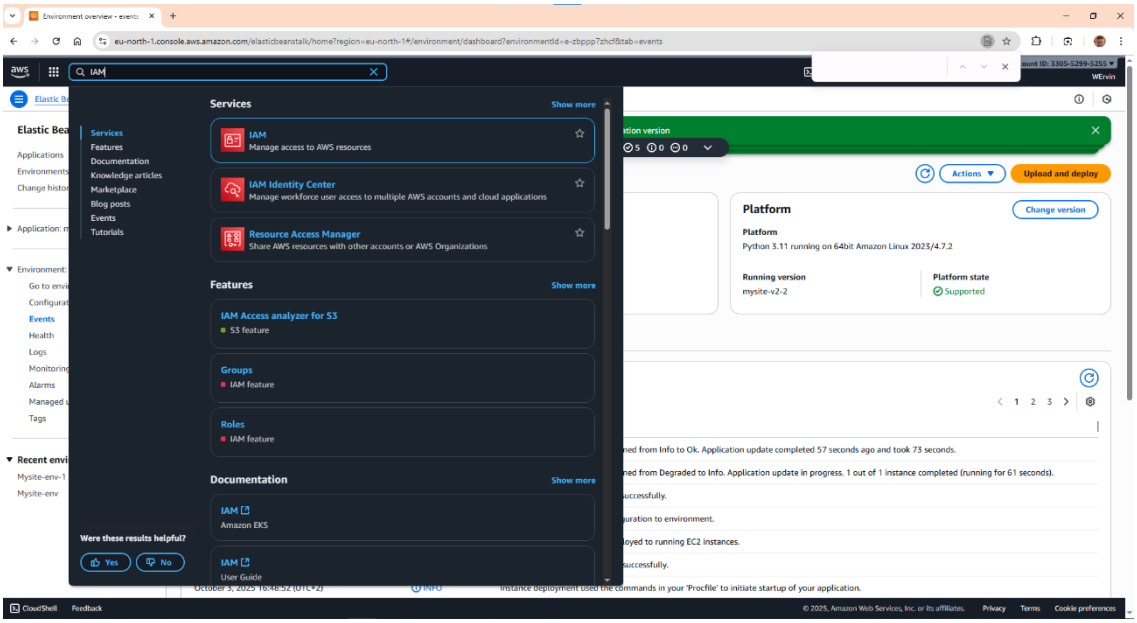
```
printf "web: gunicorn mysite.wsgi:application --bind 127.0.0.1:8000\n" > Procfile
git add Procfile hello/views.py mysite/urls.py requirements.txt
git commit -m "Add Procfile (Gunicorn) + root health endpoint"
git push origin main
```

41. lépés: Deploy ellenőrzés



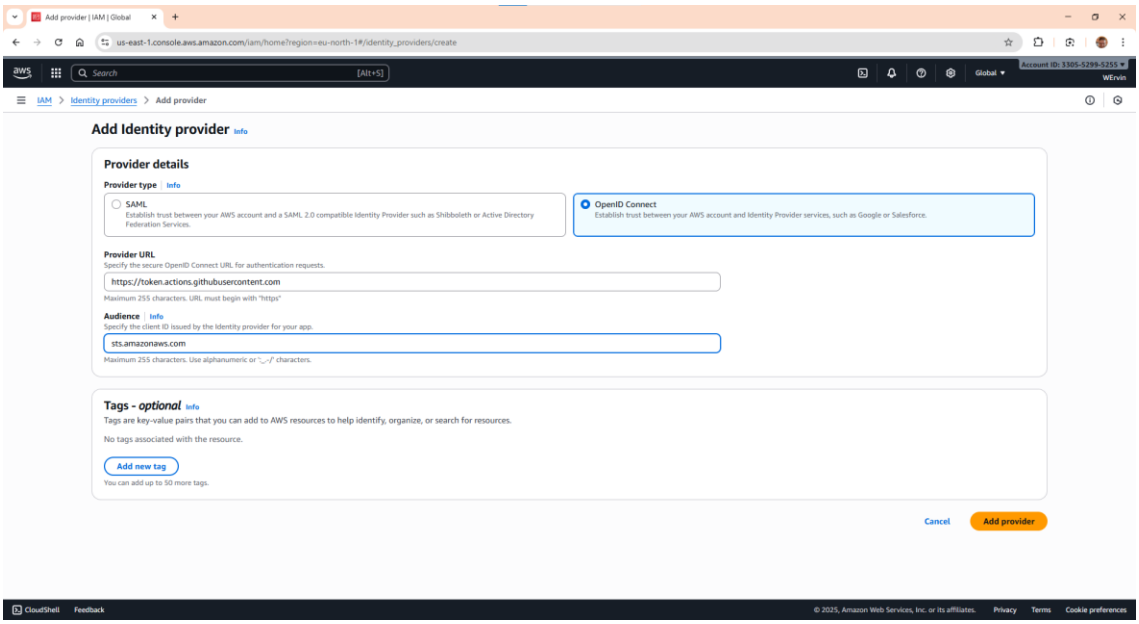
Sikeres deploy AWS Elastic Beanstalkon: a mysite-env-1 környezet állapota OK, futó verzió mysite-v2-2 (Python 3.11, AL2023). A jobb oldalon a domain 200 OK választ ad: ‘Hello, CI/CD GitHubActions, AWS! Minden rendben.’

42. lépés: IAM megnyitás



AWS konzol -> keresősávba: IAM(Identity and Access Management) AWS Management Console és bejelentkezés a felhasználóval

43. lépés: Identify provider hozzáadás



Identify provider -> Add provider

OPENID connect

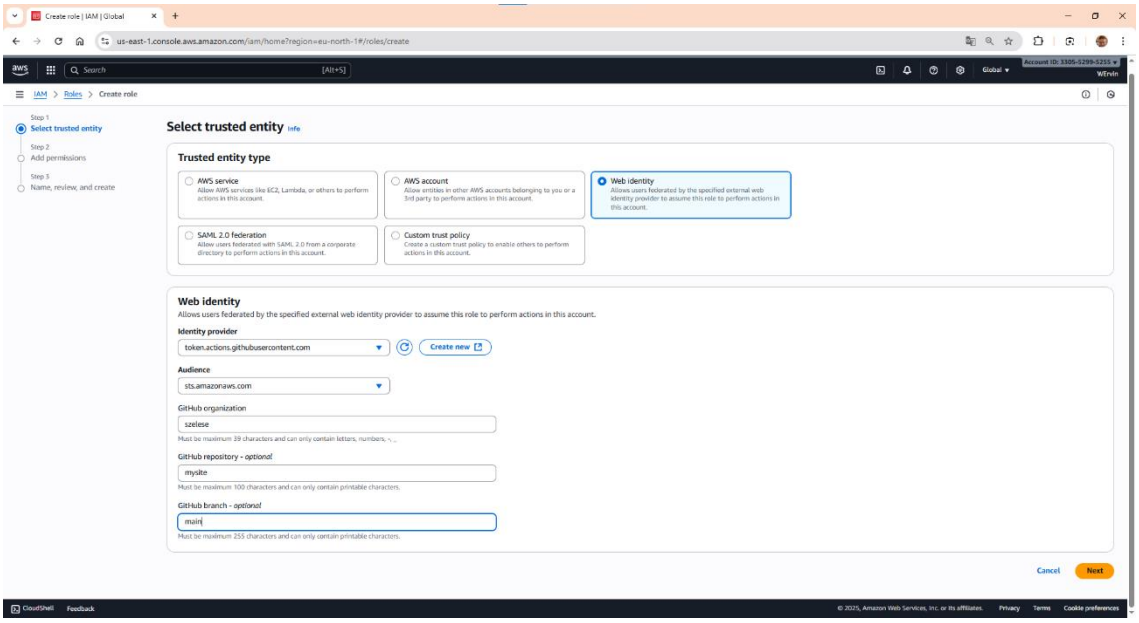
Provider URL:

https://token.actions.githubusercontent.com

Audience:

sts.amazonaws.com

44. lépés: Step1 beállítása



IAM/roles/create role:

WEB Identity

Identity provider: már jó, token.actions.githubusercontent.com

Audience: sts.amazonaws.com

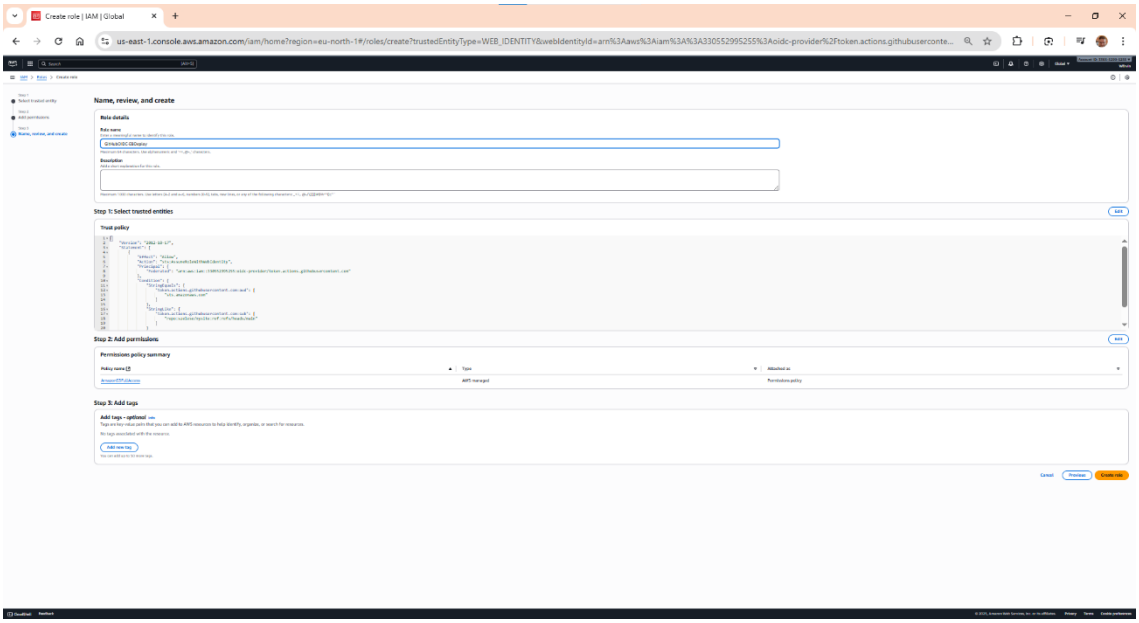
GitHub organization: szelese # saját github név

GitHub repository: mysite # repó név, ahonnan deployolni akarunk

GitHub branch: main # legtöbbször

utolsó 2 opcionális, de biztonsági okokból jobb megadni

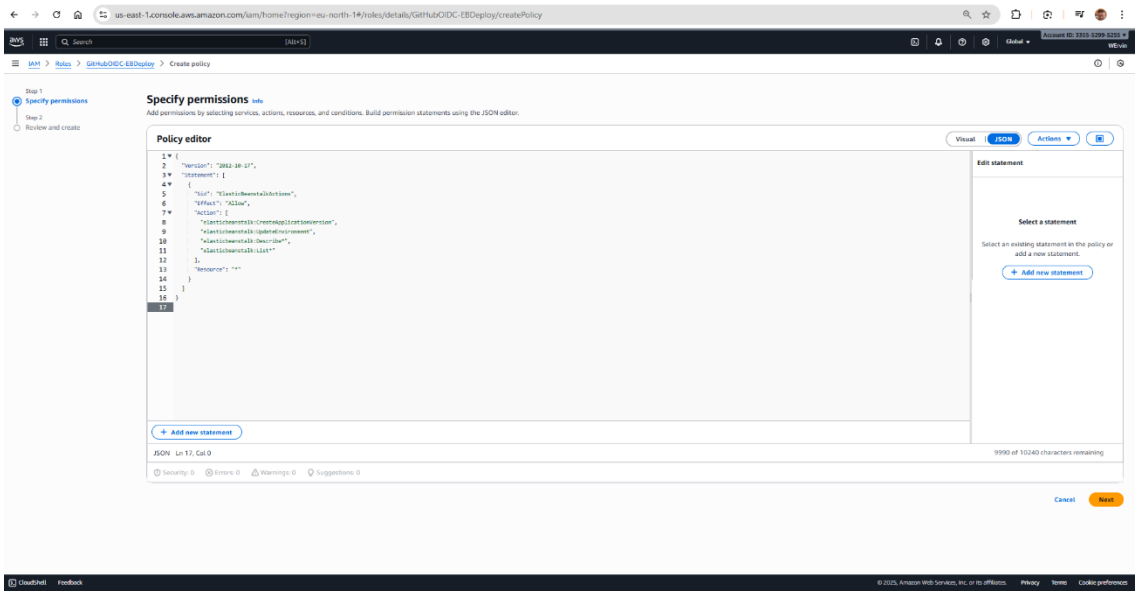
45. lépés: Step3



beírjuk a Role namebe: GitHubOIDC-EBDeploy

Create role

46. lépés: Permissions beállítása



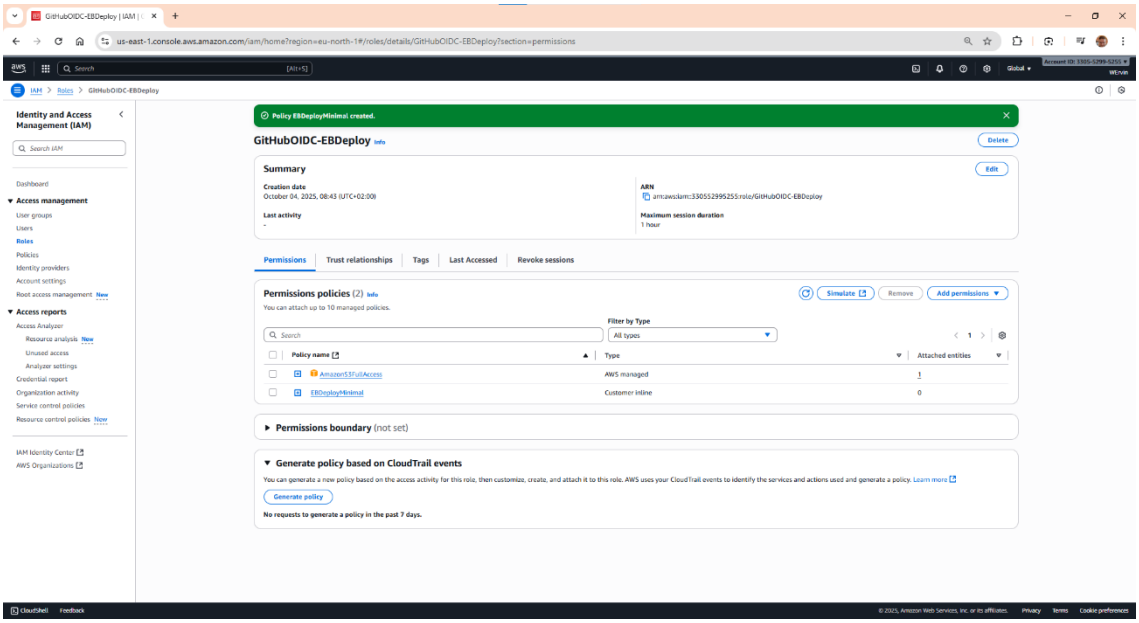
IAM->Roles->GitHubOIDC-EBDeploy->megnyitás

Permissions->Add permissions->Create inline policy->JSON

Policy Editorba beírni ami a képen van->NEXT

Név: EBDeployMinimal -> Create policy

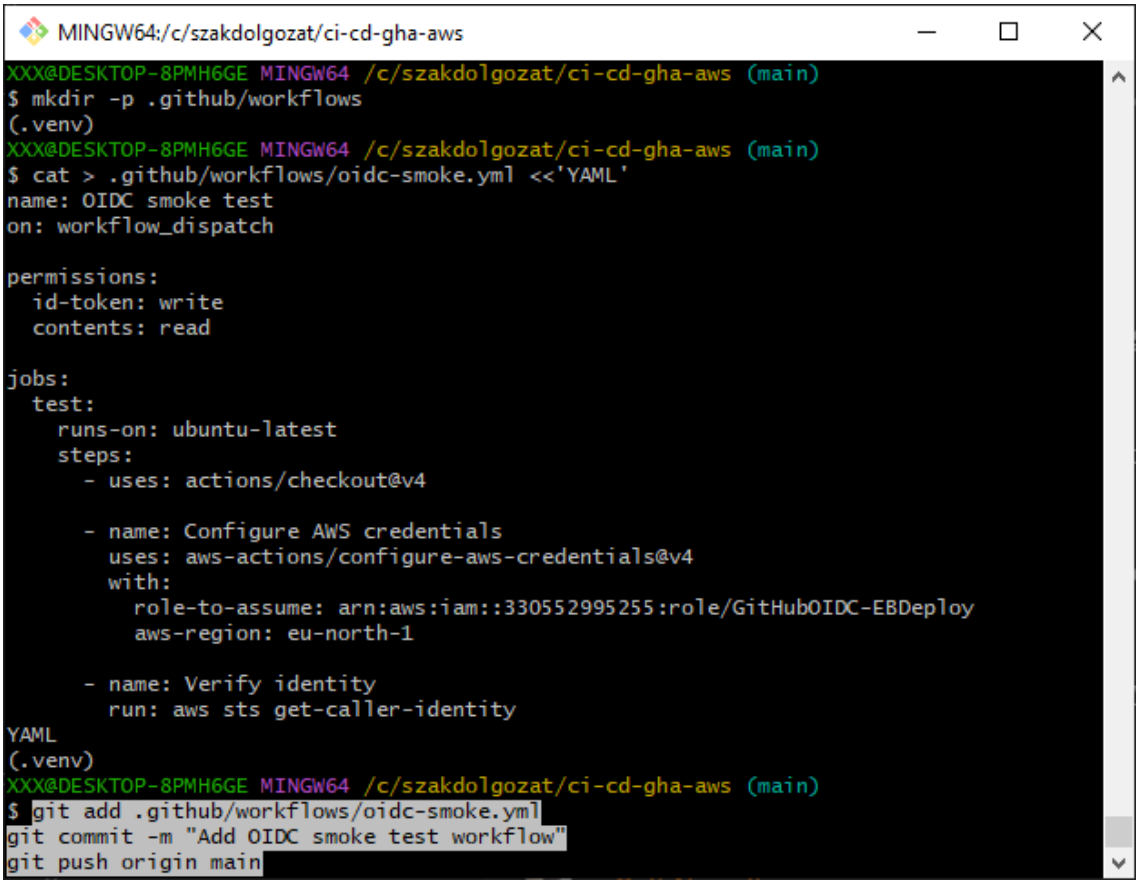
47. lépés: Policy ellenőrzés és ARN másolás



Permission policies-nél az előbb létrehozott két policyt kell látni

ARN-t kimásolni : arn:aws:iam::330552995255:role/GitHubOIDC-EBDeploy

48. lépés: GitBashon és commit&push

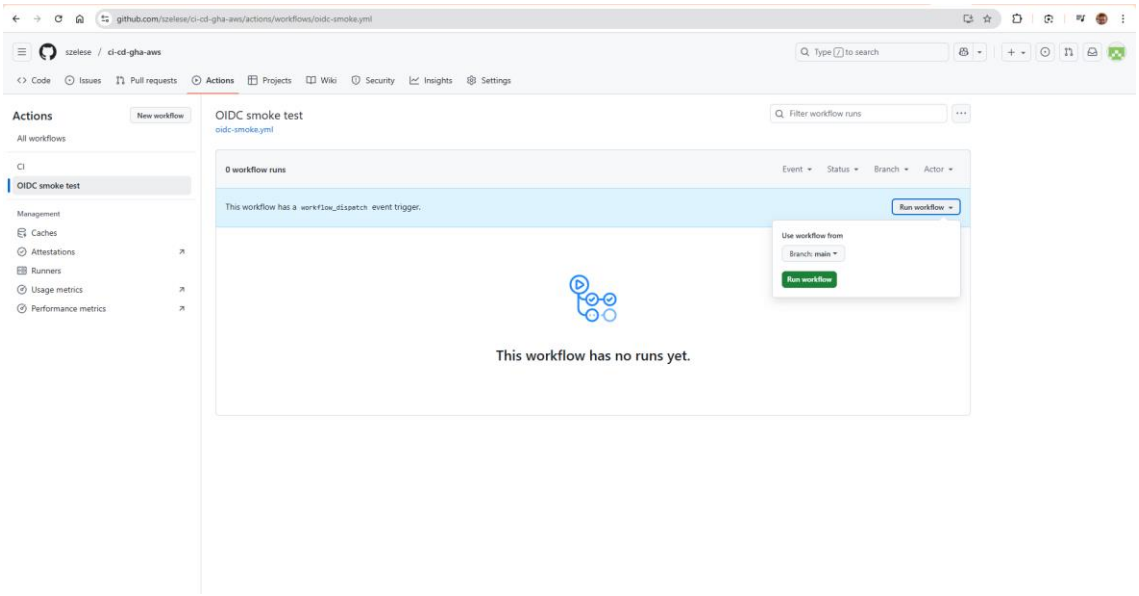


```
mkdir -p .github/workflows
```

file létrehozás a megadott tartalommal

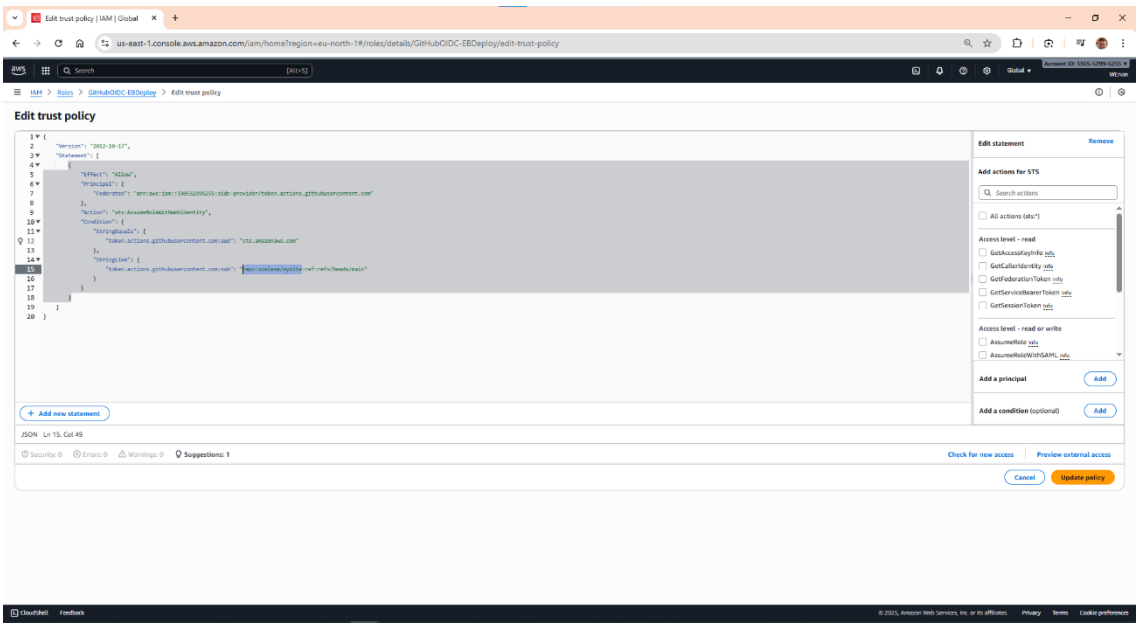
```
# commit&push
git add .github/workflows/oidc-smoke.yml
git commit -m "Add OIDC smoke test workflow"
git push origin main
```

49. lépés: Smoke test futtatása



GitHub -> Actions ->OIDC smoke test -> Branch -> Run workflow

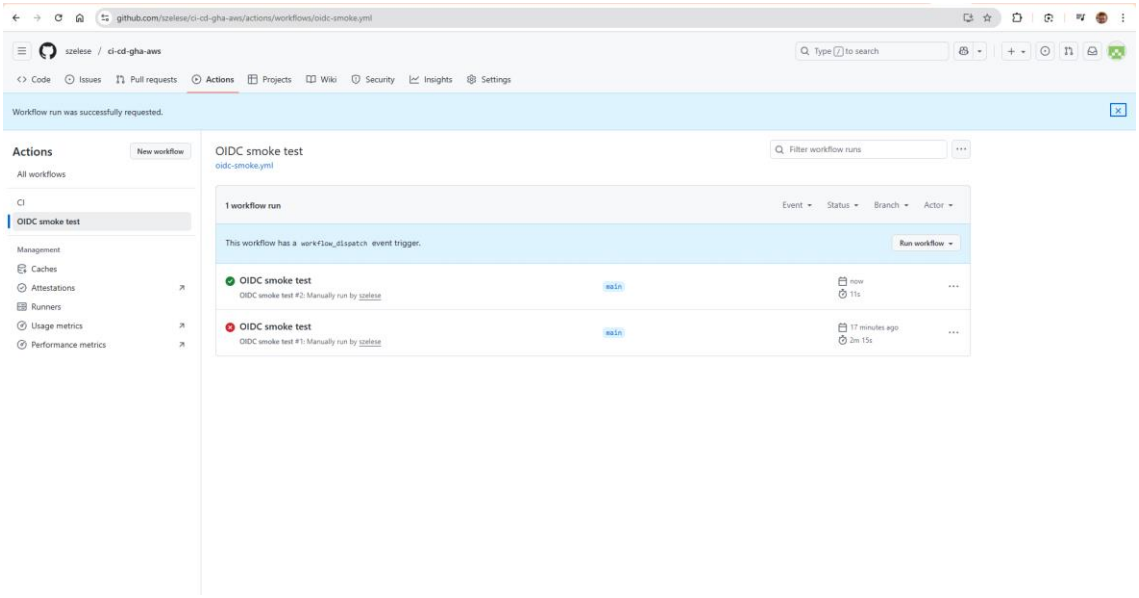
50. lépés: ellenőrzés és hibakeresés



rossz helyre mutat az alapbeállított repo:

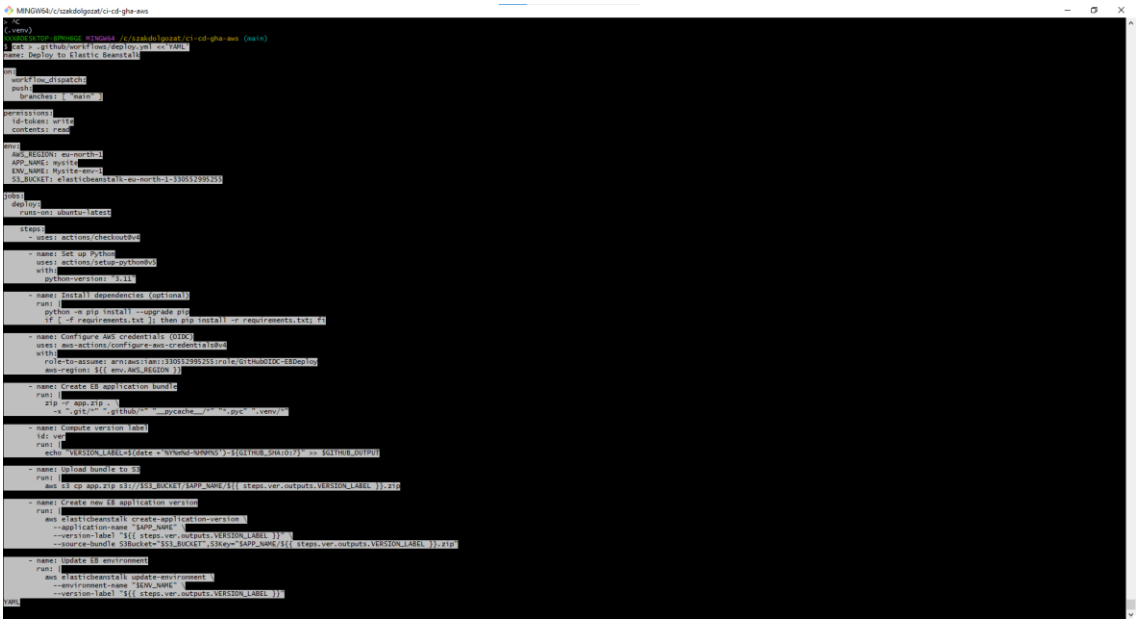
valódi repónk: szelese/ci-cd-gha-aws , ezért nem fut jól a smoke test, javítás!

51. lépés: Smoke test újrafuttatása

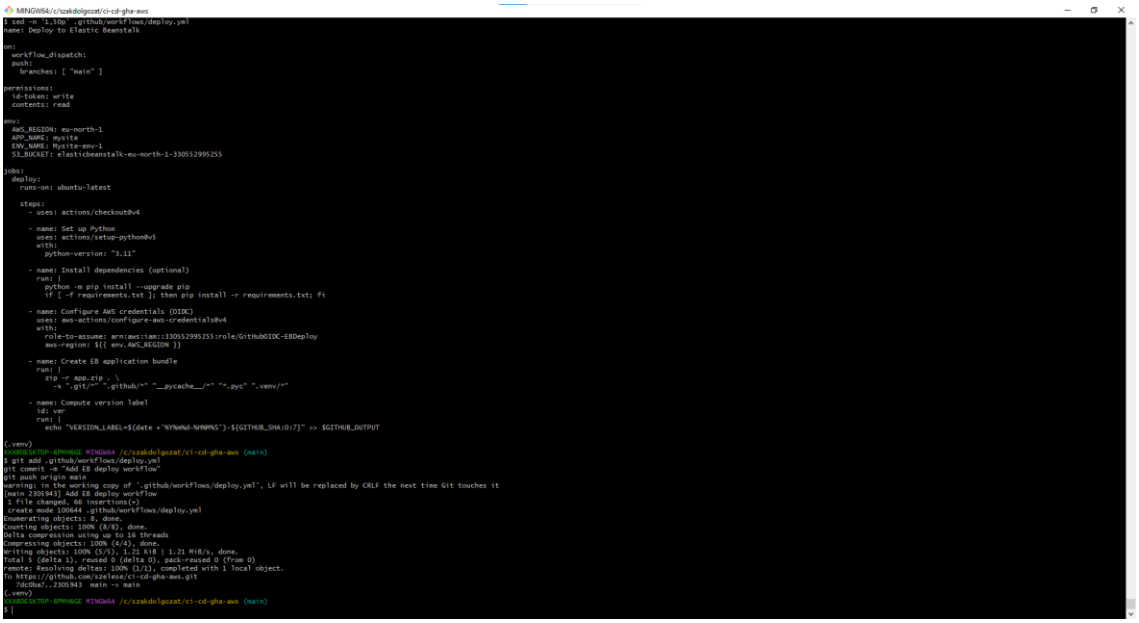


Sikeres OIDC smoke test GitHub Actions és AWS IAM role között — a kapcsolat hitelesen működik (zöld pipával jelzett futás).

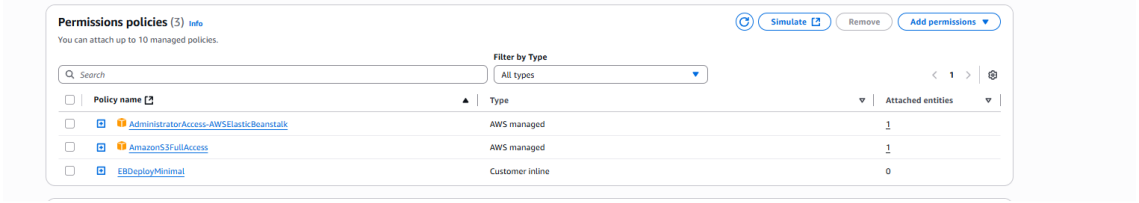
52. lépés: a deploy file létrehozása



53. lépés: ellenőrzés, commit&push



54. lépés: manuális futtatás hibát dobott, hibakeresés után még egy permissiont hozzáadása



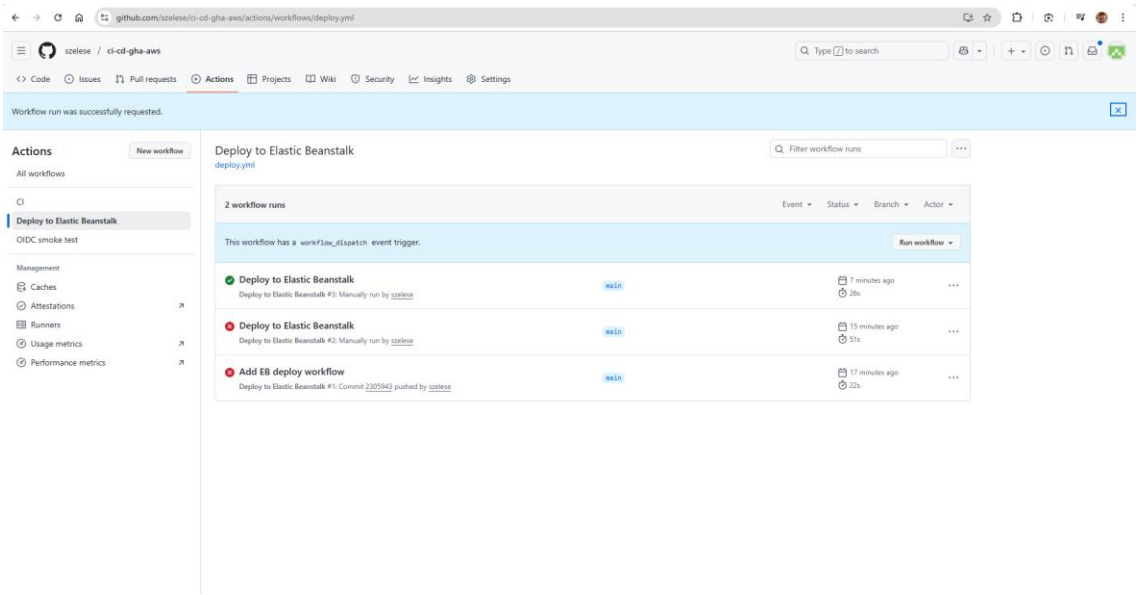
AWS Console → IAM → Roles → GitHubOIDC-EBDeploy.

Permissions fül → Add permissions → Attach policies.

AdministratorAccess-AWSElasticBeanstalk

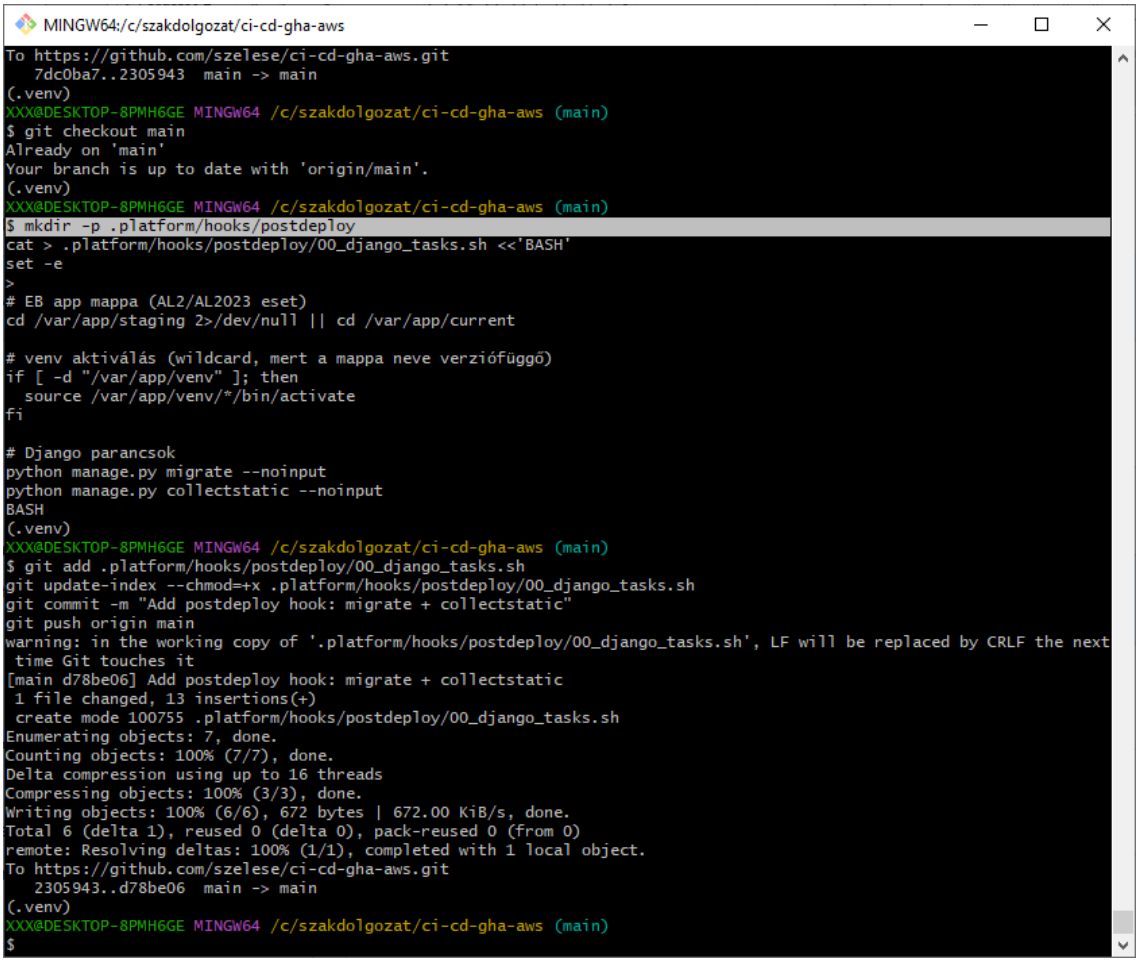
Add permissions

55. lépés: GitHub Actions -> Re-run jobs



A képen látszik, hogy a Deploy to Elastic Beanstalk legutóbbi futás zöld - tehát a CI→CD lánc működik.

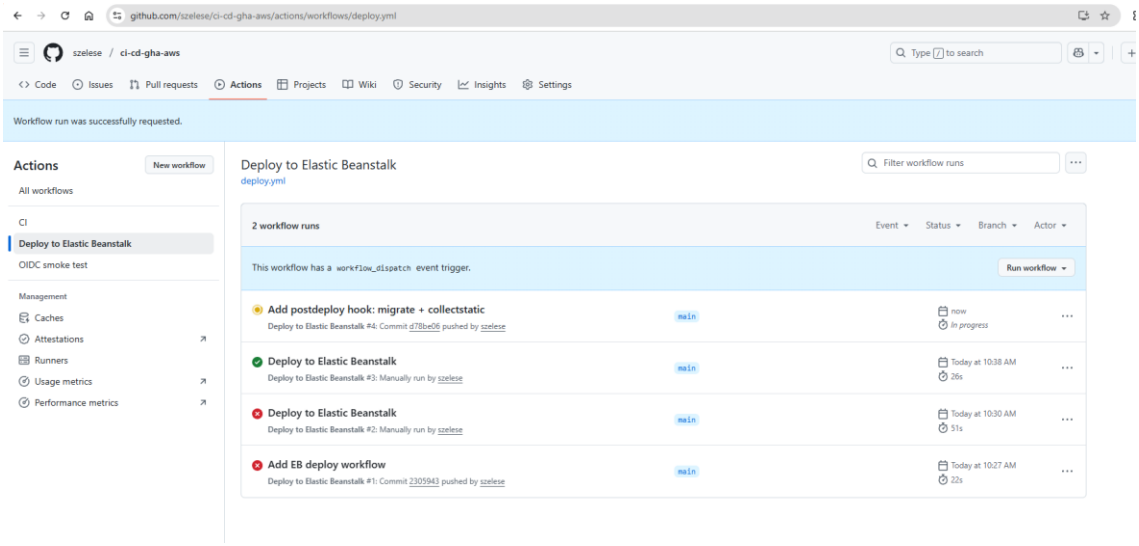
56. lépés: post-deploy hook (migrate + collectstatic)



Nem kötelező a működéshez, de erősen ajánlott: migrációk, statikus fájlok, titkok a szerveren vannak, ismételt automatizált

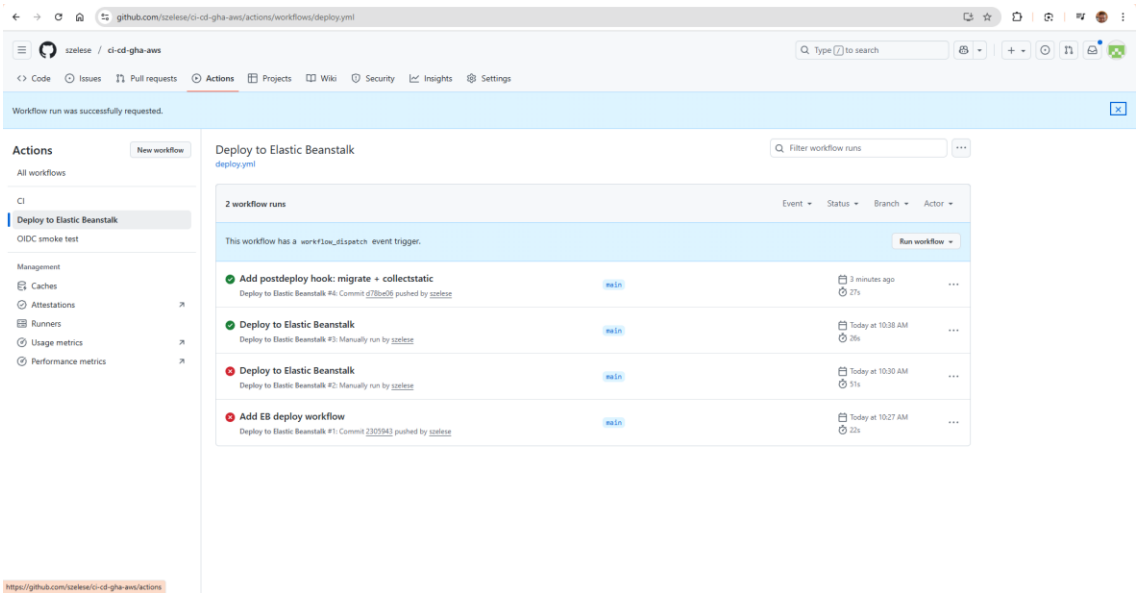
```
mkdir -p .platform/hooks/postdeploy
cat > .platform/hooks/postdeploy/00_django_tasks.sh <<'BASH'
git add .platform/hooks/postdeploy/00_django_tasks.sh
git update-index --chmod=+x .platform/hooks/postdeploy/00_django_tasks.sh
git commit -m "Add postdeploy hook: migrate + collectstatic"
git push origin main
```

57. lépés: GitHub Actions

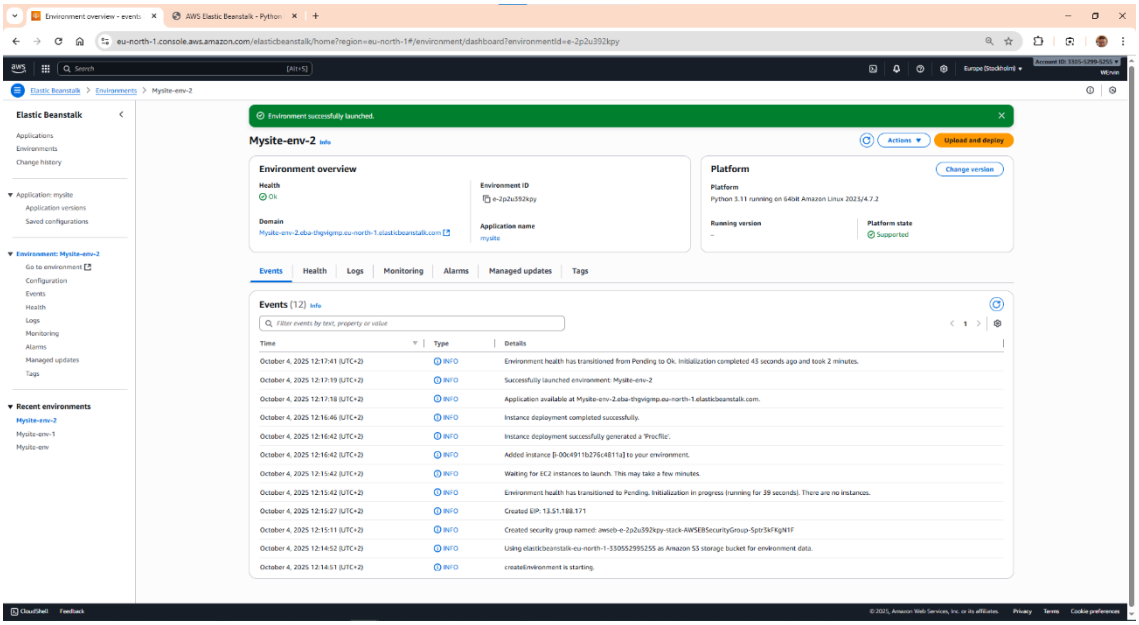


push&commit után látható, hogy automatikusan elkezdi futtatni a GitHub Actions

58. lépés: a futás sikeres volt a post-deploy hook aktív

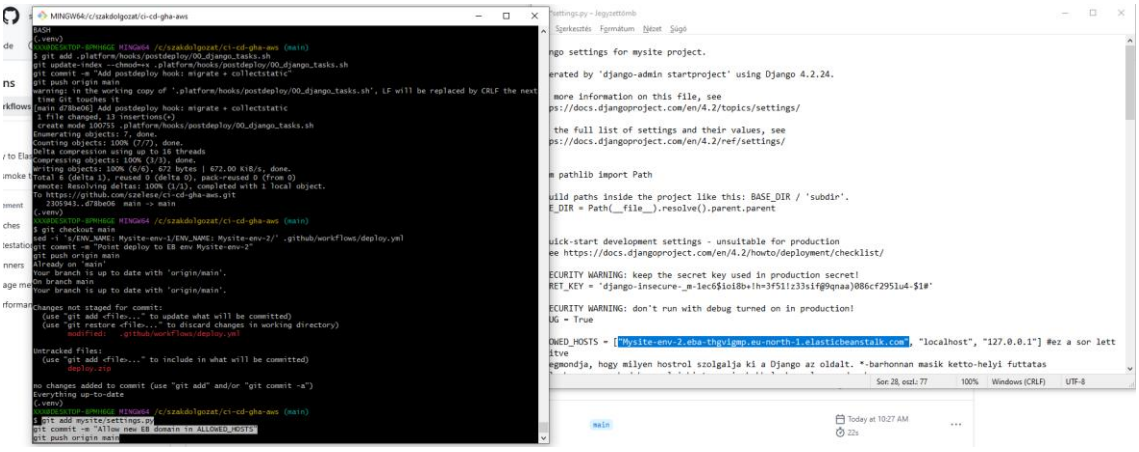


59. lépés: egy új szerver létrehozása: mysite-env-2



#biztonsági okokból a mysite-env-1 -et később Suspended-felfüggesztett állapotba tettem

60. lépés: a deploy workflow új enviromentre állítom



settings.py filet módosítom, hogy az allowed host csak a megadott lehessen:

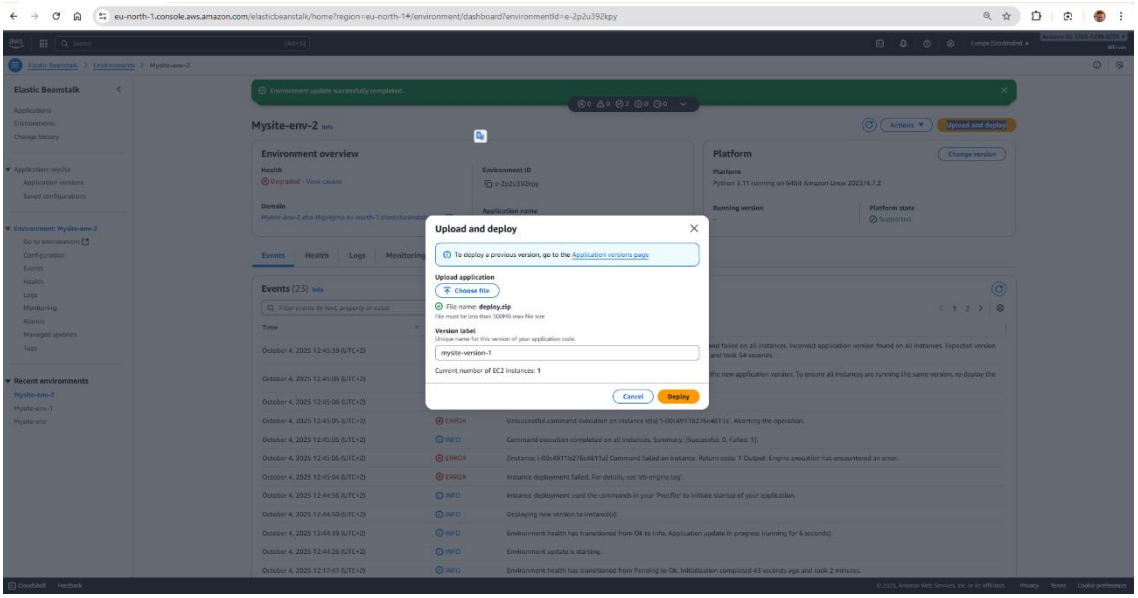
Mysite-env-2.eba-thgvigmp.eu-north-1.elasticbeanstalk.com # biztonság

commit&push

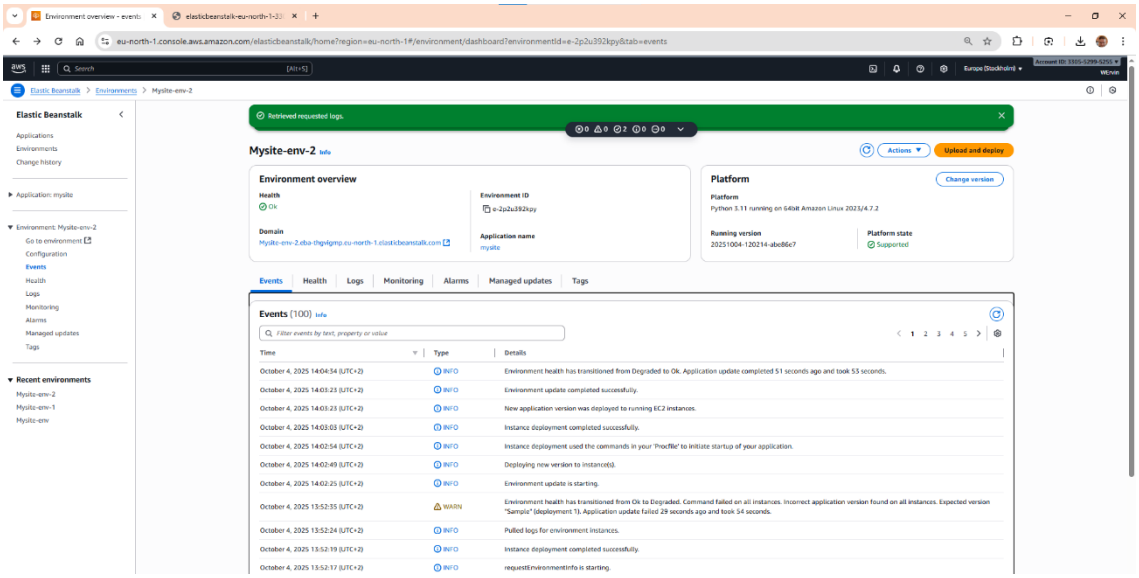
61. lépés: lokálisan már a Mysite-env-2, de a repóban még az 1-es ezért commit&push módosítás


```
git add .github/workflows/deploy.yml
git commit -m "Switch ENV_NAME to Mysite-env-2"
git push origin main
```

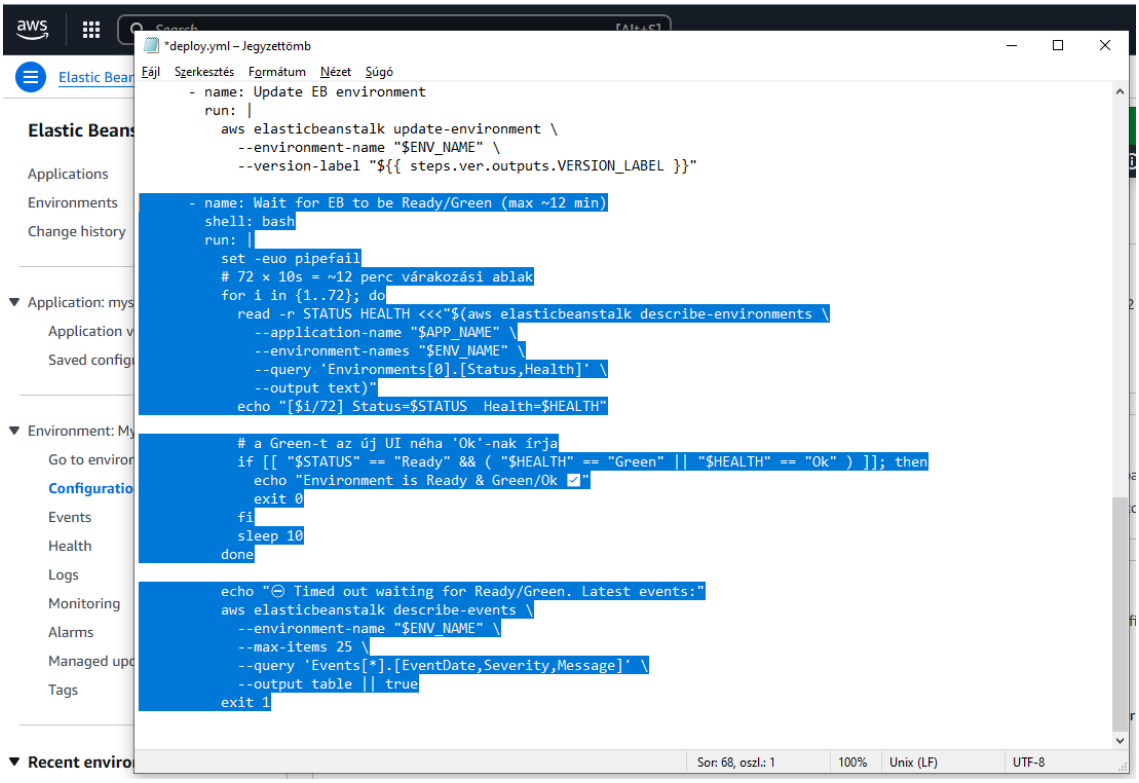
62. lépés: soha többet nem kell manuálisan deployolni, mint a képen



63. lépés: az új szerver environment felállt, stabil készen áll a használatra



64. lépés: Hibakeresés és előfeltétel



Az alábbiakkal azt érjük el, hogy a deploy workflow csak akkor fejeződjön be sikeresen, ha az EB környezet Ready + Green/Ok állapotba kerül. Ha nem, a workflow pirossal leáll, és kiírja az utolsó EB eseményeket. egyszerű hibakeresés és előfeltétel, hogy a szerver csak akkor álljon fel, ha a rendszer stabil.

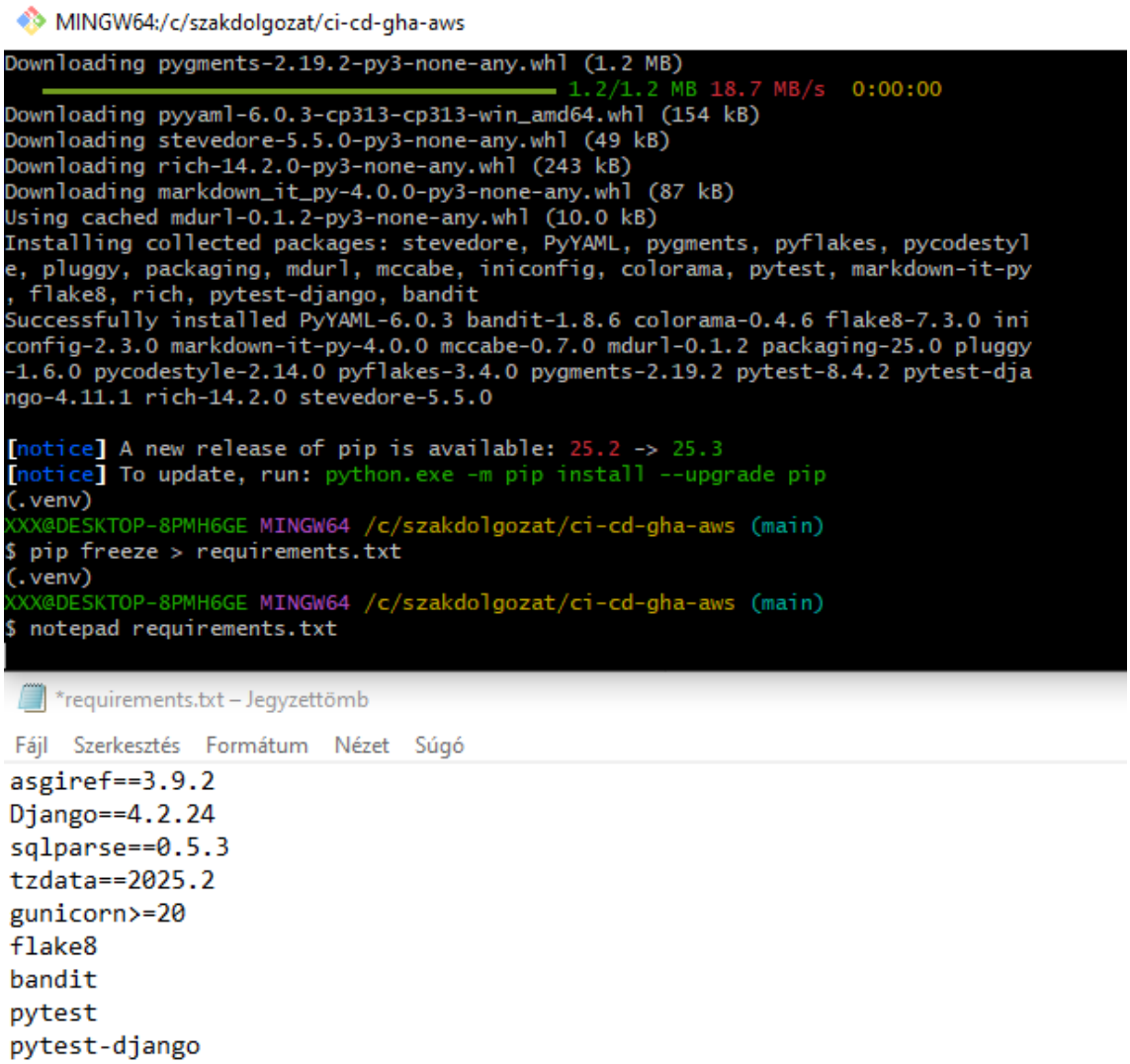
GitBash-ban utána commit&push:

```
git checkout main
git add .github/workflows/deploy.yml
git commit -m "Deploy: wait until EB is Ready & Green"
git push origin main
```

65. lépés: CI bővítés: flake8, bandit, pytest

A CI pipeline megerősítéséhez telepítjük a minőségbiztosításhoz szükséges eszközöket: flake8 (kódstílus ellenőrzés), bandit (biztonsági rések keresése) és pytest-django (unit tesztek futtatása). Ezeket hozzáadjuk a requirements.txt fájlhoz, hogy a GitHub Actions futtatója is telepíteni tudja őket.

pip freeze > requirements.txt parancs minden telepített csomagot beleírt, nem ideális. fölösleges környezetnagyítás és nehezebb később frissíteni, ezért manuálisan fűzöm hozzá a requirements.txt végére a most telepített függőségeket



```
pip install flake8 bandit pytest pytest-django
notepad requirements.txt #manuálisan beírni
# pip freeze > requirements.txt
```

66. lépés: alap unit teszt létrehozása

Létrehozzuk az első unit tesztet a hello/tests.py fájlban. Ez a teszt azt ellenőrzi, hogy az alkalmazás főoldala (index nézet) sikeresen betöltődik-e (200 OK státuszkód).

```
XXX@DESKTOP-8PMH6GE MINGW64 /c/szakdolgozat/ci-cd-gha-aws (main)
$ notepad hello/tests.py
```

*tests.py – Jegyzettömb

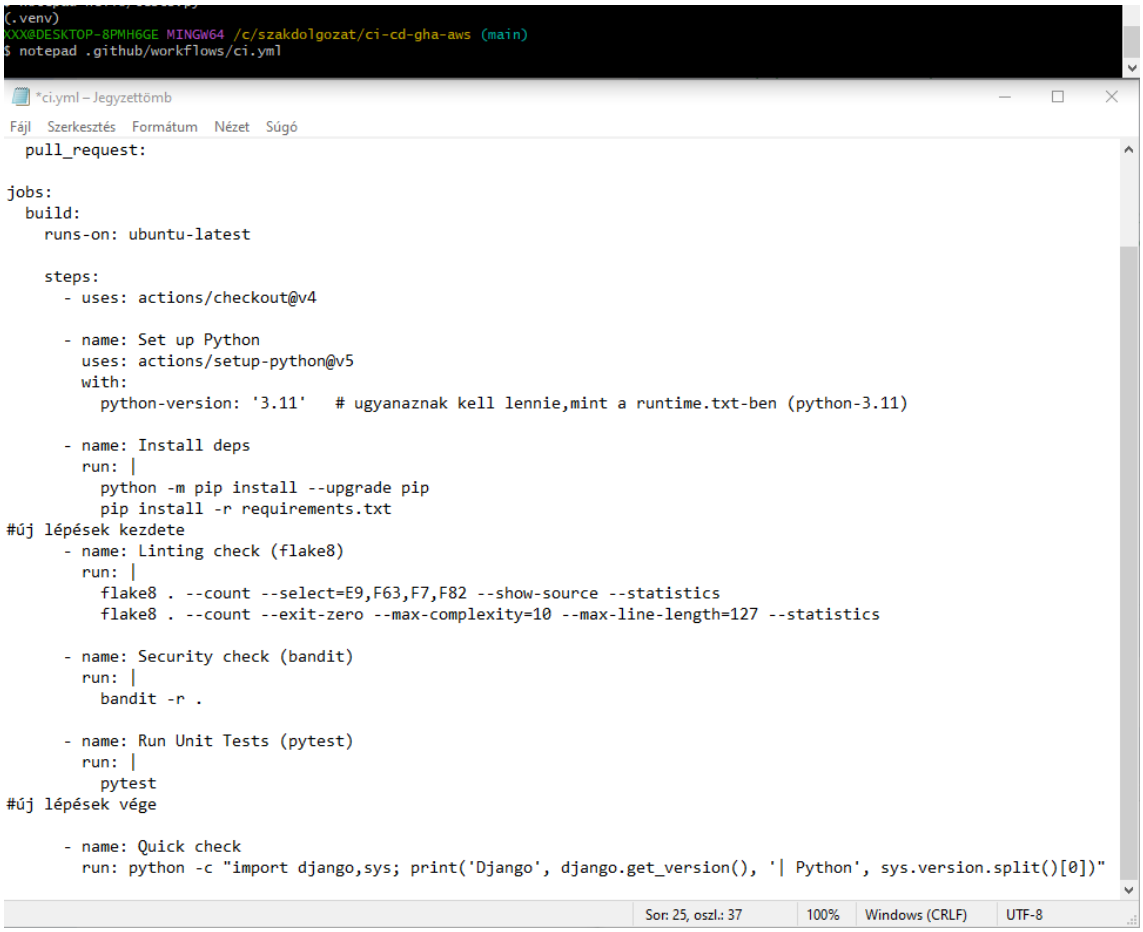
Fájl Szerkesztés Formátum Nézet Súgó

```
from django.test import TestCase
from django.urls import reverse

class HelloViewTests(TestCase):
    def test_index_view_status_code(self):
        """
        Teszteli, hogy a 'hello.urls'-ben 'index'-nek nevezett
        főoldali nézet 200-as (OK) státuszkódot ad-e vissza.
        """
        url = reverse('index')
        response = self.client.get(url)
        self.assertEqual(response.status_code, 200)
```

notepad hello/tests.py #manuálisan beírni a megadott kódot

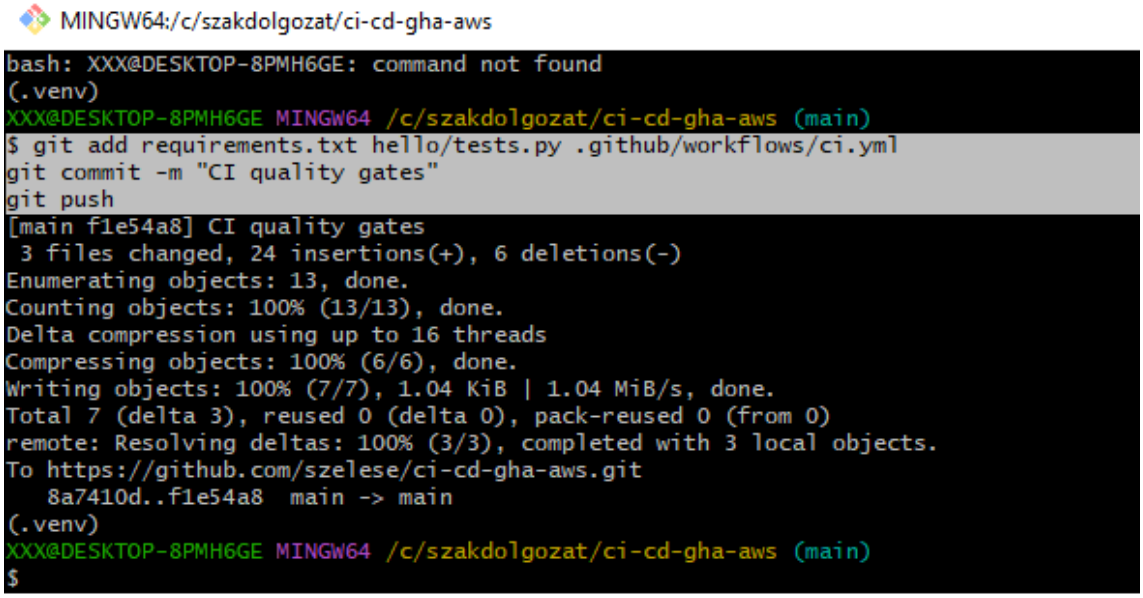
67. lépés: a ci.yml workflow kibővítése

Módosítjuk a .github/workflows/ci.yml fájlt. Az eddigi 'Quick check' elé beillesztünk három új lépést: 'Linting check (flake8)', 'Security check (bandit)' és 'Run Unit Tests (pytest)'. Ezek a lépések minőségi kapuként (quality gates) fognak működni: ha bármelyik hibát talál, a pipeline leáll, és megakadályozza a hibás kód bekerülését. Ezzel biztosítottuk a pipeline CI részét.



notepad .github/workflows/ci.yml

68. lépés: commit&push



```
git add requirements.txt hello/tests.py .github/workflows/ci.yml
git commit -m "CI quality gates"
git push
```

69.lépés: A kibővített CI pipeline futásának ellenőrzése

A módosított fájlok (requirements.txt, hello/tests.py, ci.yml) git commit és git push parancsai után a GitHub Actions automatikusan elindítja a kibővített CI pipeline-t. A futás sikertelen. Security check hiba

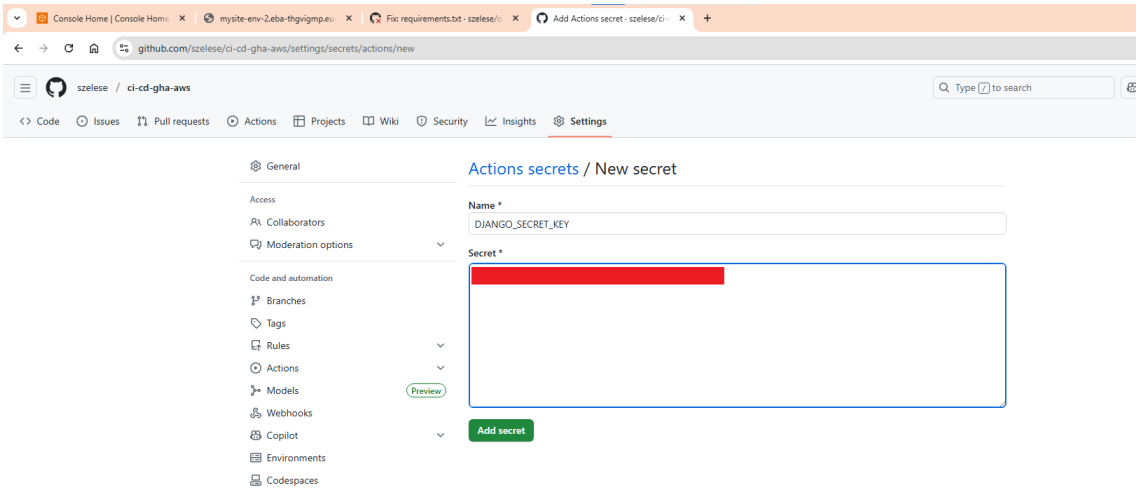
70.lépés: mysite/settings.py módosítása

```
XXX@DESKTOP-8PMH6GE MINGW64 /c/szakdolgozat/ci-cd-gha-aws (main)
$ notepad mysite/settings.py
```

```
Generated by 'django-admin startproject' using Django 4.2.24.

For more information on this file, see
https://docs.djangoproject.com/en/4.2/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/4.2/ref/settings/
"""

from pathlib import Path
import os
from django.core.management.utils import get_random_secret_key

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent


# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/4.2/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = os.environ.get("DJANGO_SECRET_KEY", get_random_secret_key())
```

a statikus kulcsot kicseréljük: ha létezik DJANGO_SECRET_KEY nevű környezeti változó, azt fogja használni, egyébként generál egy véletlen kulcsot

71. lépés: GitHub-repo secret beállítása



Settings/Secrets and Variables/Actions/New repository secret

Name: DJANGO_SECRET_KEY

Secret: **********

72. lépés: Workflow file frissítése

```
MINGW64:/c/szakdolgozat/ci-cd-gha-aws
XXX@DESKTOP-8PMH6GE MINGW64 /c/szakdolgozat/ci-cd-gha-aws (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
(.venv)
XXX@DESKTOP-8PMH6GE MINGW64 /c/szakdolgozat/ci-cd-gha-aws (main)
$ notepad mysite/settings.py
(.venv)
XXX@DESKTOP-8PMH6GE MINGW64 /c/szakdolgozat/ci-cd-gha-aws (main)
$ python - <<'PY'
from django.core.management.utils import get_random_secret_key
print(get_random_secret_key())
PY
=4!-p-psx^rde=sj!@o_zkbr_mfwc57cuvv(ydmi!cp1rlhw!5
(.venv)
XXX@DESKTOP-8PMH6GE MINGW64 /c/szakdolgozat/ci-cd-gha-aws (main)
$ notepad .github/workflows/ci.yml
```

```
*ci.yml – Jegyzettömb
Fájl  Szerkesztés  Formátum  Nézet  Súgó
name: CI

on:
  push:
    branches: [ "main" ]
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest
    env:
      DJANGO_SECRET_KEY: ${{ secrets.DJANGO_SECRET_KEY }}
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
```

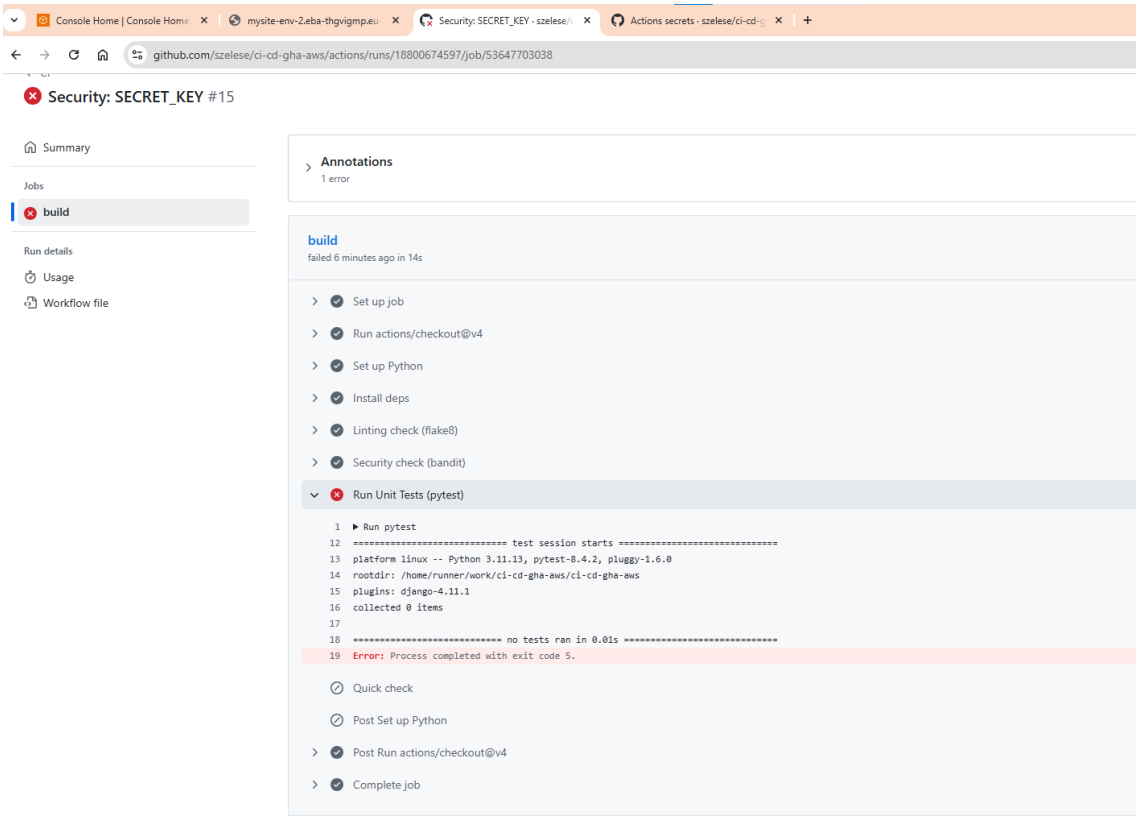
```
    env:
      DJANGO_SECRET_KEY: ${{ secrets.DJANGO_SECRET_KEY }}
```

73. lépés: commit&push

```
MINGW64:/c/szakdolgozat/ci-cd-gha-aws
XXX@DESKTOP-8PMH6GE MINGW64 /c/szakdolgozat/ci-cd-gha-aws (main)
$ notepad .github/workflows/ci.yml
(.venv)
XXX@DESKTOP-8PMH6GE MINGW64 /c/szakdolgozat/ci-cd-gha-aws (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   .github/workflows/ci.yml
        modified:   mysite/settings.py

no changes added to commit (use "git add" and/or "git commit -a")
(.venv)
XXX@DESKTOP-8PMH6GE MINGW64 /c/szakdolgozat/ci-cd-gha-aws (main)
$ git add mysite/settings.py .github/workflows/ci.yml
git commit -m "Security: SECRET_KEY"
git push
```

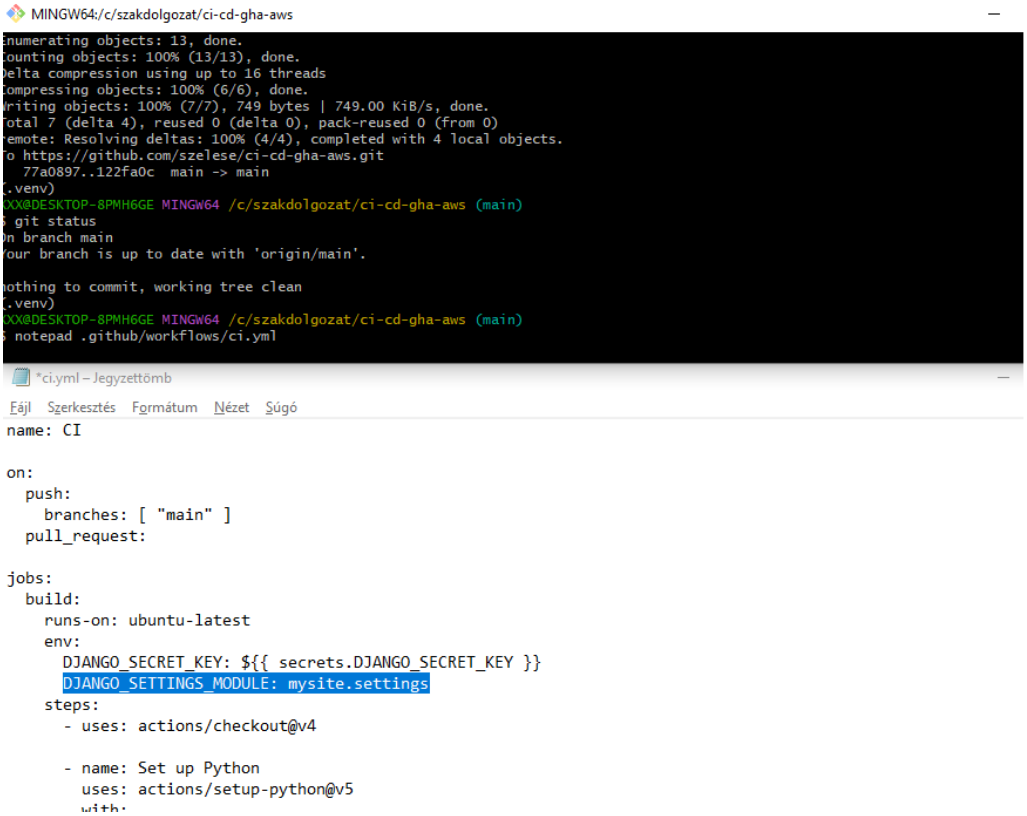
```
git add mysite/settings.py .github/workflows/ci.yml
git commit -m "Security: SECRET_KEY"
git push
```

74. lépés: Unit Tests, pytest beállítása:



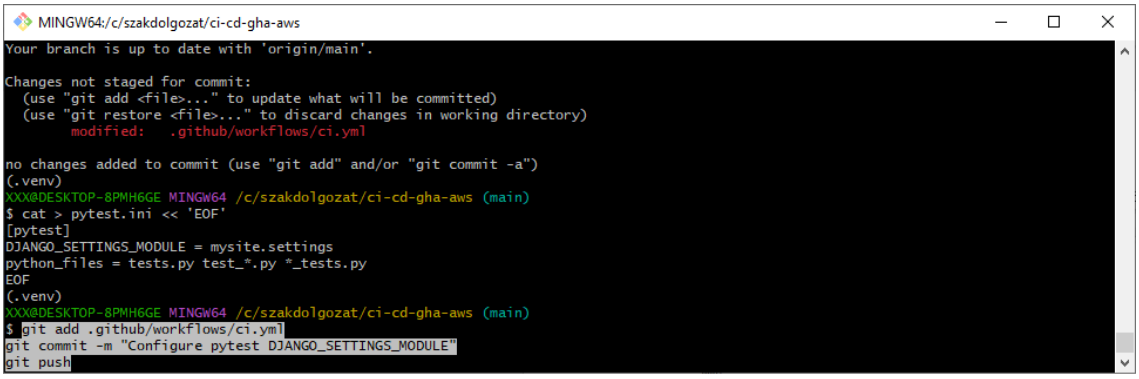
Nem talál futtatható teszteket, pedig van tesztosztály. De nem aktiválódik automatikusan, ezért manuálisan aktiváljuk.

75. lépés: DJANGO_SETTINGS_MODULE beállítása a CI-ben



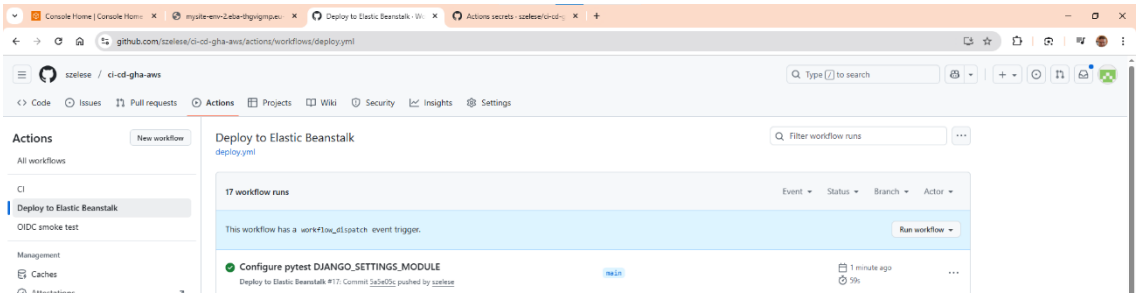

```
DJANGO_SETTINGS_MODULE: mysite.settings
...
      - name: Run Unit Tests (pytest)
        run: |
          pytest --ds=mysite.settings
```

76. lépés: pytest.ini file létrehozása, commit&push



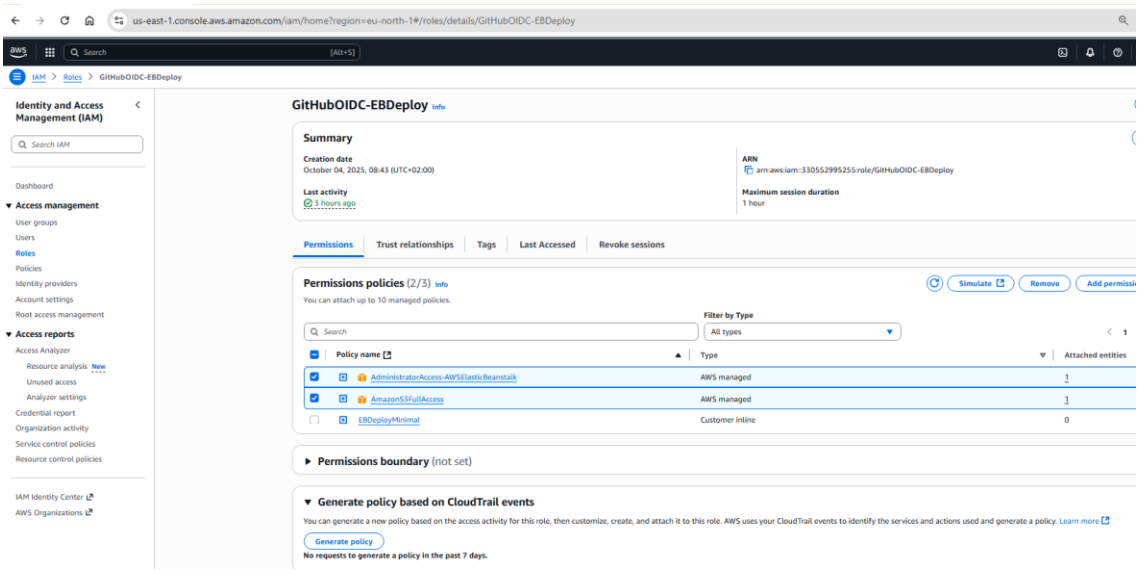

```
cat > pytest.ini << 'EOF'
[pytest]
DJANGO_SETTINGS_MODULE = mysite.settings
python_files = tests.py test_*.py *_tests.py
EOF
git add pytest.ini .github/workflows/ci.yml
git commit -m "Configure pytest DJANGO_SETTINGS_MODULE"
git push
```

77. lépés: A kibővített CI pipeline futásának ellenőrzése



A módosított fájlok (requirements.txt, hello/tests.py, ci.yml) git commit és git push parancsai után elindul a kibővített CI pipeline-t. A futás sikeres!

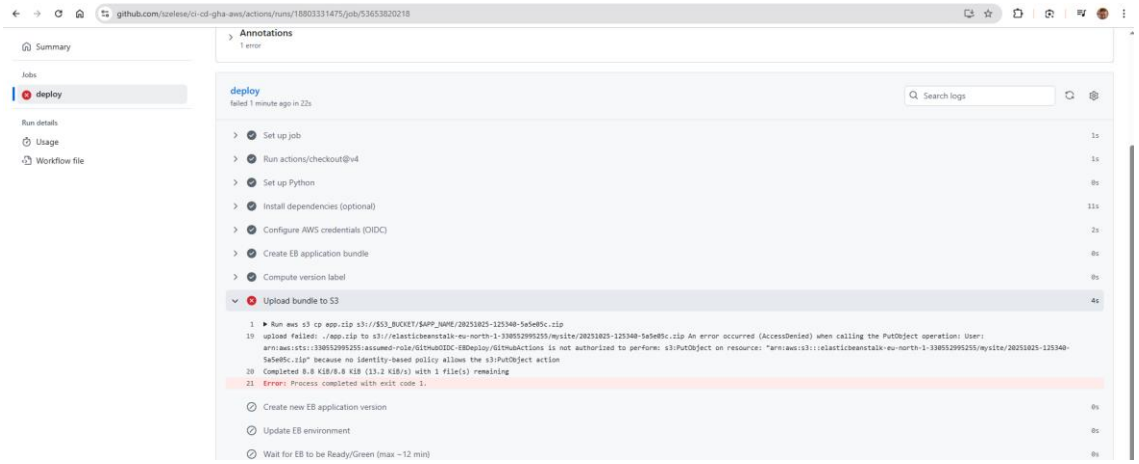
78. lépés: Admin Policy javítása



két „lusta” admin policy eltávolítása: AmazonS3FullAccess, AdministratorAccess-AWSElasticBeanstalk

AWS/IAM/Roles/GitHubOIDC-EBDeploy
Legkisebb jogosultság beállítása

79. lépés: iteratív hibakereső ciklus

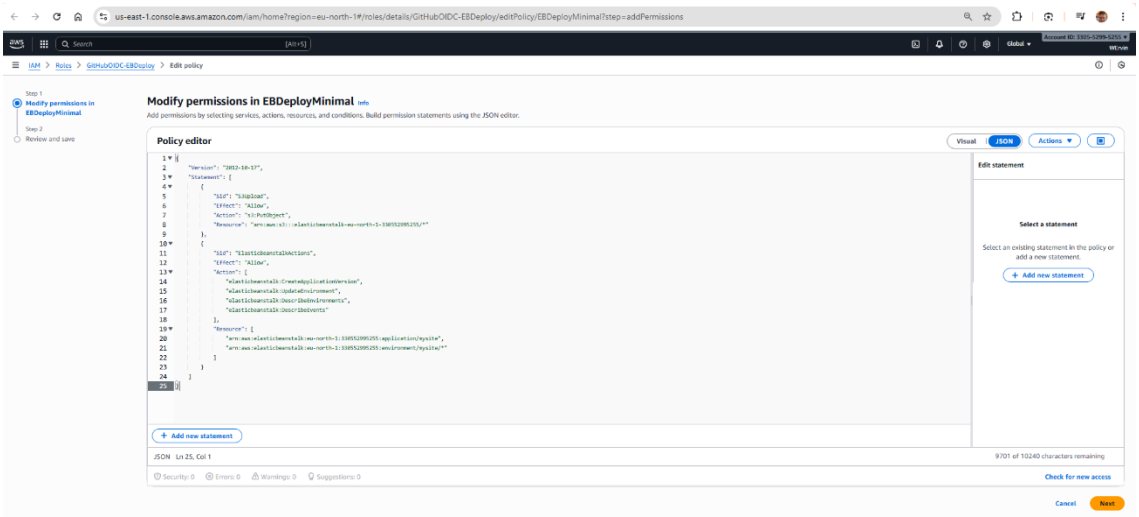


Most addig fogjuk futtatni a deploy.yml-t, amíg ki nem írja minden egyes hiányzó jogosultságot.

Addig ismételjük ezt a fázist, ameddig minden egyes hiányzó jogosultságot hozzá nem adunk.

Lehet látni, hogy az s3:PutObject (fájlfeltöltés) hiányzik.

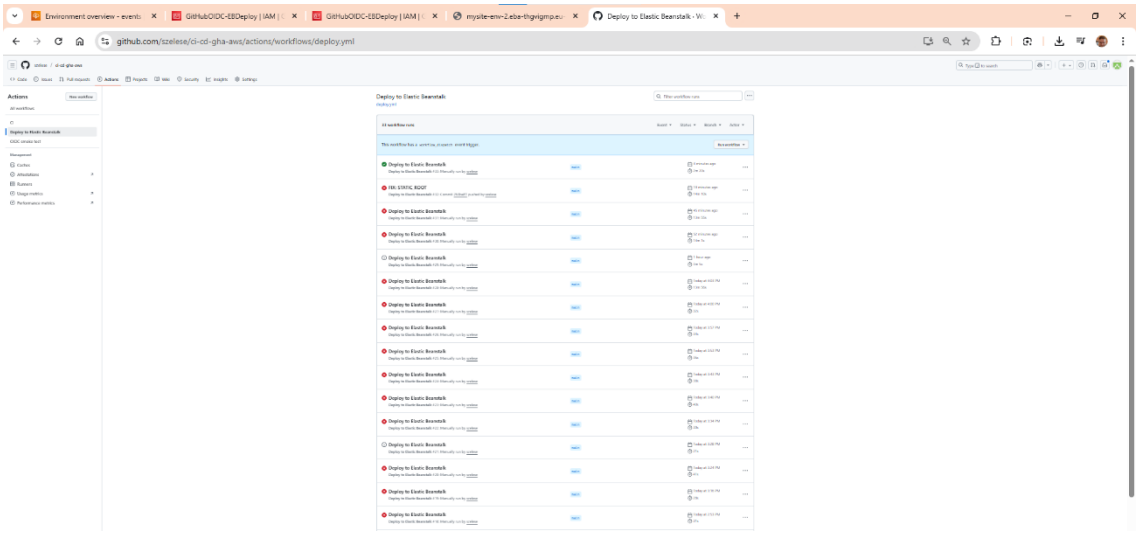
80. lépés: Policy módosítás



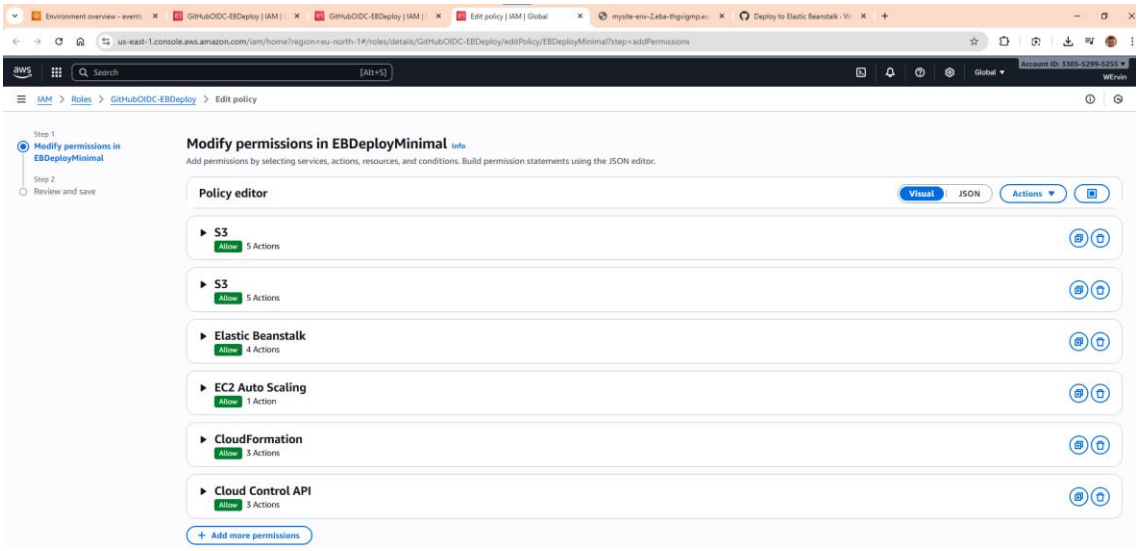
elmentjük és utána GitHub Actions/Deploy to Elastic Beanstalk/run workflow

ezzel egyesével adogatjuk hozzá a szükséges jogosultságokat, ezáltal biztosítjuk, ha valami még hiányzik visszaugrunk 1 lépést, ha minden jó, akkor 81. lépés jöhet

81. lépés: a 15. iteráció után sikeres deploy



82. lépés: a helyes policy beállítása



JSON:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "S3ObjectAccess",
         "Effect": "Allow",
         "Action": [
            "s3:PutObject",
            "s3:GetObject",
            "s3:GetObjectAcl",
            "s3:PutObjectAcl",
            "s3:DeleteObject"
         ],
         "Resource": "arn:aws:s3:::elasticbeanstalk-eu-north-1-330552995255/*"
      },
      {
         "Sid": "S3BucketAccess",
         "Effect": "Allow",
         "Action": [
            "s3:CreateBucket",
            "s3:PutBucketPolicy",
            "s3:PutBucketOwnershipControls",
            "s3:ListBucket",
            "s3:GetBucketPolicy"
         ],
         "Resource": "arn:aws:s3:::elasticbeanstalk-eu-north-1-330552995255"
      },
      {
         "Sid": "ElasticBeanstalkCloudformationAutoscalingAccess",
         "Effect": "Allow",
         "Action": [
            "elasticbeanstalk:CreateApplicationVersion",
```

```
            "elasticbeanstalk:UpdateEnvironment",
            "elasticbeanstalk:DescribeEnvironments",
            "elasticbeanstalk:DescribeEvents",
            "cloudformation:GetTemplate",
            "cloudformation:DescribeStackResource",
            "cloudformation:DescribeStackResources",
            "autoscaling:DescribeAutoScalingGroups"
         ],
         "Resource": [
            "arn:aws:elasticbeanstalk:eu-north-1:330552995255:application/mysite",
            "arn:aws:elasticbeanstalk:eu-north-1:330552995255:environment/mysite/*",
            "arn:aws:elasticbeanstalk:eu-north-
1:330552995255:applicationversion/mysite/*",
            "arn:aws:cloudformation:eu-north-1:330552995255:stack/awseb-*",
            "arn:aws:autoscaling:eu-north-
1:330552995255:autoScalingGroup:*:autoScalingGroupName/*"
         ]
      }
   ]
}
```

83. lépés - A Deploy csak sikeres CI után fusson
A futásidők elemzése alapján a CI és a Deploy HIBÁSAN egyszerre indultak (push-ra), így a Deploy elindulhatott akkor is, ha a minőségi kapu (flake8 + Bandit + pytest) még nem zárult le. A szakdolgozat 12. ábráján is szekvenciális folyamat szerepel: Git push → CI minőségi kapu → (csak siker esetén) Deploy. Ezt most a deploy.yml módosításával kényszerítjük ki.

ci-cd-gha-aws > .github > workflows > deploy.yml

```yaml
name: Deploy to Elastic Beanstalk

on:
  workflow_run:
    workflows: ["CI"]
    types:
      - completed

permissions:
  id-token: write
  contents: read

env:
  AWS_REGION: eu-north-1
  APP_NAME: mysite
  ENV_NAME: Mysite-env-2
  S3_BUCKET: elasticbeanstalk-eu-north-1-330552995255

jobs:
  deploy:
    if: ${{ github.event.workflow_run.conclusion == 'success' }}
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4
        with:
          ref: ${{ github.event.workflow_run.head_sha }}
          fetch-depth: 0

      - name: Set up Python
        uses: actions/setup-python@v5
```

```
commit & push
git add .github/workflows/deploy.yml
git commit -m " CI quality gates enforced before deployment (sequential CI→CD flow)"
git push
```

Összegzés

A dokumentumban bemutatott lépések eredményeként sikerült létrehozni egy teljesen működő CI/CD pipeline-t, amely automatikusan végrehajtja a kód integrációját, tesztelését és telepítését az AWS Elastic Beanstalk környezetbe. A rendszer a GitHub Actions segítségével biztosítja a folyamatos integrációt, míg az AWS IAM OIDC alapú hitelesítés biztonságos kapcsolatot teremt a GitHub és az AWS között.

A folyamat végén az alkalmazás éles környezetben is stabilan, hibamentesen futott, zöld („Healthy”) állapotot jelezve az AWS konzolon. A megoldás egyaránt demonstrálja a felhőalapú infrastruktúrák rugalmasságát és az automatizált DevOps-megközelítés gyakorlati előnyeit.

2. melléklet:

CI/CD pipeline kulcskódrészletek és workflow-definíciók

A következő kódrészletek a szakdolgozat gyakorlati megvalósításának legfontosabb elemeit szemléltetik. A teljes forráskód nyilvánosan elérhető a GitHub-repozitóriumban: https://github.com/szelese/ci-cd-gha-aws

.github/workflows/ci.yml CI workflow

Ez a workflow minden „push” eseménykor elindul a main ágon, lefuttatja a lintelést, a biztonsági szkennelést és a teszteket.

```
name: CI

on:
  push:
    branches: [ "main" ]
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest
    env:
      DJANGO_SECRET_KEY: ${{ secrets.DJANGO_SECRET_KEY }}
      DJANGO_SETTINGS_MODULE: mysite.settings
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'   # Should be the same as in runtime.txt (python-
3.11) / ugyanannak kell lennie,mint a runtime.txt-ben (python-3.11)

      - name: Install deps
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
#new steps begin új lépések kezdete
      - name: Linting check (flake8)
        run: |
          flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
```

```
          flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --
statistics

      - name: Security check (bandit)
        run: |
          bandit -r .

      - name: Run Unit Tests (pytest)
        run: |
          pytest --ds=mysite.settings
#new steps end új lépések vége

      - name: Quick check
        run: python -c "import django,sys; print('Django', django.get_version(), '|
Python', sys.version.split()[0])"
```

.github/workflows/deploy.yml CD workflow (deploy.yml)

Ez a GitHub Actions folyamat az OpenID Connect (OIDC) hitelesítéssel bejelentkezik az AWS-be, létrehozza az új Elastic Beanstalk-verziót és frissíti a környezetet.

```
name: Deploy to Elastic Beanstalk

on:
  workflow_run:
    workflows: ["CI"]
    types:
      - completed

permissions:
  id-token: write
  contents: read

env:
  AWS_REGION: eu-north-1
  APP_NAME: mysite
  ENV_NAME: Mysite-env-2
  S3_BUCKET: elasticbeanstalk-eu-north-1-330552995255

jobs:
  deploy:
    if: ${{ github.event.workflow_run.conclusion == 'success' }}
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4
        with:
          ref: ${{ github.event.workflow_run.head_sha }}
```

```
          fetch-depth: 0

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.11"

      - name: Install dependencies (optional)
        run: |
          python -m pip install --upgrade pip
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi

      - name: Configure AWS credentials (OIDC)
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::330552995255:role/GitHubOIDC-EBDeploy
          aws-region: ${{ env.AWS_REGION }}

      - name: Create EB application bundle
        run: |
          zip -r app.zip . \
            -x ".git/*" ".github/*" "__pycache__/*" "*.pyc" ".venv/*"

      - name: Compute version label
        id: ver
        run: |
          echo "VERSION_LABEL=$(date +'%Y%m%d-%H%M%S')-${GITHUB_SHA:0:7}" >>
$GITHUB_OUTPUT

      - name: Upload bundle to S3
        run: |
          aws s3 cp app.zip s3://$S3_BUCKET/$APP_NAME/${{
steps.ver.outputs.VERSION_LABEL }}.zip

      - name: Create new EB application version   # Version check and creation /
Verzió ellenőrzése és létrehozása
        run: |
          aws elasticbeanstalk create-application-version \
            --application-name "$APP_NAME" \
            --version-label "${{ steps.ver.outputs.VERSION_LABEL }}" \
            --source-bundle S3Bucket="$S3_BUCKET",S3Key="$APP_NAME/${{
steps.ver.outputs.VERSION_LABEL }}.zip"

      - name: Update EB environment
        run: |
          echo "Updating environment: $ENV_NAME to version: ${{
steps.ver.outputs.VERSION_LABEL }}"
          aws elasticbeanstalk update-environment \
            --application-name "$APP_NAME" \
            --environment-name "$ENV_NAME" \
            --version-label "${{ steps.ver.outputs.VERSION_LABEL }}"

      - name: Wait until new app version is ACTIVE (≤15 min)
```

```
        shell: bash
        run: |
          set -euo pipefail
          TARGET="${{ steps.ver.outputs.VERSION_LABEL }}"
          for i in {1..90}; do
            read -r STATUS HEALTH VERSION <<<"$(aws elasticbeanstalk describe-
environments \
              --environment-names "$ENV_NAME" \
              --query 'Environments[0].[Status,Health,VersionLabel]' \
              --output text)"
            VERSION="${VERSION//$'\r'/}"
            echo "[$i/90] Status=$STATUS  Health=$HEALTH  Version=$VERSION
(target=$TARGET)"
            if [[ "$STATUS" == "Ready" && ( "$HEALTH" == "Green" || "$HEALTH" == "Ok"
) && "$VERSION" == "$TARGET" ]]; then
              echo "✅ Deployed app version is ACTIVE: $VERSION"
              exit 0
            fi
            sleep 10
          done
          echo "⛔ The app version did not switch in time ($TARGET). Latest events:"
          aws elasticbeanstalk describe-events \
            --environment-name "$ENV_NAME" \
            --max-items 40 \
            --query 'Events[*].[EventDate,Severity,Message]' \
            --output table || true
          exit 1
```

A fenti kódrészletek a szakdolgozatban bemutatott CI/CD rendszer működésének gerincét alkotják. A pipeline-megoldás a folyamatos integráció és automatikus telepítés minden lépését lefedi, és az OpenID Connect-alapú hitelesítésnek köszönhetően statikus kulcsok nélkül, biztonságosan működik.

3. melléklet

Teljesítménymutatók és terheléses tesztelés

3.1. Pipeline futási statisztikák

Az alábbi táblázat a GitHub Actions és az AWS Elastic Beanstalk környezetben lefuttatott CI (Continuous Integration) és CD (Continuous Deployment) folyamatok időmérési eredményeit tartalmazza.

A következő statisztikákat a GitHub CLI -vel generáltattam és mivel az első melléklet 79. és 80. lépésében megadott legkisebb jogosultság miatt 15 iterációs lépés nagyon sok rossz adatot generált, ezért (is) tisztítanom kellett az adatokat. Pontosan ennek az adatsornak az elemzése közben vettem észre, hogy a CI és CD ágam párhuzamosan fut és a CD ág futásának nem előfeltétele a CI ág futása.

CI futászáma	CI futás idő (mp)	Deploy futászáma	Deploy futás idő (mp)
1	16	1	26
2	29	2	27
3	12	3	22
4	15	4	33
5	18	5	43
6	14	6	22
7	16	7	23
8	19	8	31
9	14	9	93
10	17	10	50
11	14	11	59
12	19	12	60
13	29	13	73
14	22	14	59
15	20	15	140
16	22	16	148
17	20	17	144

Forrás: GitHub Actions Workflow Run és AWS Elastic Beanstalk Logs (2025)

saját szerkesztés

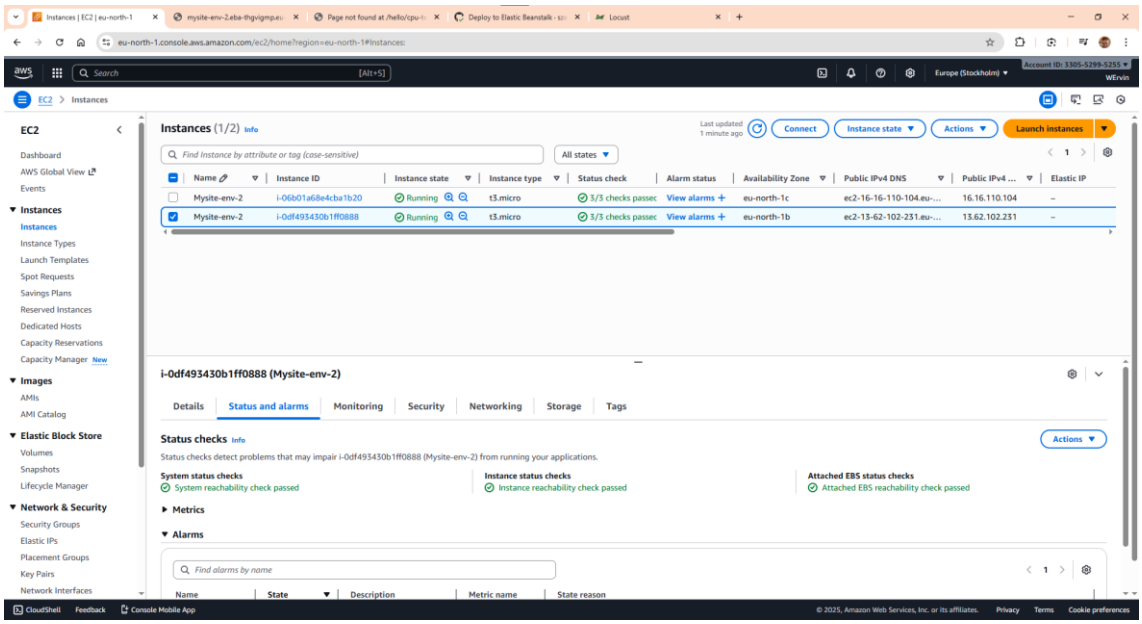
https://github.com/szelese/ci-cd-gha-aws/actions

3.2. Automatikus visszaállítás (Rollback) demonstrációja

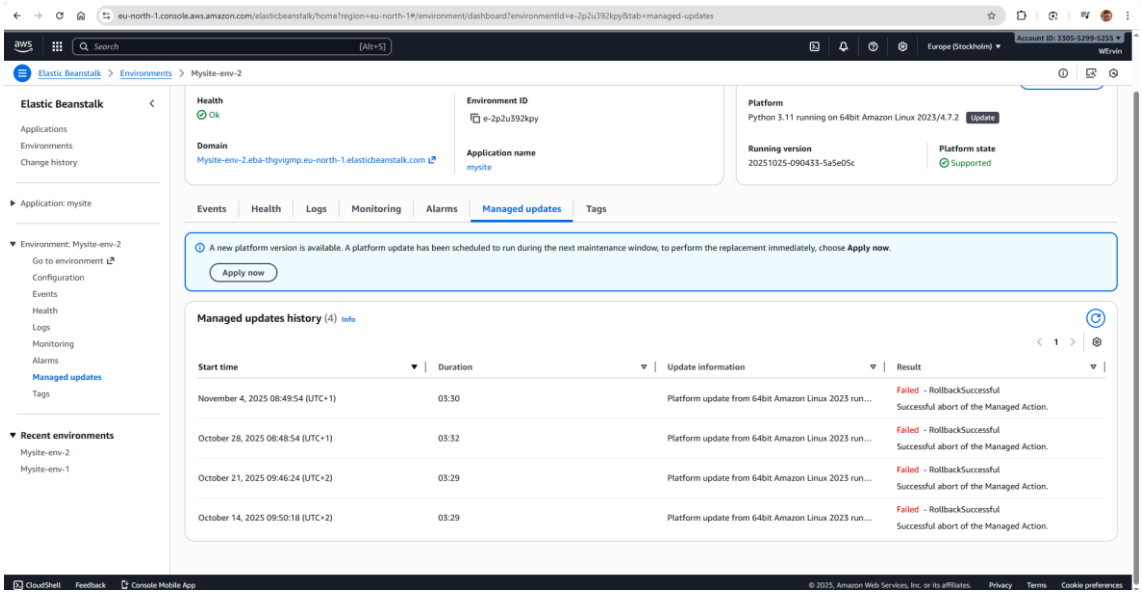
A CI/CD pipeline megbízhatóságának kulcseleme a hibakezelés. A DORA-mutatók közül a Helyreállítási Idő (MTTR) mérésére szándékosan hibás konfigurációt töltöttem fel az Elastic Beanstalk környezetbe, hogy teszteljem a rendszer automatikus hibajavító képességét. A cél a szakdolgozat pontjában leírt visszaállítási idő igazolása volt.

Hiba diagnosztikája:

A hibás telepítési kísérlet azonnali hatása az EC2 példány szintjén volt látható. Az alkalmazás nem tudott elindulni, ezért a példány elérhetetlenné vált, ahogy azt az Instance reachability check failed (Példány elérhetőségi ellenőrzés sikertelen) állapot is jelezte.



Automatikus visszaállítás:

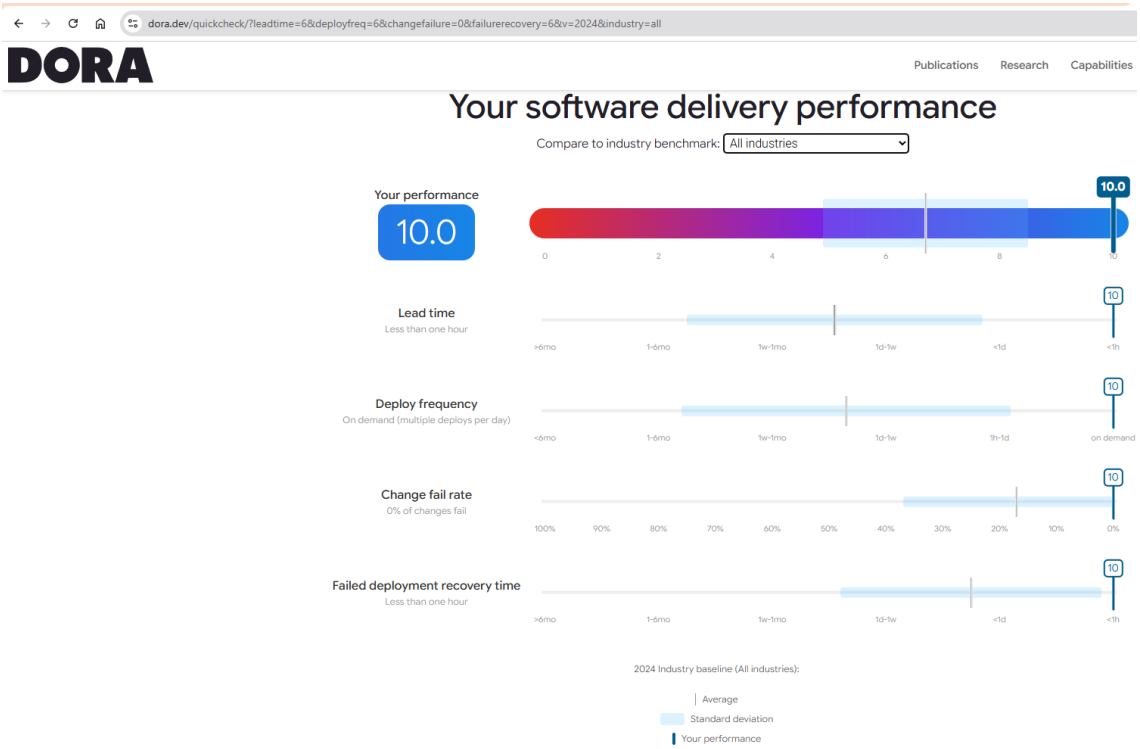


Az Elastic Beanstalk felügyeleti rétege azonnal észlelte, hogy az új verziót futtató EC2 példány "beteg" (nem felelt meg az állapotellenőrzésnek). Ennek eredményeként a frissítést Failed (Sikertelen) státuszúra állította, és automatikusan elindított egy RollbackSuccessful (Sikeres visszaállítás) folyamatot.

Eredmény

Ahogy a fenti képen a "Duration" oszlop is alátámasztja (03:30, 03:32, 3:29, 3:29), a rendszer kevesebb mint 4 perc alatt sikeresen visszaállt az előző, stabilan működő alkalmazásverzióra. Ez a gyakorlati teszt igazolta, hogy a választott PaaS-megoldás automatikus hibakezelése megfelel a szakdolgozatban is említett "Elit szintű" MTTR-követelménynek.

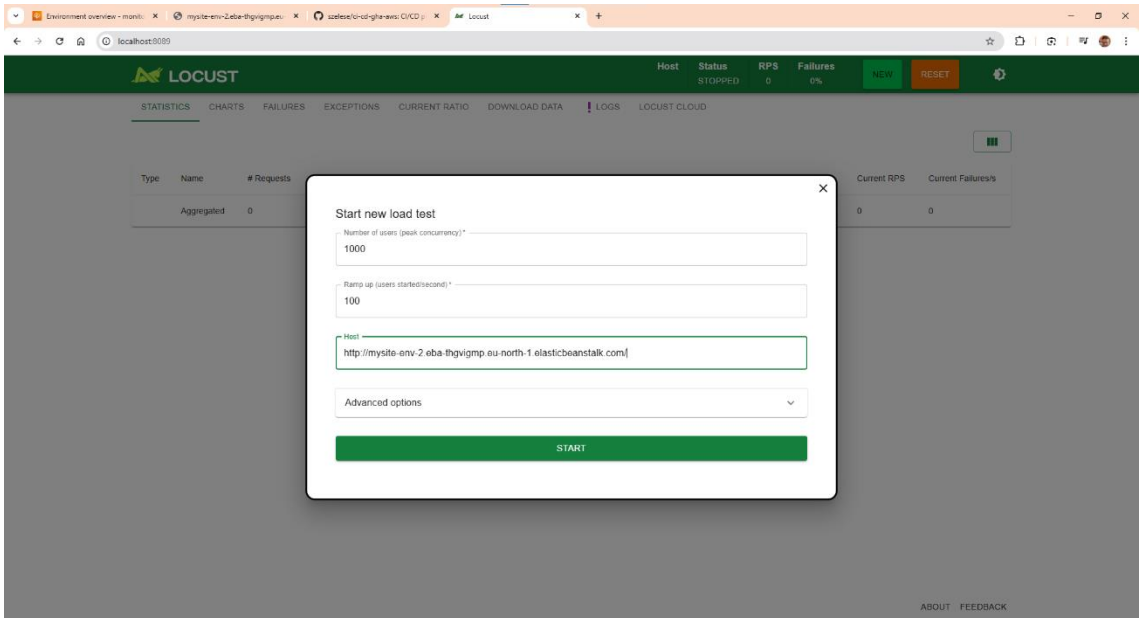
DORA mutatók:



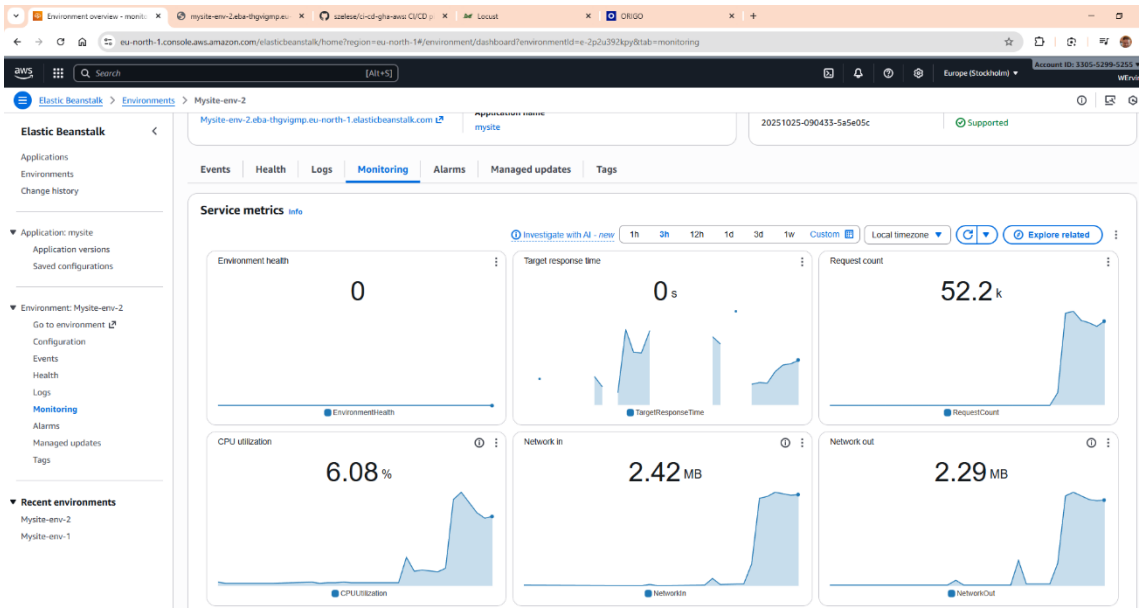
A https://dora.dev/quickcheck/ oldalára betápláltam a mért adatokat (Forrás: saját mérés).

3.3. Terheléses tesztelés (Locust)

Az alap pipeline-terhelés nem volt elegendő a skálázás kiváltásához, ezért ki kellett bővíteni a pipeline-unkat egy célzottan a CPU-t célzó view-val. A Locust telepítése után célba vettem a főágat a programmal.



Eredményre nem vezetett, de már 1 fokkal hatásosabb volt, mint a PowerShelles tesztelés.

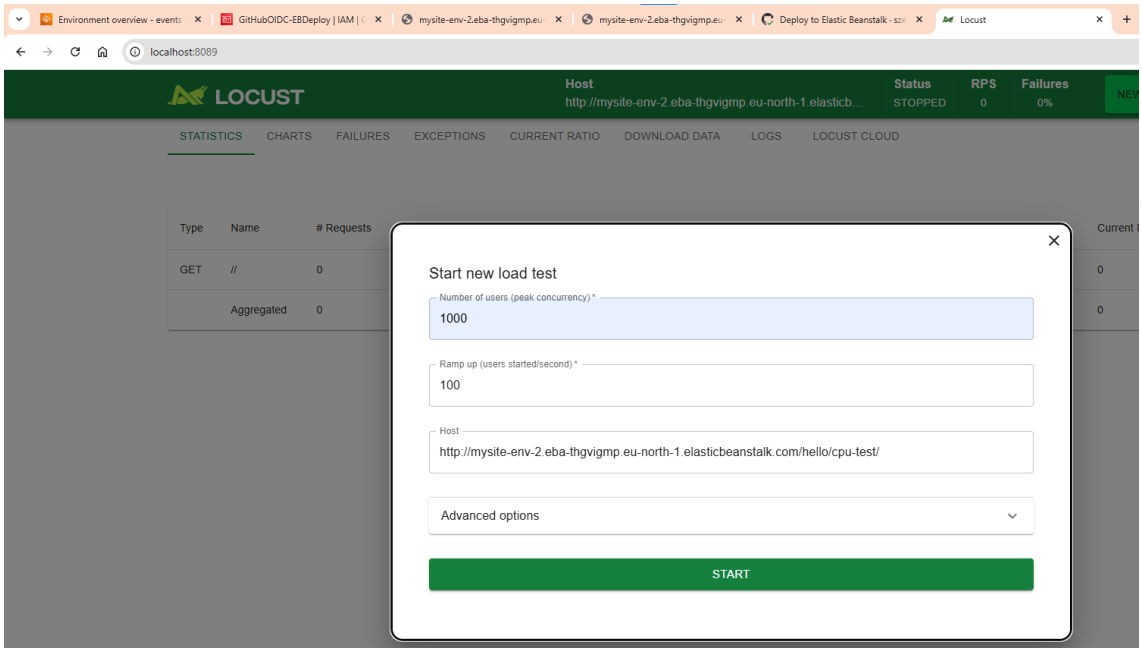


Diagnosztika

A Locust teszt (4% hibaarány és megnövekedett válaszidővel) és az AWS metrikák (magas hálózati forgalom mellett alacsony CPU-kihasználtság) egyértelműen jelzik a problémát. A "Hello World" alkalmazáslogika minimális számítási igénnyel bír, így nem generál érdemi CPU-terhelést. A szűk keresztmetszetet (bottleneck) ebben az esetben

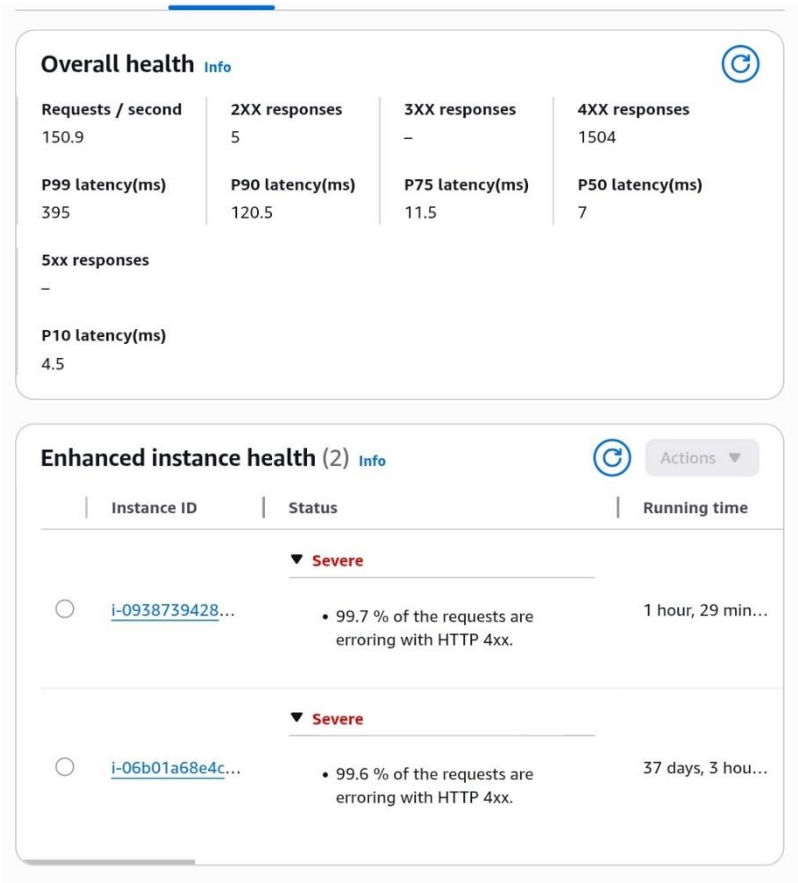
nem a CPU, hanem vélhetően a Gunicorn-kapcsolatok limitje vagy a t3.micro példány hálózati I/O korlátja okozza. Míg a szerver a kérések alatt telítődött, a CPU kihasználatlan maradt. Ezzel a végponttal az auto-skálázási küszöb (60% CPU) elérése nem lehetséges. Ezért bővítettük az alapview-unkat pusztán tesztelési célból, hogy bebizonyíthassuk a pipeline automatikus skálázhatóságát.

http://mysite-env-2.eba-thgvigmp.eu-north-1.elasticbeanstalk.com/hello/cpu-test/

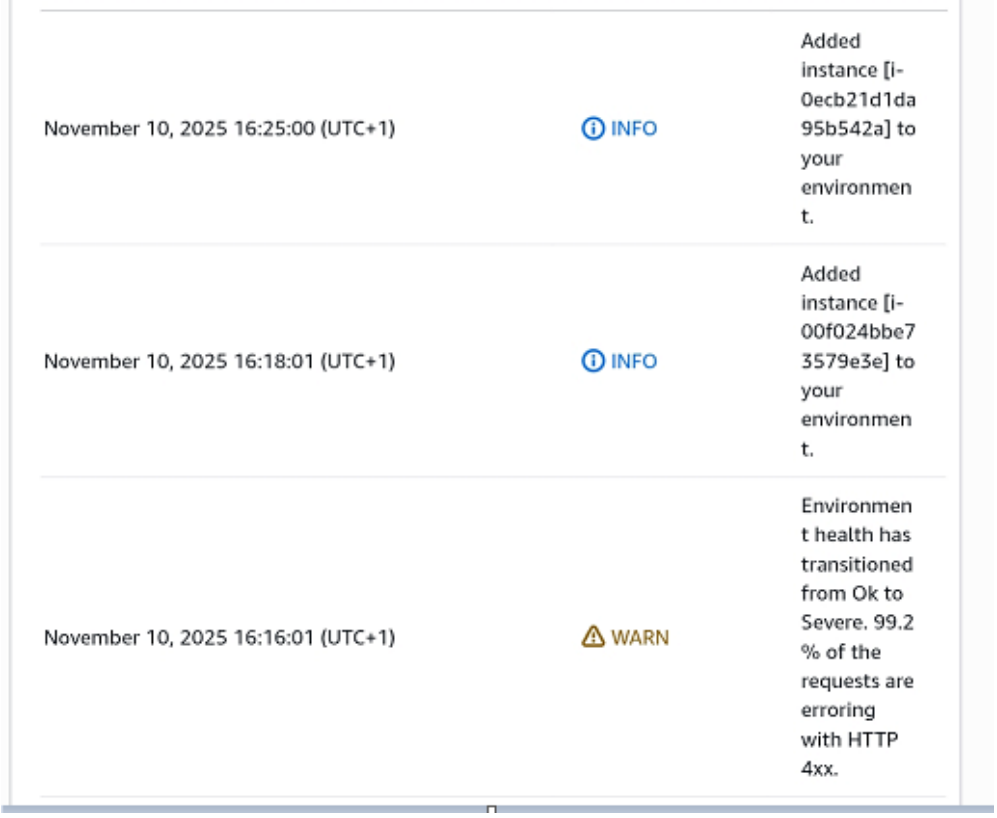


túl gyengére sikerült még mindig a tesztem, ezért bővítettem egy Hashteszttel.

az új teszttel már jó lett példányok túl terhelődtek, lásd 19-es ábrát is:

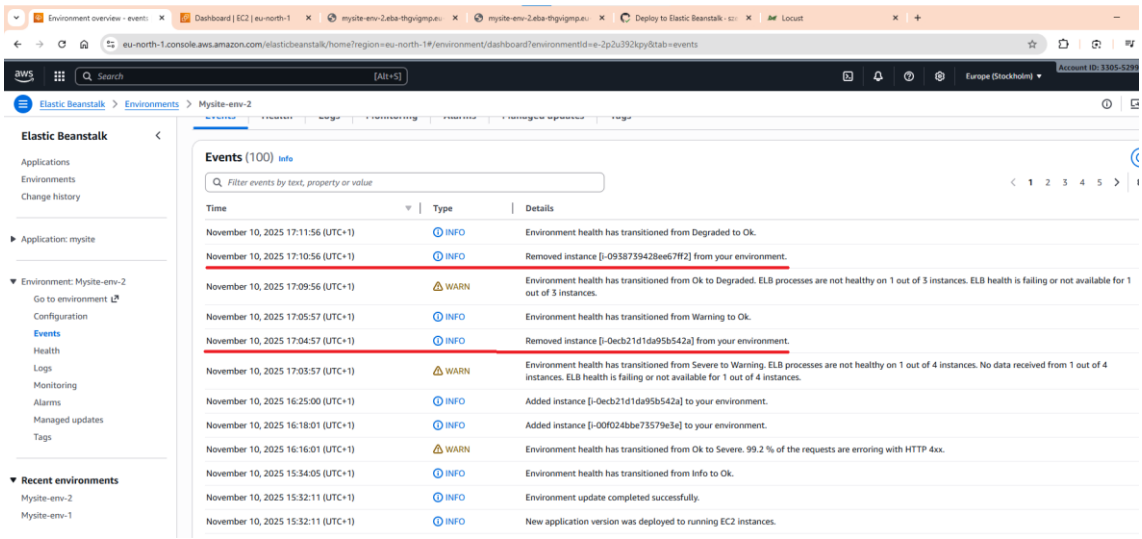


új példányok jöttek létre automatikusan:



A stressztesztet a Locuston:

A terhelés vége után a rendszer automatikusan kikapcsolta a két példányt.



Ez a folyamat bizonyítja a rendszer automatikus horizontális skálázhatóságát és az önhelyreállító (self-healing) képességét.