Budapesti Corvinus Egyetem
Gazdálkodástudományi Kar
Számítástudomány Tanszék

SZOLGÁLTATÁSFOLYTONOS ALKALMAZÁSOK MŰKÖDTETÉSE

Készítette: Oroszi Róbert
Gazdaságinformatikus szak
2015

Konzulens:
Dr. Mohácsi László

Tartalomjegyzék

Rövidítésjegyzék

BYOD (Bring Your Own Device) Olyan vállalati szabályozás, amely megengedni, hogy a szervezet tagjai a saját eszközeiket használják a vállalati adatok és rendszerek elérésére. Mivel a munkavállalók a céges szabályok ellenére is magukkal viszik az eszközeiket, ezért inkább támogatják őket a szervezetek, ahelyett hogy korlátoznák a használatukat.

commit message Az a szöveg, amely a feltöltött kód mellé kerül csatolásra. Általában rövid, lényegretörő üzenet, amely összefoglalja a kód által elvégzett műveletet. Az üzenet tartalmazhat olyan paramétert, amely alapján az üzenetek más üzenetekhez rendelhetők, illetve csoportosíthatóak. Gyakran szokás az üzenetben jelezni a kapcsoló hibajegy azonosítóját.

full-text search engine Olyan adatbázisokat szokás full text search engine-nek nevezni, amelyek szövegkeresésre nyújtanak megoldást. Ezek a rendszerek meg tudják különböztetni az általános célú szavakat, és a szövegbányászati algoritmusoknak köszönhetően a szó szinonimáit, csonka szavakat is képesek felismerni. A legtöbbet használt szövegkeresési motor a Lucene, amely többek között az ElasticSearch és a Solr adatbázisok alapjait adja.

funkció kapcsoló (feature flag) Az élesítésnek egy olyan formája, amely során a funkció elérhető produkciós környezetben, viszont csak a kiválasztott felhasználók számára. A kapcsoló használata bevált módszer a kísérleti-, félkész-, validálás alatt álló funkciók bevezetésére

IaaS Infrastructure as a Service a felhő alapú szolgáltatások egy olyan típusa, amelyben nem egy konkrét kerül értékesítésre, hanem a számítási kapacitás egy virtualizált környezetben. A legismertebb ilyen

szolgáltatók az Amazon Web Services, Joyent és Google Cloud.

IRC A mozaikszó az Internet Relay Protocol összevonásának eredménye. 1988-ban készített protokoll, a mai napig nagy felhasználótábornak örvend, a nyílt protokollnak és könnyű integrációnak köszönhetően.

Jabber Egy XMPP (Extensible Messaging and Presence Protocol) alapú instant üzenetküldésre alkalmas rendszer. Csoportos és egy az egy közötti beszélgetések lebonyolítására is alkalmas. A nyílt protokollnak köszönhetően sok kompatibilis kliens és szerverimplementáció is létezik.

PaaS Platform as a Service a felhőszolgáltatások egy típusa, amely a számítási kapacitáson túl a szoftverkomponensek futtatására már elő van készítve, azaz az infrastruktúra kiépítése és karbantartása a szolgáltatás része. A leggyakrabban használt PaaS szolgáltatók az AWS Beanstalk, Heroku.

push notification A kommunikáció olyan formája, amikor az üzenet küldését követően a fogadó azonnal értesítést kap, ellentétben a pull alapú értesítésekkel, amelyben a bizonyos időnként kerül ellenőrzésre, hogy érzekezett-e értesítés

QA team A minőségbiztosítási csapat, amely a szoftver manuális tesztelését és a tesztelésautomaziálást végzi. Feladata a szoftver hibáinak felderítése, az előzetesen működés ellenőrzése. A minőségbiztosítási csapatnak nem feladata a hibák kijavítása.

Runbook A szoftverek működtetésének és azok működése során fellépő incidensek kezelésének lépéseit tartalmazó gyűjtemény.

SaaS Software as a Service, azaz olyan szoftverértékesítési forma, amely során a vásárló előfizetés útján fér hozzá a szoftverhez. A SaaS modell esetén az alkalmazás központilag hosztolt és általában vékony kliensről (leggyakrabban böngésző) kerül elérésre.

spagetti kód A kifejezést olyan programkódra használják pejoratív értelemben, amelyek komplex vagy kevésbé komplex feladatot struktúrálatlanul, átláthatatlanul oldalanak.

1. fejezet

Bevezetés

Az interneten böngészve a felhasználó gyakran észre sem veszi, hogy egy weboldalon történő két kattintása között elképzelhető, hogy az éppen böngészett rendszer verziót váltott, azaz frissítette a rendszerét. De hogyan történik mindez? Hogyan lehetséges leállás nélkül naponta többször verziót váltani, akár földrajzilag elosztott rendszereket populálni? Ezekre a kérdésekre kíván a szakdolgozat válaszokat adni.
Bemutatásra kerül, hogy hogyan lehetséges a kód minőségének folyamatos ellenőrzése és annak integrálása, hogyan tudja a szervezet kezelni és megszervezni saját kommunikációját annak érdekében, hogy a szoftverrendszerek fejlesztése folyamatos legyen.
Míg maguk a szoftverfejlesztési metodológiák köre az elmúlt évek egyik kedvenc témája, addig a szoftverrendszerek működtetése egy kevésbé a figyelem központjában lévő szakmai feladat. A dolgozat nem kívánja a működtetés minden apró mozzantát bemutatni, hanem egy résztémakör fontos pontjaira próbál fókuszálni, felhívva a figyelmet a meghatározó sarokpontokra. Továbbá próbálja egy olyan megoldást feldolgozva bemutatni a kódkészítés-ellenőrzés-beolvasztás-élesítés-hibadetektálás-javítás folyamatát, melynek használatával és segítségével a tanuló szervezet reszponzívabb és átláthatóbb lehet.
A dolgozat alapvetően a webes alkalmazásokra fókuszál, de technológiafüggetlen, és az itt leírt pontok akár asztali- vagy mobilalkalmazásoknál is felhasználhatók.

2. fejezet

Folyamatos integráció

Continuous Integration

> A Continuous Integration - azaz a folyamatos integráció - egy olyan szoftverfejlesztési módszertan, amelyben a fejlesztőcsapat tagjai által írt kód napi rendszerességgel kerül (vagy automatizálással minden funkció, javítás implementálása után) integrálásra a korábbi fejlesztések közé. Minden új kód integrálása során automatizált tesztek ellenőrzik, hogy a rendszerbe való illesztés során okozott-e valamilyen hibát az új kódrészlet, illetve megfelel-e az adott fejlesztőcsapat által meghatározott minőségi kritériumoknak és ennek eredményéről a lehető leghamarabb visszajelzést ad. *Continuous Integration (original version)* 2002

A szoftverfejlesztés során általában egy projekten több fejlesztő is dolgozik a kód különböző vagy akár egyazon részein is. A fejlesztők az alkalmazás másolatával dolgoznak a saját munkaállomásukon (nem pedig közvetlenül egy helyen). Ha egy feladat elkészül, akkor fel kell másolni a módosított fájlokat egy közös tárolóba/-szerverre. Viszont a felmásolás előtt mindenképpen szükséges frissíteni a saját lokális példányukat, hogy elkerüljék a ütközéseket, illetve egymás munkájának felülírását, ezt a folyamatot nevezik integrációnak. Azonban előfordulhat, hogy a központi tároló és a saját lokális másolat között akkora különbség lehet, hogy nagy mértékben kénytelen a fejlesztő módosítani a saját fejlesztését, majd a javítás elvégzése után következhet ismét egy frissítés (integrálás), majd esetleg a megismételt javítás. Ez mint látható, egy ördögi kör. Ennek a megoldására jött létre a Continuous Integration, amelynek segítségével a fejlesztők adott időközönként megismétlik az integrációs lépést, hogy minél kevésbé térjen el a lokális verzió a központi tárolóban lévő verziótól.

Hiába tűnik ez egy triviális és egyszerű megoldásnak, ez a megközelítés csak a 2000-es évek elején született meg, viszont azóta töretlen népszerűségnek örvend. Mára a folyamatos integráció összekapcsolódott az automatikus fordítással (build automation), azonban alapvetően nem szükséges része. Tehát például egy céges előírás, amely megköveteli, hogy a fejlesztők kötelesek minden reggel frissítsék a lokális másolatuk frissítésére, tulajdonképpen folyamatos integrációnak tekinthető, mert ezáltal megvalósítják a rendszeres integrációt. Ezek alapján kijelenthető, hogy a Continuous Integration valójában csak egy verziókezelő (pl.: git, Subversion, Mercurial) használatát követeli meg.

Az automatizált fordítás, habár nem alapvető része a folyamatos integrációnak mégis a fejlesztést és a fejlesztők munkáját nagyban megkönnyíti. Ez a művelet történhet

bizonyos időközönként (munkaidő után éjszaka) vagy bizonyos események során, mint például ha fejlesztők feltöltik a közös tárolóba a fájljaikat (commit). Az integrációs lépések és az automatikus fordítás közé érdemes lehet beépíteni a kódminőség ellenőrzést és a tesztelést, amely garantálja, hogy az új funkciók nem törik el a régi funkciókat. Továbbá érdemes az integrációról, a tesztelésről, és a fordításról riportokat generálni, melyekkel az eredmények - egy nem fejlesztő számára is - érthetőbb formába kerülnek.
A folyamatos integrációra, kódminőség ellenőrzésre, tesztesetek futtatására, riportok értelmezésére már sok szoftver elérhető, többek között a Bamboo, amely a Jira és az Confluence mögött álló Atlassian terméke, vagy a Travic CI, amely ingyenes szolgáltatásként elérhető bármilyen nyílt forráskódú szoftver számára, illetve a mára de-facto rendszernek tekinthető Jenkins, amely a következő fejezetben kerül bemutatásra.

2.1. Jenkins

https://jenkins-ci.org/

A Jenkins egy ingyenesen elérhető, nyílt forráskódú folytonos integráció támogató eszköz. A legtöbb verziókezelő rendszert támogatja, és több mint 300 kiegészítő érhető el hozzá, melyek segítségével könnyedén testreszabható. A legnagyobb előnye, és amiért az iparági sztenderddé válhatott, az az, hogy rendkívül moduláris, könnyen konfigurálható és képes kiszolgálni egy egyéni projekt és egy nagyobb cég igényeit is.
A legtöbb platformra (Windows, Linux disztribúciók, OSX, BSD) elérhető szoftver, amellett hogy nyílt forráskódú kereskedelmi támogatással is rendelkezik.
Azon túl, hogy a folyamatos integráció lépések sorozatába rendezhető, melyet a webes felületen keresztül, konfigurációs fájl vagy akár szakterület-specifikus nyelv (DSL) használatával is testre szabható a létrehozott feladatok (job) egymásba is integrálhatóak, a függőségi viszony (upstream, downstream) meghatározásával.
A kiegészítők segítségével testreszabható a bemeneteli forrás (verziókezelők, lokális mappa), az indítási jelzés (start gomb megnyomása, a tárolóba való feltöltés, a programozható interfészen keresztüli hívás, a szülőfeladat állapotváltozása), a kódminőségellenőrzés (checkstyle, linting), a kód építése (ant, grails, gradle), a tesztek lefuttatása (xUnit, TAP) és eredményük analizálása, majd továbbításuk további rendszerekbe (tesztekkel lefedett kód százalékos változása a kód átnézést kezelő

rendszerbe, az eltört tesztekről email értesítés) és egyéb szervezet specifikus értesítés kiküldése (release manager értesítése, a különbséget beküldő értesítése). Amint látható, a Jenkins képes alkalmazkodni a szervezetek által felállított igényekhez, technológiákhoz, ennek is köszönhető töretlen népszerűsége mind a vállalati, mind az open source világban.

2.2. TDD

Test Driven Development

A folyamatos integráció és az automatikus fordítás egyik legfontosabb velejáró kulcsszava a TDD, azaz a teszt vezérelt fejlesztés.
A TDD használatához egy elég erős szemléletváltásra van szükség, ezért legalább annyi ellenzője van, mint támogatója. A teszt vezérelt vagy teszt irányított fejlesztés a nevével ellentétben nem egy tesztelési megoldás, hanem sokkal inkább tervezési. Johansen, 2011
Egy alkalmazás jó és jól működéséhez könnyen bővíthetőnek kell lennie, így tud a termék, szolgáltatás megújulni, a továbbfejlesztés zökkenőmentesen zajlani. De hogyan történik az új funkciók tervezése, beépíthetőségi megvalósításának tervezése?
A TDD ezt próbálja elősegíteni azáltal, hogy a teszteseteket még a tervezés fázisában kell megírni, így a problémák a munka korai fázisában kiderülhetnek. A TDD keretrendszerek általában könnyen olvashatóak még a fejlesztésben kevésbé járatos projektrésztvevők számára is, a 2.1 ábra ezt kívánja bemutatni. 2.1
Mivel a TDD teszteseteket a fejlesztés alatt folyamatosan futtatják - ellenben az egységtesztekkel, melyek általában az integráció során futnak le -, ezért kimondottan gyorsnak, mindenhol, bármilyen sorrendben futtathatónak kell lenniük, mert egyébként a fejlesztők nem fogják felhasználni őket. Teszteseteknek csak akkor kellene változniuk, ha a kód ezt megköveteli, illetve ha a specifikáció változik.
A fejlesztési folyamat négy lépésre bontható:

Feladatok meghatározása

Ebben a lépésben az ügyfél igényeknek megfelelő funkcionalitást kell összegyűjteni tesztesetek formájában, és meg kell határozni, hogy mit kellene tennie a rendszernek.

Fontos, hogy nem azt kell itt kitalálni, hogy az adott fejlesztő hogyan valósítsa

```
1  suite( "This test covers the login window for the application.",
2    function(){
3
4    suiteSetup(function(){
5      this.application = new Application();
6    });
7
8    test( "after logging in, the username should be 'MyMscThesis'",
9      function(){
10       var user = this.application.login({
11             username: "MyMscThesis",
12             password: "MySecretPassword"
13           });
14       user.name.should.equal( "MyMscThesis" );
15   });
16
17 });
```

2.1. ábra. Egy TDD teszt mocha és should keretrendszerekkel

meg az adott feladatot, hanem hogy mit kell majd megvalósítani. A "mit" meghatározása által a fejlesztő is jobban megérti a feladatot, így kisebb a félreértés lehetősége. A teszteset megírása után következik a megvalósítás.

Megvalósítás

A tesztesetek elkészültével következik az előre specifikált feladat megvalósításának megtervezése, melyet a megvalósítás követ.

A megvalósításnak kellene a legkönnyebb résznek lennie, ha mégsem könnyű, akkor a következő problémák fordulhattak elő.

Túl nagy feladat került meghatározásra az előző lépésben, így a következő lépésben az adott feladat kisebb részfolyamatokra bontásának kell megtörténnie.

Felmerül a kérdés, hogyan lehet könnyű a megvalósítás, ha előtte még egy alkotóelemet is el kell készíteni? Ilyenkor az adott alkotóelem feladatait kell meghatározni úgy, hogy a fejlesztők lesznek az ügyfelek és a ő igényeiket kell kielégíteni, és TDD alapján lefejleszteni.

Ha nem nagy lépésről van szó, de mégis nehéz megvalósítani, akkor refaktor-

álni kell az adott részt. Ezt okozhatja az, hogy koszos a kód, vagy nincs jól megtervezve. A refaktorálást a már korábban megvalósított tesztesetek teszik lehetővé, melyek biztosítják, hogy a kódújraszervezés során nem fog eltörni egyik korábbi vagy jövőbeni komponens sem.

Ellenőrzés

A megfelelő eszközzel le kell ellenőrizni, hogy sikeresek lettek-e a tesztek. Ennek gyorsnak és könnyűnek kell lennie. (A tesztek nem futhatnak néhány másodpercnél lassabban.) Ezáltal folyamatosan ellenőrzött, tervezett és fejlesztett lesz a kód.

Tisztítás

Ha sikeresen le lehet futtatni a teszteket, akkor következik a kód tisztítása. A működő kódot át kell nézi, a duplikációkat eltüntetni.

Ajánlott beszédesebb neveket választani a változóknak, a függvények hosszát érdemes csökkenteni kiszervezés segítségével, újrafelhasználhatóság szem előtt tartása mellett. Mivel az ügyfél számára fontos funkcionalitás már le van tesztelve, ez a művelet már könnyedén elvégezhető, ugyanis ha például egy metódus neve elgépelésre kerül, akkor rögtön jelez a teszt, hogy hiba van. Ettől tisztább és karbantarthatóbb lesz a kód.

Érdemes megfigyelni, hogy az összes lépés megfeleltethető a tervezés egy-egy részének. A meghatározott feladatok automatizálásának köszönhetően lehetséges, hogy kis, biztonságos, önellenőrzött lépésekben történjen a rendszer felépítése és tervezése. Ez segíti a feladat megértését, rákényszerít az átlátható megvalósíthatóságra a folyamatos tisztítás és az újratervezés segítségével.

TDD előnyei:

- Refaktorálást segíti.
- A kód módosítása könnyebb, hiszen hiba esetén a tesztek eltörnek, amely azonnali visszacsatolást ad a fejlesztő számára.
- Könnyebb egy tesztelhető kódot refaktorálni (a tervezés miatt).
- Az is segítség lehet, hogy hol nincs hiba.

- Segít tesztelhetővé tenni az alkalmazást.
- Rákényszerít, hogy ne legyen az alkalmazásban spagetti kód.
- Felesleges funkció nem kerül megvalósításra, csak az, ami a teszthez szükséges.
- Előre kell tervezni.
- Gyors, folyamatos visszajelzés kapható a funkció állapotáról (nem csak az adott fejlesztőknek, de a csapat többi tagjának és a projektmenedzsernek is).
- Jobban ellenőrizhető a munka.
- Van, hogy egy funkciónak nincs látható eredménye egy hétig. Ezzel szemben a TDD-nél naponta meg lehet mondani, hogy mennyi sikeres tesztet sikerült írni.
- Segít megérteni a feladatot a példákon keresztül.
- Időcsökkentő tényező a hibajavításnál és a refaktorálásnál.
- Hibajavításnál segíthet pontosabban megjelölni a hiba helyét.
- Akár dokumentációként is szolgálhat a teszt.
- Biztosítja, hogy az új kód nem érint más tesztelt egységet.
- Ha nincs TDD, akkor gyorsabban készül a szoftver, de nehezebben módosítható később.
- A TDD tisztítás része akár kódfelülvizsgálatnak is tekinthető.
- Stabilitást elősegítheti.
- Segíti a kódstrukturálást, megköveteli a modularizációt, mert egyébként a teszt megírása bonyolultabbá válik, mint a tesztelendő kód.

TDD hátrányai:

- Nő a fejlesztési idő (refaktorálásnál csökken).
- Ha nem tiszta, hogy mit kell tenni az adott feladattal, könnyen előfordulhat, hogy rossz teszt kerül megírásra, amit át kell majd írni. (Újabb időnövelő tényező.)

- Menet közben történt koncepcióváltásnál ki kell dobni a teszteket. (Újabb időnövelő tényező.)
- A program működése nem lesz hibamentes, ha a tesztek sikeres lefutnak.
- Nem csodafegyver. A rendszer tesztelésének (minőségbiztosításának) csak egy kis részét képezi (egységtesztelés, elfogadási tesztelés, integrációs tesztelés, regressziós tesztelés mellett).
- Csak tapasztalt fejlesztőkkel érdemes használni.
- A tervezést nem mindig lehet úgy alakítani, hogy az megfeleljen a TDD-nek.
- Hálózati műveleteket, fájlrendszert igénylő komponensek tesztlésére nem ajánlott.
- Páros programozásban a legjobb használni, mert így a tesztírás és az implementáció jól feloszthető.
- A tesztek írása unalmas lehet egyesek számára. Nagy fegyelemre van szükség.
- Nehéz belerázódni, ezért arra a következtetésre lehet jutni, hogy semmi értelme.
- Nehéz megmagyarázni a menedzsereknek/ügyfeleknek, hogy az elején, a fejlesztés első lépéseinél miért van szükség a többletidőre, miközben látható eredmény nem készül.

Az automatizált tesztekkel és azok folyamatos futtatásával a szervezet folyamatosan képet kaphat arról, hogy az implementáció lépései hol tartanak, a folyamatos visszajelzések már előre tudják értesíteni a döntési pozícióban lévő szereplőket, így azok információhiánya csökkenthető. Továbbá egy olyan környezet, ahol a folyamatos integráció gyakorlata használatban van, a felmerülő problémák esetén be tudja építeni az automatizált ellenőrzések közé a problémák megoldásából megszülető újabb ellenőrzéseket, amellyel meg tudja előzni, hogy ismét elkövesse ugyanazt a hibát.
A dolgozat a folyamatos integráción belül is csak a tesztvezérelt fejlesztéssel foglalkozik, de természetesen az egységtesztek, az integrációs tesztek is teljesértékű részei a folyamatnak, viszont míg az utóbbiakat a szervezetek többsége használja addig mindez a TDD-ről nem mondható el, ezért került az kiemelésre és megvizsgálásra.

3. fejezet

Folyamatos értesítés

Continuous Notification

A folyamatos integrációnál, mivel naponta több verzió is kikerülhet, sőt párhuzamosan akár több csapat vagy fejlesztő is a produkciós rendszerbe helyezhet át funkciókat, nagyon fontos, hogy a rendszer folyamatosan tájékoztassa a csapatokat az élesített és elkészített funkciók működéséről, hibáiról, illetve a többi szereplő által fejlesztett funkciók állapotáról. Mindehhez két dologra van szükség: a kommunikációra és a rendszer minden apró részének monitorozására, mert ezáltal a hibák gyorsan kiszűrhetőek, a reakció idő csökkenthető és mindenki valós időben láthatja, hogy mi történik a rendszerben. Transzparens lesz, amelynek köszönhetően a szervezetnek könnyebb a részegységek megismerése, felelősségi köreik meghatározása.

3.1. Chat rendszer

A fejlesztői csapatok, illetve a csapattagok közti kommunikáció rendkívül fontos, mely történhet verbálisan és írásban. A verbális kommunikáció megszakíthatja a munkát, illetve távoli munkavégzés esetén (főleg ha a résztvevők más országból végzik a feladataikat), akkor sokkal jobb megoldás egy chatrendszer bevezetése. A leggyakrabban használt megoldások a szervezeten belüli azonnali üzenetküldésre a következőek:

- Jabber
- IRC
- HipChat
- Campfire
- Slack

De miért kell chat, miért nem elég az email alapú kommunikáció?

A legtöbb szervezetben sajnos még mindig az email a legfontosabb kommunikációs eszköz, azonban az email lassú, nem alkalmas azonnali (instant) üzenetek küldésére, illetve gyakran kimaradnak a levelezésből megfelelő emberek, ezzel ellentétben a chaten gyorsan lehet üzeneteket váltani, visszakereshető és mindenki részese a beszélgetésnek. Természetesen a chattel nem lehet kiváltani az emailt, viszont a két eszközt kombinálva, egymást segítve lehet jól használni. Továbbá a chat mellett érvelve elmondható, hogy az emberek könnyebben elviselik a chaten

előforduló zajt, mint az emailezésben. A előző fejezetben megismert folyamatos integráció esetén minden kódfeltöltésnél az értesítések emailen keresztül történő feldolgozása, adminisztrációja hatalmas erőforrásigényt igényelne, illetve a sok üzenet okozta zaj feltehetően egy bizonyos szint felett az üzenetek automatikus törlésével járna, ezért van szükség egy ennél dinamikusabb, skálázhatóbb megoldásra, mint például egy chatrendszerre. A chat előnyeivel és a folyamatokba történő integrálásával az 5 fejezet foglalkozik részletesebben.

3.2. Naplózás, naplógyűjtés

Az alkalmazások viselkedésének monitorozása rendkívül fontos, ugyanis amint kikerülnek a belső homokozóból, akkor megváltozhat a működésük a felhasználók különböző tevékenységeire, elvégzett interakcióiktól függően; ezeknek a problémáknak az utólagos felderítésére használják a naplózást és a naplógyűjtést. Érdemes kiemelni, hogy a fejlesztési (development), tesztelési (staging/user acceptence testing) illetve az előnézeti (preview) szerverek is mind egyfajta homokozónak számítanak, hiszen valós felhasználók nem használják, a felhasználók szimulálása - akár tesztelők, akár automatizált tesztek által - szinte sosem tökéletesek, ezért van szükség a produkciós környezet által minden elvégzett művelet naplózására.
A jól használt naplózással a hibák hamarabb kiszűrhetőek, a hibák forrása könnyebben lokalizálható, azonban minden rendszer naplóinak egyidejű figyelése egyszerűen lehetetlen. Ezért használnak a legtöbb rendszerben központi naplózást és naplógyűjtést, amelynek segítségével a hibák felfedezése, forrásaiknak felderítése és a megoldása is könnyebben megoldható.
Miért lehetne a központi naplózással könnyebben megoldani egy problémát? Amennyiben az alkalmazás egyik példánya, tegyük fel lassan válaszol, miközben a többi példány hiba nélkül működik azonos terhelés mellett (ezt könnyű ellenőrizni, hiszen egymás mellé helyezhető az alkalmazás példányok naplója a naplógyűjtő rendszerben), akkor nagy valószínűséggel a hiba a példányt futtató eszközön van csak jelen, azaz nem az alkalmazásban kell keresni hibát. Ezzel az egyszerű megállapítással már sikerült is meghatározni a szükséges felelősök körét, akik elkezdhetik a probléma mélyebb vizsgálatát és remélhetőleg annak elhárítását.

Továbbá a központi naplózás könnyen implementálható a folyamatos értesítés munkafolyamataiba, amellyel egy-egy alkalmazás/termék felelőse azonnal értesülhet a fennálló hibáról, a javítást azonnal elkezdheti.
A piacon több naplógyűjtő alkalmazás is elérhető mind telepíthető, mind SaaS formában, az egyik legelterjedtebb az nyílt forráskódú szoftvercsomag, az úgynevezett ELK stack, amely az ElasticSearch adatbázis, Logstash naplógyűjtő és Kibana vizualizáció egységekből tevődik össze.

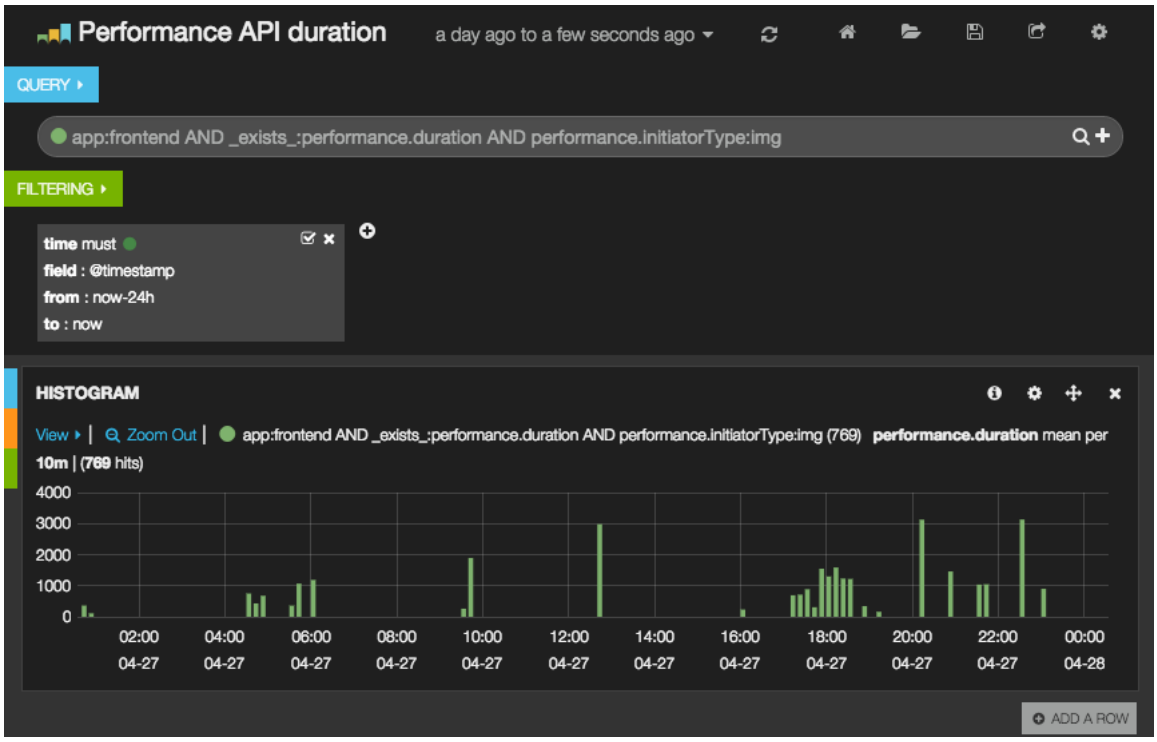
3.2.1. ELK stack

Mint ahogy már említésre került, az ELK három komponensből épül, pontosabban három önállóan is elérhető termék összekapcsolásából.
Az **E**lasticsearch egy full-text search engine, amely egy skálázható keresésre optimalizált elosztott adatbázisrendszer, amely a sémamentes tárolásnak köszönhetően az eltérő struktúrával rendelkező naplóbejegyzések tárolására tökéletes megoldást tud nyújtani.

A **L**ogstash egy naplógyűjtő rendszer, amelynek feladata a különböző protokollokon érkező (syslog, fájl, http) naplóbejegyzések egységesítése és összegyűjtése, az adatok metaadatokhoz hozzárendelése (geoip, ujjlenyomatszámítás), transzformálása (jelszavak eltüntetése, eltelt idő kiszámítása), majd továbbítása további feldolgozásra (incidens kezelés), illetve gyakran egy perzisztens tárolóba, amely általában az ElasticSearch szokott lenni.

A harmadik alkotóelem a Logstash-en keresztül beérkező ElasticSearch-ben tárolt adatok megjelenítése, amely a Kibana rendszer segítségével történik meg.

A 3.1 ábrán látható a Kibana interfésze, ez egy webes alkalmazás, amelyben adhoc és állandó irányítópulton lehet vizsgálni az adatokat. A rendszer képes valós időben is megjeleníteni adatokat és az ElasticSearch mögött álló Lucene rendszernek köszönhetően a lekérdezéshetőség bő arzénálja áll rendelkezésre a riportok tökéletesítésére.



3.1. ábra. A Kibana interfésze

3.3. Alkalmazáshibák

A naplózással a hibák könnyen észrevehetővé és később visszakereshetővé válnak, azonban a következőkre nem nyújt megoldást:

- hibák rögzítése a jegykezelő (issue kezelő) rendszerben
- egy hiba csak egyszer kerüljön rögzítésre
- automatikus értesítés a hibáról
- a hibaszázalék vizualizálása komponensenként

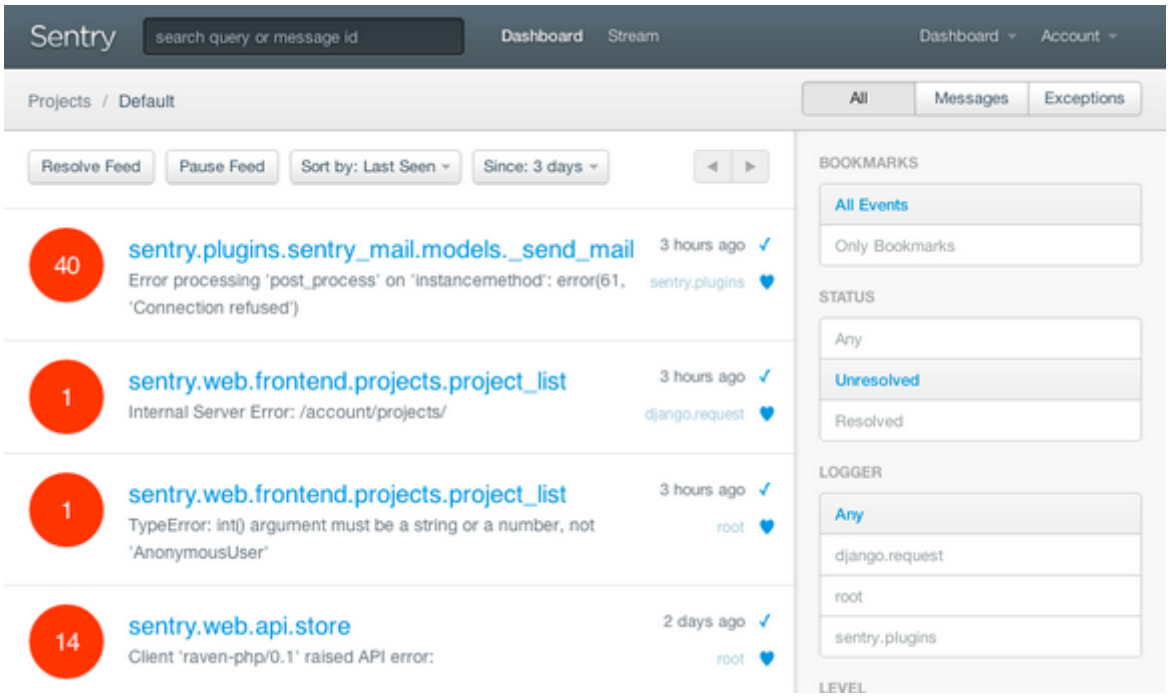
Természetesen egy alkalmazáshiba monitorozó rendszer nélkül is működhet jól, azonban előfordulhat, hogy nagy mennyiségű hibánál a naplógyűjtő rendszerben a szervezet tagjai átsiklanak problémák felett vagy egyszerűen elfelejtik azokat, mert a hibák megszokottá válnak, nem kezelik őket a megfelelő prioritással. A másik probléma az lehet, amikor a hibajelentést közvetlenül a jegykezelő rendszerbe kötik be. Ez azért lehet probléma, mert egy nagy felhasználóbázissal rendelkező

rendszernél, ha a hiba csak a felhasználók néhány százalékánál fordul elő, akkor is szükségtelenül nagy mennyiségű hibajegy keletkezését fogja előidézni.

Az alkalmazáshiba monitorozás leginkább a nem webes alkalmazásoknál (mobil és asztali) terjedt el az utóbbi időben, ugyanis ezeknél a rendszereknél a naplók nem, vagy csak nehezen gyűjthetők valós időben.

Legelterjedtebb alkalmazások:

- Sentry - `http://getsentry.com`
- Exceptional - `http://www.exceptional.io/`
- Bugsense - `http://www.bugsense.com/`



3.2. ábra. A Sentry interfésze

3.4. Monitorozás

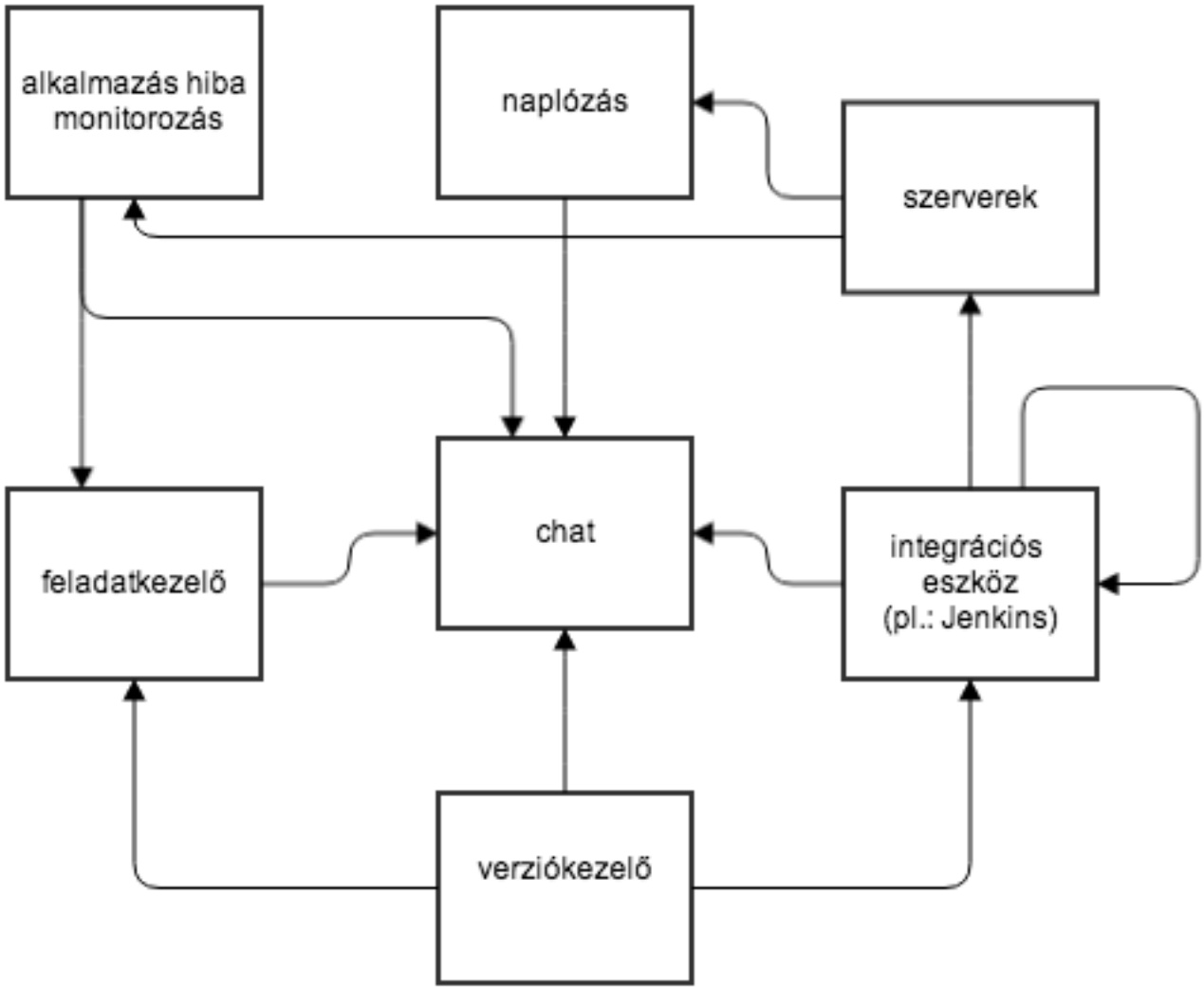
A monitorozás a naplózás egy része, viszont míg a naplózást alapvetően öndiagnosztikaként került bemutatásra a 3.2 pontban, ezért fontos kiemelni, hogy a monitorozás külső diagnosztikát jelent. Olyan értelemben külső, hogy az alkalmazás működését bizonyos időközönként ellenőrzik. Ez jelentheti a másodpercenkénti

többszöri vizsgálatát is annak, hogy működik-e a szoftver, mekkora válaszidővel rendelkezik vagy akár biztonságossági ellenőrzéseket is lehet végezni. Általában a külső forrásból érkező ellenőrzések eredményei is a naplózó rendszerbe kerülnek bele, és az alkalmazás belső naplóbejegyzéseivel együtt kerülnek elemzésre, ezáltal próbálva meg rekonstruálni a felhasználók által érzékelt teljesítményt.

3.5. Integráció

A fejezetben említett szolgáltatások, szoftverek (alkalmazáshiba monitorozó, naplógyűjtő, chat) használatával a fejlesztők, adminisztrátorok jobban megismerhetik saját alkalmazásaikat, azonban ennyi felület, különböző eszköz, aggregálatlan értesítés nyomon követése szinte már lehetetlen, ezért a legtöbb szervezet próbálja ezeket az eszközöket egymásba integrálni. A 3.3 ábrán látható a szolgáltatások egy lehetséges integrációja. Első ránézésre az ábra bonyolultnak és összetettnek tűnhet, mert sok különböző egység közötti összeköttetés megvalósítását igényli, emellett a részegységeknek valós időben kell kommunikálniuk, ismerniük kell a sorrendet, és képeseknek kell lenniük egymás üzeneteinek megértésére.
Ezeknek az igénynek a kielégítésére minden részegység azt tudja, hogy kit kell neki értesítenie; az értesítések pedig ún. *webhook* (Lindsay, 2007) használatával történnek, amely egy egyszerű HTTP kérés, ezáltal valós időben történhet meg az következő részegység értesítése.
Viszont hogyan képesek megérteni egymás üzeneteit a részegységek?
Erre sajnos nincs sztenderd specifikáció, azonban szinte minden említett eszköz nyílt forrású és széles körben elterjedt, ezért egymás üzeneteinek feldolgozása, megértése már megoldott beépülő modulok használatával.

A 3.3 ábrán felrajzolt folyamatnak az első szintje az, amikor a kód bekerül a verziókezelőbe, amely össze van kapcsolva a feladatkezelővel, ahol akár a kapcsolódó jegy lezárásra is kerülhet a kód mellé csatolt üzenet szövege (commit message) alapján, majd a chaten keresztül értesülhetnek a csapat tagjai, a projektmenedzser vagy az adott szervezeten belül bárki, aki érintett lehet az adott funkció, szolgáltatás fejlesztésében.
A verziókezelő másik kapcsolata az integrációs eszközzel van, amely lefuttatja az



3.3. ábra. A szolgáltatások egy lehetséges integrálása

előre definiált ellenőrző, tesztelési és riportálási feladatokat, amelyek garantálják, hogy az új kódsorok megfelelnek a céges előírásoknak, nem okoznak problémát az alkalmazás futtatásában. A folyamat végén értesítést küldhet a közös chatbe az integráció sikerességéről, hiba esetén értesítheti a megfelelő személyeket, ugyanis a hiba kijavításáig senki tud hibamentes kódot beküldeni az integrációs folyamatba, ezért ennek az értesítésnek a legmagasabb prioritással kell megtörténnie. Az integrációs folyamat kimenete lehetséges, hogy újabb integrációs folyamat bemenete lesz; ez olyankor fordulhat elő, amikor az alkalmazás egyik ága egy másik ágba kerül beolvasztásra. Ez történhet például a fejlesztői ág tesztelői ágba való automatikus vagy félautomatikus olvasztásakor, vagy a minőségbiztosítási csapat (QA team) tesztelői ág jóváhagyásaként a produkciós ágba való olvasztáskor.

A 3.3 ábrán továbbhaladva az integrációs eszköz után az alkalmazás a felhasználói elfogadási szerverre (user acceptance test server) vagy előnézeti (staging, preview) szerverre kerül, amely egy homokozószerű, zárt, a produkciós szerverrel megegyező architektúrájú környezetben nyújt lehetőséget az alkalmazás tesztelésére. Az alkalmazás majd ezután kerülhet ki a produkciós szerverre. Mind a felhasználói elfogadási-, az előnézeti- és a produkciós szerverek hibáit már naplózni kell, illetve az ezeken futó alkalmazásokét is; hiba esetén pedig értesíteni kell (a chaten keresztül) a hibát okozó komponens, vagy az alkalmazás hibás részéért felelős személyt, csapatot (ideális esetben, ha a hibából a hiba pontos helye is megállapítható, akkor a verziókezelő rendszer segítségével akár a hibás kódsort beküldő személy is értesíthető). A felelősök a hibáról hibajegyet készíthetnek a feladatkezelőbe, majd a fejlesztőcsapatok javítva a hibát a kódot beküldik a verziókezelőbe, és a folyamat újraindul.

Fontos felhívni a figyelmet, hogy a 3.3 ábrán rendkívül komoly hangsúlyt kapott a chat rendszer, természeten nem kötelező chatet használni, azonban a valós idejű értesítésekre, illetve a reakcióidő csökkentése érdekében mindenképpen ajánlott. Továbbá az ábrán nem szerepel, de a chat alapú notifikálás mellett érdemes egyéb értesítési formákat is használni, mint például az emailt, vagy magas prioritású hibáknál (alkalmazásleállás) érdemes lehet megfontolni az sms, okostelefonos valós idejű üzenet (push notification), esetleg csipogó használatát is.

3.5.1. Automatizálás

Ha egy szervezet úgy dönt, hogy a folyamatait mélyen egymásba integrálja, akkor rövid időn belül találkozni fog a következő problémákkal:

Rendszer-automatizáció lépései

Hiába automatizált a folyamat, ha az egy-egy pontról történő elindítás/újraindítás nem lehetséges, bárhonnan, bármikor, akkor rengeteg plusz munkát igényel a megelőző (és már korábban sikeresen végrehajtott) lépések újbóli végrehajtása.

Ilyen probléma lehet az integrációs rendszer elindítása a kód egy pillanatnyi állapotától kód beküldés nélkül, amelyet gyakran nem tesznek lehetővé, pedig a kód életében sokszor előfordul, hogy az integráció egy kívülálló ok miatt (instabil hálózati kapcsolat, a rendszer általános leterheltsége) sikertelen stá-

tuszba kerül és újra kell indítani új kód beküldése nélkül.

Távoli irányítás

Lehet, hogy egy asztali számítógépen könnyen elvégezhető a produkciós rendszerben egy verzióváltás, de miért ne lehetne ezt megtenni bármilyen más eszközről, akár okostelefonról keresztül?

Kommunikáció külső eszközök használatakor

Ha egy üzenet érkezik (például alkalmazás újraindítás szükségességéről), akkor a csapat tagjainak meg kell beszélniük, hogy ki javítja meg a problémát.

Ezekre a problémákra egyszerű megoldásként javallott a robotok alkalmazása, amely képesek emulálni a felesleges lépéseket a folyamatokban; egyszerű felületet, hozzáférést nyújtanak akár okostelefonok számára is, illetve könnyen illeszkednek a kommunikációba.

Hubot - http://hubot.github.com/. A GitHub ingyenesen elérhető chat robotja a 3.5.1 alfejezetben említett problémákra próbál egyszerű és univerzális megoldást nyújtani. Képes együttműködni a legismertebb chatrendszerekkel és mindezek mellett észrevétlenül integrálódik a feladatkezelő rendszerekkel, a hibagyűjtő alkalmazásokkal. Továbbá az elérhető dokumentációknak, a nagy mennyiségű közösségi támogatásnak és a széles körű használatának köszönhetően bárki készíthet hozzá egyéni kiegészítőket melyek, a saját belső rendszerekkel is könnyen integrálódhatnak. A kiegészítők hivatalos gyűjteménye a `https://hubot-script-catalog.herokuapp.com/` címen érhető el.

Fontos megjegyezni, hogy egy chat robottal nem a chat kerül automatizálásra, hanem a rendszert felépítő szolgáltatások.

A robottal történő automatizálásnak két oka van, azzal szemben, hogy egy új rendszer kerüljön bevezetésre. Az első, ha a szervezet rendelkezik már chatrendszerrel, akkor a szervezeti tagjainak nagyobb lesz az ellenállása egy új rendszerrel szemben, mint a meglévő chat kibővítése ellen lenne. A másik ok pedig a kontextus váltások minimalizálása. ha a felelős személyeknek elég a chaten egy megfelelő parancsot kiadni (például ahelyett, hogy belépnek a feladatkezelő rendszerbe, ahol átállítják a megfelelő állapotra a hibajegyet), akkor időt spórolhatnak. Mindemellett a chaten keresztül a csapat többi tagja is látja, hogy éppen ki mit csinál, vagy akár

később a chat naplóból is visszaellenőrizhető a végrehajtott lépések sorozata. A chat alapú működéssel bővebben az 5. fejezet foglalkozik.

4. fejezet

DevOps

Az informatikai fejlesztések evolúciójának köszönhetően megszületett agilis irányzatok (agilis fejlesztés, scrum, kanban irány, extrém programming) megszüntették a klasszikus funkcionális csapatokfelépítést, ennek a következő iterációja a DevOps irányzat, amely a fejlesztés (**Dev**elopment) és az üzemeltetés (**Op**erations) szavaknak a keresztezéséből született meg.

4.1. Kialakulása

Keresztfunkciós és termékcsapatok.
A keresztfunkciós (cross-functional) csapatok és termékcsapatok létrejöttével az egy-egy funkcióra specializálódott csapatok (minőségbiztosítás, frontend, middleware, backend csapatok) felbomlottak és kész szolgáltatások fejlesztésére rendezkedtek be. A szolgáltatást nyújthatják akár cégen belül, akár cégen kívül, de a csapat felelőssége a következőkre terjed ki:

- tervezés
- megvalósítás
- tesztelés
- dokumentálás
- élesítés
- monitorozás
- kiértékelés
- marketing

Így pedig a fejlesztés és az üzemeltetés szétválasztása a szervezet méretétől függetlenül a csapat méretét figyelembe véve túl költséges lenne.

A felhőalapú szolgáltatások.
A felhőalapú szolgáltatások előretörésével, az infrastruktúra alapú () szolgáltatások tekintetében valamelyest, de a platform alapúaknál () teljességgel kijelenthető, hogy az üzemeltetési feladatok csökkennek és egy kisebb csapatban nincs szükség dedikált személyre. Természetesen ez nem jelenti, hogy kevesebb a feladat, csupán a

feladatok alakulnak át, hiszen egy-egy számítási egység megfeleltethető egy-egy alkalmazásnak, amelynek köszönhetően a hosztok provizionálása szükségtelen, csupán az alkalmazások provizionálására van szükség.

Transzparencia a fejlesztés és az üzemeltetés területén.

> DevOps is the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support.
>
> Mueller (*What Is DevOps?*)

Mueller megfogalmazása alapján könnyen belátható, hogy a fejlesztők és az üzemeltetés feladatai gyakran összekapcsolódnak, kiegészítik, fedik egymást, tehát érdemes szervezeti szinten is jelezni a két feladatkör kapcsolatát a DevOps feladatkör használatával. Továbbá a két munkakör mélyebb integrációjával a fejlesztés állapotai (development, staging, production) érthetőbbé válnak az érintettek számára, ugyanis míg az üzemeltetés a fejlesztés első lépéseit nem látja, így automatizálni, optimalizálni sem tudja, illetve közvetlenül nem tud segítséget nyújtani az új eszközök kiválasztásában, addig a fejlesztők az éles környezetben felmerülő problémákkal nincsenek tisztában. Viszont ha az üzemeltetés részt vesz a fejlesztésben, megérthetik az egy-egy funkció mögött rejlő döntéseket, megismerhetik az alkalmazás gyenge pontjait és hiba esetén sokkal hamarabb tudják lokalizálni a problémát vagy akár még a fejlesztési szakaszban kijavításra kerülhet a hiba. A fejlesztők megérthetik az infrastruktúra határait, a konfiguráció menedzsmentet és mindezek hatásait az alkalmazásra.
A DevOps irányzat akkor tud a legsikeresebb lenni, ha a szervezet illetve a szervezet tagjai nem egy új szereplőként tekintenek rá hanem a szervezeti kultúrába integrálják, az ugyanis nem más mint a tudásmegosztásnak egy olyan formája, amely mind az egyéni-, mind a szervezeti érdekeket is szolgálja.

4.2. Szerepe

A DevOps munkakör szerepe a visszajelzések felgyorsulásával párhuzamosan került elő. A 2 fejezetben bemutatásra került a folyamatos integráció, a tesztvezérelt

fejlesztés fontossága, amelyek sokkal mélyebb architekturális tudást feltételeznek, mint amire egy klasszikus programozónak szükséges van. De ez szintén igaz, a 3 fejezetben bemutatott folyamatos értesítésekre is, hiszen az szoftver folyamatosan tesztelve van, folyamatosan naplóbejegyzéseket küld, melyeknek a megértéséhez az egész rendszert felépítésével kell tisztában lenni. Természetesen ez a fajta munkakör kialakulása amellett, hogy egyesíti a fejlesztők és a üzemeltetők tudását és ahhoz vezet, hogy a felelősségi körük is összeadódik.
Míg a megnövekedett felelősség távol tarthatja a szervezet tagjait a DevOps munkakörtől, addig mind a szoftver, mind a szervezet nagyon sokat tanulhat és profitálhat az egyesített munkakör okozta horizontális látásmódtól.

Konfiguráció menedzsment.
A konfiguráció menedzsment, illetve az automatizálás a DevOps munkakör leggyakrabban emlegett előnyei. Mivel a DevOps szakemberek mind a fejlesztési-, mind a tesztelési, mind a produkciós környezetben felmerülő problémákkal tisztában vannak, viszont különböző környezetekben megszerzett tudásukat könnyen át tudják transzformálni egy másik környezetre, ezért olyan pontokon is tudnak automatizálást végezni, mely nagyban növelheti a hatékonyságot és magabiztosságot. Ilyen például a környezetfüggetlen konfiguráció menedzsment, mely a fejlesztési környezetben munkát végzők napi rutinjait egyszerűsíti és a tesztelési lehetőségeiket növeli.

Incidens menedzsment.
Az incidensek kezelése mindig nehéz dolog, főleg olyan esetekben, amikor nehéz meghatározni, hogy a szoftverrendszer mely részegysége okozza problémát. A probléma megoldása akkor tud főleg elhúzódni, ha az incidens kezelő rendelkezik vakfoltokkal a rendszert illetően ezért a hiba lokalizálása is problémát okoz. Viszont a rendszer részegységeivel, mind a működtetett kóddal tisztában lévő emberhez kerül egy incidens nagyobb valószínűséggel fogja a hibát okát megtalálni, és akár ki is javítani.

Jobban integrált folyamatok.
Mivel a klasszikus üzemeltetés nem igényel részvételt a egy-egy termékcsapat életében, ezért a termékcsapatok olyan eszközökhöz nyúlhatnak, amelyek működtetése az üzemeltetés számára ismeretlen, amely konfliktusforrás lehet a két érintett csoport között. Viszont ha kiválasztási folyamatban szereplők dolgoznak üzemeltetésen és termékcsapatban is, a szoftverkomponens cégen belüli bevezetése könnyebb lesz.

5. fejezet

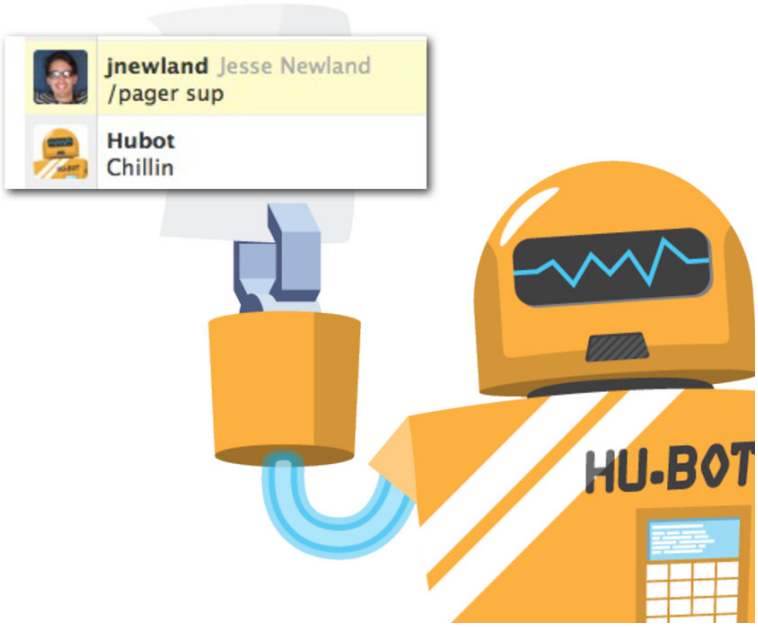
ChatOps

Jelen dolgozat tárgyalja annak szükségességét, hogy az alkalmazás integritása állandóan tesztelve legyen (continuous integration), melynek a segítségével a szervezet tagjai állandóan visszajelzést kaphatnak az éppen folyamatban lévő fejlesztések, javítások állapotáról. Továbbá a DevOps irány, mellyel a visszajelzés, a hibajavítás és önmagában a reakcióidő csökkenthető valamint a keresztfunkciós csapatok meglétével a csapatok horizontális skálázása válik lehetővé. Viszont az nem került tisztázásra, hogy mindezek hogyan kapcsolódnak össze, hogy válnak valójában elérhetővé és miképpen lehet rájuk minél gyorsabban, transzparensebben reagálni. Ez a fejezet erre hivatott válaszolni, mégpedig a ChatOps irányzattal, amely az angol Chat (beszélgetés) és Ops (operations, üzemeltetés) szavak összevonásából keletkezett.

5.1. Kialakulása

Mind a ChatOps név, mind pedig az első implementáció a GitHub cégnek köszönhető, a cég indulása idején 2006-ban már minden kommunikáció az egyik népszerű chat szolgáltatás segítségével (Campfire) történt és a cég ahogy növekedett egyre több információra volt szükség egy-egy csapatnak a többi csapattól, illetve az üzleti elemzők, a marketing adatigénye azt okozta, hogy a chaten folyamatosan írtak a fejlesztőknek és az adatbázis hozzáféréssel rendelkező kollégáknak az igényükkel, amelyet a termék fejlődése érdekében persze kielégítettek (Newland, 2013). Viszont az újra és újra előkerülő hasonló adatigénylés, amely akár csak annyi volt, hogy működik-e a weboldal éppen Japánban, sikerült-e élesíteni a kereséshez kapcsoló hibákat vagy az előző napon fizető felhasználók számosságának kérdese. Ez azt eredményezte, hogy megpróbálták a fejlesztők automatizálni az adatexportálást, azonban ezt nem ad-hoc szkriptekkel tették meg, hanem egy robottal, ami figyel a chaten történő beszélgetésekre és tud rájuk reagálni. Ennek az iránynak több előnye is volt, ugyanis az első adatigénylés után a lekérdezés automatizálásra került, a robot már végre tudta hajtani a lekérdezést, így a fejlesztő kiadta először a parancsot, amelynek hatására a kívánt eredmény láthatóvá vált a chaten azonnal és mindezt látta az is akinek az információra szüksége volt, így amikor legközelebb ismét felmerült az adatigénye már nem kellett kérdezni, hiszen ő maga is végre tudta hajtani az utasítást. Az 5.1 ábrán látható, ahogy a Campfire chatrendszerben lekérdezésre kerül az incidensek állapota, a robot válasza azt jelenti, hogy a jelenleg nincs semmi incidens, tehát minden rendszer megfelelően működik.

A másik fontos irány ami megalapozta a ChatOps létrejöttét az maga az Ops része a szónak, az üzemeltetés. GitHub már kezdetek óta a DevOps filozófiát követi, amelynek köszönhetően egy-egy termékeket több csapat is fejleszt, a csapatok tagjai folyamatosan fluktuálódnak, keverdnek. 2014-ben egy átlagos héten a fő terméken 67 ember dolgozott (Holman, 2014b), amely azt jelenti, hogy a hét bármely időszakában legalább 67 ember állíthatta élesbe terméket, ugyanis nem csak teljesen kész fejlesztések kerülhetnek élesítésre, köszönhetően a GitHub által is aktívan alkalmazott funkció kapcsolóknak (feature flag) köszönhetően (Holman, 2014a). Viszont ahhoz, hogy bárki bármikor élesíthessen egy kész vagy akár csak prototípus szintjén lévő fejlesztést teljes transzparencia szükséges, minden résztvevőnek látnia kell, hogy éppen valaki élesít vagy valaki már foglalkozik egy élesítés okozta problémával.
Az 5.2 ábrán látható, ahogy az élesítés utasítás kiadásra kerül, a jelen esetben egy speciális verzió (smoke-perf) az éles szerverek egy részére (fe1) kerül ki, ez a chatben mindenki számára látható és egyértelmű hiszen csak az angolt használják a robot vezérlésére.



5.1. ábra. Incidensek lekérdezése Hubot segítségével, forrás: Newland, 2013, p. 83

Az 5.3 ábrán viszont már látható, amint egy elrontott élesítés után értesíteni kell az érintetteket, hogy a művelet sikertelen volt, ezért vissza kell vonni, ennek a parancsnak a nagyszerűsége az, hogy nem csak a chaten aktív emberek látják, hanem frissíti a szolgáltatás publikus státusz oldalát is (`https://status.github.com`).

5.2. ábra. Élesítés robot segítségével, forrás: Newland, 2013, p. 40

5.3. ábra. Értesítés rossz élesítésről, forrás: Newland, 2013, p. 48

```
@hubot payment incident? on last week
```

5.4. ábra. Az előző heti incidensek listázása a fizetési rendszerhez

Robot szerepe.
A chatrendszerbe a robot úgy csatlakozik ahogyan egy ember is tenné, minden üzenet megérkezik hozzá és ezekre reagál. A robot önmagában nem tartalmaz üzleti logikát, mindössze a felhasználói üzenetek alapján képes programozható interfészekhez (API) hívásokat intézni.

5.2. Tulajdonságai

Jobb kommunikáció.
Egy csapatban sok különböző szereplő lehet, amennyiben a csapat agilisan fejleszt akkor feltehetően lesznek technikai és nem technikai emberek, akik annak ellenére, hogy egy 5-6 fős csapatban dolgoznak eltérő információval rendelkeznek. Egy nagyon egyszerű példa, ha a csapat egy olyan termékért felelős, amelyben lehet fizetni, viszont a fizetési szolgáltatás egy hibája, komoly csökkenést fog okozni a csapat által fejlesztett szoftver sikerességi riportjában, viszont egy másik szolgáltatás vagy szoftverkomponens állapota könnyen ellenőrizhető egy ChatOps alapú szervezetben, az 5.4 példa ezt hivatott bemutatni.

Egy másik gyakran előforduló probléma az éjszakai leállások, amikor az ügyeletes akár reggelig a probléma javításán dolgozott és reggel még munkaidő kezdetére

nem ért be, akkor könnyen látható, hogy meddig próbálta megjavítani a hibás részt illetve, hogy mit tett a javítás elvégzése érdekében.

Transzparens.

Az előző pontban kifejtett okok is részben a transzparenciát segítik elő, de a chat rendszer használata már önállóan segíti az átláthatóságot. Egy szervezet általában sok formáját használja a kommunikációnak:

- szöveges üzenetek
- tennivalók listája / issue kezelő rendszerek
- hibabejelentő rendszerek
- telefonhívások
- videóhívások
- email
- sárga cetli

Azonban a sok eszköznek köszönhetően biztosan kimaradnak olyan érintettek, akiknek szükségük lenne információra, viszont ha mindezt egy könnyen elérhető felületen keresztül centralizálják, akkor mindenkinek könnyebbé válik az információhoz való hozzáférés. Az 5.5 példa kívánja bemutatni, hogy a chat mint munkaszervezési platform miért tud jól működni, ha a példában látható Joey és Paul ezt a beszélgetést telefonon vagy emailben ejti meg, akkor Bill semmiképp sem tudott volna csatlakozni a beszélgetéshez, és az implementáció csak egy későbbi szakaszában derült volna fény a Bill által említett problémára.

Természetesen a centralizálás nem azt jelenti, hogy minden beszélgetést a chaten kell megejteni, hiszen felesleges lenne ennyire központosítani mindent és előbb vagy utóbb feltehetően munkavállalói ellenállás köszönhetően megszűnne a chat, de az 5.5 példánál maradva, a videóbeszélgetés ugyan nem elérhető chaten, viszont a beszélgetés arról, hogy keresés működésének megbeszélése videóbeszélgetés formájában történt meg az igen, illetve a link a videóbeszélgetéshez (amely valószínűleg támogatja a videóbeszélgetések rögzítését) elérhető. Ez pedig a chat alapú munkavégzés legnagyobb előnye, a kronológikus áttekinthetőség, a

```
Joey: @coworker2: We should talk about the new search feature!
Paul: Sure, a quick videocall?
Joey: Sure
Paul: hubot videocall
hubot: Click here to join the video call
       http://videocall.me/conversation/8046741f-1d67
Bill: hey @Joey, @Paul I will join too because I'm going to implement it
      and there are a few things in the specification which won't work :(
```

5.5. ábra. Egy elképzelt beszélgetés, mely jól személteti, hogy a beszélgetéshez, amely elvileg csak Paul és Joey között zajlott volna, még csatlakozott Bill is

kereshetősége a korábbi beszélgetésnek, bár ez igaz az emailekre is, viszont itt egy csapat minden tagja megtekintheti, nem csak azok akik rajta voltak a címzett listán.

Egy interfész.

A legtöbb szervezet életében kerülnek bevezetésre új szoftverek, melyeknek használata hosszú betanítási folyamatot igényel, illetve ha nehezen illeszkedik a szervezeti kultúrába akkor gyakran a munkavállalók ellenségesen állhatnak a bevezetéssel szemben. Egy nagyon egyszerű példán lehetne ezt szemléltetni, ha be szeretne vezetni a szervezet egy munkaidő nyilvántartó szoftvert, akkor annak feltehetően a kitöltéséhez a következő lépéseket végre kell hajtani:

Authentikáció

Amennyiben az adott szoftver támogatja, illeszkednie kell a szervezet authenkációjához (LDAP, OpenID), amennyiben ez nem lehetséges abban az esetben a felhasználóknak egy újabb jelszót is meg kell jegyezniük, amely bizonyára nem fogja gyorsítani a bevezetést.

Érkezés vagy távozás kiválasztása

Az érkezés vagy távozás kiválasztása

Idő megadása

Az érkezési vagy távozási idő megadása, esetleg további egyéb megjegyzés kíséretében.

```
Bill (08:32): hubot arrived
...
Bill (17:34): hubot leaving for my daughter's violin concert
```

5.6. ábra. Munkaidő nyilvántartás chaten keresztül robottal

```
Steve (08:32): <answering question>
Steve (08:35): <asking question>
Steve (08:43): <answering question>
Steve (09:02): <answering question>
hubot (09:02): Hey @Steve it seems you've been here for 30 minutes
               but you haven't checked in yet, shall you?
Steve (09:02): hubot arrived at 08:32
```

5.7. ábra. A munkaidő nyilvántartás bevezetése sokkal felhasználóbarátabb tud lenni egy robot segítségével

Kijelentkezés

Ezzel szemben az 5.6 ábrán látható egy elképzelt munkaidő nyilvántartás robottal chat rendszerben történő rögzítése, könnyen belátható, hogy a robottal történő adminisztrálás sokkal gyorsabb és egyszerűbb, mint egy különálló felületen.

Természetesen ennek a bevezetése is sokáig eltarthat, hiszen lehet, hogy a szervezet tagjai chaten sem fogják elküldeni az üzenetet a robotnak arról, hogy megérkeztek, mint ahogy a célszoftverbe se lépnének be, viszont míg a be nem jelentkezés esetén a szoftver esetleg emailt tud küldeni vagy a felettest tudja értesíti addig a robot sokkal jobb felhasználói élmény biztosítása mellett képes ezt megtenni, az 5.7 ábrán ezt lehet megtekinteni. Mivel a chat a kommunikáció központi forrása, ezért a robotnak egyszerűen a felhasználók cselekvéseit kell figyelemmel követnie és azokra reagálnia.

Ezen túl egyre több szervezetben a szervezet tagjai már nem csak asztali eszközökön kapcsolódnak az internet, hanem több eszközről. Ilyen például a BYOD (Bring Your Own Device) irányzat amellyel azokat a céges szabályokat illetik melyek megengedik, hogy a munkavállalók saját személyes eszközeiket használják a

munkavégzéshez, azonban ez lehet:

- asztali számítógép
- notebook
- mobiltelefon
- okosóra
- tablet
- könyvolvasó

A lista hosszának csak a képzelet, illetve az elérhető internet képes eszközök számossága szab határt és ezt a számot csak többszörözi az a tény, hogy ezek az eszközök különböző operációs rendszerek különböző verzióit futtatják. Ezek alapján is könnyen belátható, hogy mennyivel egyszerűbb egy szöveges kommunikációra szakosodott chat rendszert használni, mint egy saját felhasználói interfésszel rendelkező rendszert bevezetni, amely feltehetően nem is fogja támogatja az elérhető eszközök töredékét sem.

Természetesen nem minden szoftver helyettesíthető mindössze egy pár szöveges utasítással azonban érdemes belegondolni, hogy sok kis adminisztrációs rendszer sokkal felhasználóbarátabb és akár erőforrásmegtakarítóbb lenne, ha csupán néhány parancs kiadásával lehetne irányítani ahelyett, hogy egy komplex felületen keresztül lehet őket elérni, amely ráadásul csak az asztali gépen meghatározott böngészőből érhető el.

Absztrakció.
Az absztrakció építése mindig költséges, de a szervezet jövőbeni lehetőségeit nagyban növeli. Ha a fent említett munkaidőszervezési példánál maradva a szervezet a szoftver mellett dönt és egy év után úgy dönt, hogy a munkaidőnyilvántartás szükséges, viszont a használt szoftvert lecseréli egy jobb piaci ajánlatnak köszönhetően, akkor a migráció költsége a kisebb tényező lesz ahhoz képest, amelyet a munkavállalókban okozott frusztráció fog okozni a átállást bejelentését követően. Viszont ha a munkaidőnyilvántartás chaten keresztül kerül megoldásra, a háttérben lévő implementáció vagy szoftver könnyen cserélhető, hiszen dolgozók előtt rejtve volt eddig is, ezért

nem is fogják észrevenni, hogy mi történt a háttérben (a használatához szükséges parancsok módosítása nem szükséges).

Aszinkron kommunikáció.

A kommunikációt két részre lehet tagolni a résztvevő felek időbeosztása alapján: szinkron és aszinkron. A szinkron kommunikáció során a résztvevő feleknek az időbeosztása meg kell, hogy egyezzen, azaz az egyik résztvevő fél kérdésére a másik résztvevő fél azonnal válaszol, tehát a válasz megérkezésének késésétől el lehet tekinteni, azaz nullának tekinthető. A kérdező résztvevő azonnali válaszadásra számít a válaszoló részéről, így működik a szóbeli beszélgetés. Ezzel ellentétben az aszinkron kommunikáció során a résztvevő felek nem számítanak azonnali reakcióra, ilyen kommunikációra példa az email, amely elküldése után a küldő tisztában van vele, hogy a válasz megérkezése és az elküldés között eltelt idő több mint nulla lesz.

Szinkron kommunikáció jellemzői:

- a résztvevő felek időben egyszerre vesznek részt a kommunikációban
- gyors módja a kommunikációnak
- rövid reakcióidő
- erős egymásrautaltásg
- nagy információáramlás
- párhuzamosan csak egy kommunikációban lehet részt venni
- jellemző típusai:
  - személyes találkozás (több ember esetén megbeszélés)
  - hívás (videó, hang)

A szinkron kommunikáció jellemzően gyors, pörgős, információgazdag, viszont cserébe nagy koncentrációt igényel az idő egy bizonyos pontjától (kommunikáció kezdete) egy másik pontjáig (kommunikáció vége). Egy szervezet esetében ezek a meetingek, szemtől szembe lezajló beszélgetések, brainstorming vagy akár videó- vagy audióhívások lehetnek, egyetemen a tanórák, a csoportos projektfeladatok

jellemzően a szinkron kommunikációt használják.

Aszinkron kommunikáció jellemzői:

- a résztvevő felek eltérő időben vesznek részt a kommunikációban
- lassú módja a kommunikációnak
- hosszú reakcióidő
- párhuzamosan több kommunikációban is részt lehet venni
- elosztott kommunikáció
- közvetett egymásrautaltság
- alacsonyabb szintű információáramlás
- jellemző típusai:
  - email
  - SMS
  - chat üzenet
  - levél

Az aszinkron kommunikáció jóval lassabb, mint a szinkron, viszont időben nincsenek a résztvevők megkötve, amely természetesen azt jelenti, hogy a reakcióidő megnőhet, illetve lassabb lehet a kommunikáció, csökkenhet a hordozott információmennyiség.

Az előző felsorolás alapján könnyen látható, hogy a szinkron kommunikáció sokkal hatékonyabb és gyorsabb, mint aszinkron társa, azonban sok esetben szinkron nem megvalósítható illetve nagyobb erőforrás igénnyel rendelkezik mint az aszikron:

Földrajzilag elosztott csapatok esetén

Földrajzilag elosztott csapatok vagy csak ideiglesen eltérő időzónában lévő csapatok esetében a szinkron kommunikáció gyakran nem lehetséges, hiszen 9 órás időeltolódás esetén a munkavégzést megnehezíti ha egymásra kell várni.

Más feladat végzése közben

A szinkron kommunikáció hatékonyságával vetekedni valóban nem lehet, viszont a szinkron kommunikáció azt jelenti, hogy a kommunikáció megkezdésekor végzett feladatot meg kell szakítani (legalább ideiglenesen), kommunikálni kell, majd a kommunikáció végeztével lehet folytatni a korábban végzett feladatot, amely könnyen belátható, hogy sok megszakításnál a feladat el nem végzésébe torkollhat.

Információ hiányában

Amennyiben a kommunikációt kezdeményező félnek a kérdésére vagy adatigénylésére a válasz hosszabb ideig tart, ez akár lehet pár perc, óra, napok vagy hónapok, akkor az aszinkron kommunikációnak nincs alternatívája.

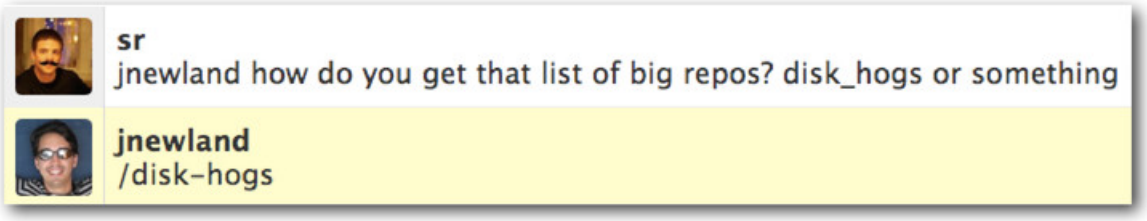
Minden feladat nem végezhető csak szinkron vagy aszinkron módon, ezért érdemes a két kommunikációs formát keverni és egy megfelelő mixet kialakítani, amelyet hozzá lehet illeszteni a szervezeti kultúrához. A túlzásba vitt szinkron kommunikáció azt jelenti, hogy a szervezet rengeteget fog kommunikálni, viszont a feladatok nem lesznek elvégezve a sok megszakításnak köszönhetően. Amennyiben az aszinkron irány erősödik meg, akkor a döntések meghozatala fog lelassulni vagy információhiányos állapotban kerülnek a döntések meghozatalra.

Ha a szervezeten belüli szinkron kommunikációt (telefonhívás, videóhívások) sikerül minimalizálni, akkor lehet növelni a szervezet tagjainak a terhelhetőségét és minimalizálni az állandóan kontextusváltás okozta frusztrációt, miközben minden kérés dokumentálásra kerül a chatrendszernek köszönhetően.

Tanulva dolgozás - Learning by Doing.

A szervezet felépítésének, mindennapjainak komplexitását jól szemlélteti egy új tag belépése a szervezetbe, akinek meg kell mutatni a munkavégzést, majd utána az egyedüli használat során rutin műveletté kell transzformálnia azt, ez mindenki számára nagy stresszor. Hasonló problémás lehet, mint a már korábban említett új szoftver bevezetése. Ott bemutatásra is került, hogy miért felhasználóbarátabb egy szöveges üzenetekkel elérhető szoftver.

Az 5.8 képen látható beszélgetés a GitHub kívánja bemutatni, hogyan tanulnak az emberek miközben végzik a munkájukat. Aki nem ért valamit, kérdezhet és miközben megkapja a választ, már az eredményt is láthatja, és ha később elfelejtené akkor bármikor visszalapozhat az első napján született beszélgetések naplójához, amikor



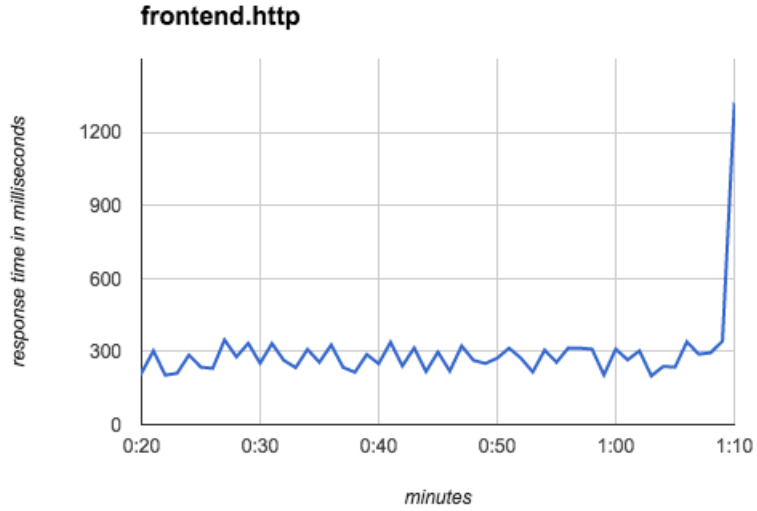
5.8. ábra. Tanulás kommunikáció közben, forrás: Newland, 2013, p. 65

megmutatták neki a parancs működését. Továbbá a kereső bárki számára elérhető és mindenkinek az üzenete elérhető, ezért megnézhető, hogy ki milyen parancsokat használ egy-egy üzenet elvégzésére.

Incidens menedzsment új szintje.
Az incidens menedzsment egy nehéz dolog, ugyanis gyakran éjszaka kell reagálni a felbukkanó hibákra a Runbook által meghatározott módon. A ChatOps előnyeinek felsorolása közül nem véletlen kerül utoljára tárgyalásra ez a fontos terület, ugyanis ebben egyesül a legtöbb pozitív tulajdonsága.
Az incidensek megfelelő kezeléséhez szükségesek a következők:

- kezelnie kell az incidens hibajegyét
- a lehető legtöbb információra van igény
- az ügyeletet végzőnek tudnia kell értesíteni a felelős csapatot
- dokumentálni kell minden elvégzett műveletet
- figyelni kell a metrikák (hibaszázalék, válaszidő) változását
- küldeni kell értesítést a státusz oldalra
- követni kell a Runbook utasításait
- ha nem sikerül elhárítani a hibát az ügyelet lejártáig, tájékoztatnia kell a következő ügyeletest

```
hubot (1:12): Failure detected in frontend, alert created with id 1337:
              http://yourduty.com/incident/frontend/1337, @Brian has
              been alerted
Brian (1:12): hubot pager ack 1337
hubot (1:13): Incident 1337 has been acknowledged by Brian
Brian (1:14): hubot graph frontend.http @-1h
hubot (1:14):
```




```
Brian (1:15): Something broken for sure, notification should be sent
Brian (1:17): hubot status yellow for frontend
hubot (1:17): Status has been set to yellow and team has been notified
Brian (1:18): Let's see who deployed last time, they rolled
              out something new
Brian (1:18): hubot deploy frontend
hubot (1:18): Last deploy for frontend was 2 days
Brian (1:18): Nothing happened here, so no new code,
              but something is broken, interesting
Brian (1:24): hubot graph frontend.exception-rate @-1h
....
hubot (3:00): Oncall rotation has been changed
hubot (3:00): Incident 1337 is still active,
              @Simon has been notified
Simon (3:00): Popping in, @Brian you can get some sleep
Brian (3:00): Thanks @Simon, I tried my best, the error is still on
```

5.9. ábra. Incidens kezelés ChatOps használatával

Az 5.9 beszélgetés jól szemlélteti, hogy a chat segítségével mennyivel könnyebben végre lehet hajtani az elvégzendő a feladatokat.

- Az értesítés kiküldése után, az incidens kezelése már a robot segítségével elvégezhető.
- Az információ összegyűjtése és a metrikák könnyen elérhetőek, mert a grafikonok és a legutóbbi élesítés dátuma lekérdezhetőek egyszerűen chaten keresztül.
- A felelős csapat értesítése és a státuszoldal frissítése is a korábban bemutatott módon elvégezhető.
- Minden lépés dokumentálva van, ugyanis minden beírt parancs látható mindenki számára (és naplózásra kerül).
- A Runbook utasításainak követése is megoldható a robot segítségével, vagy akár a kereső használatával könnyen kideríthető, hogy volt-e már korábban ilyen probléma akár ennél az alkalmazásnál vagy másiknál. Amennyiben volt, az ott használt parancsok használatával is el lehet kezdeni a hibajavítást.
- A hiba meg nem javítása esetén nem szükséges felvilágosítani a következő ügyfeletest, ugyanis az előzmények megtekintésével gyorsan világossá válhat számára is, hogy milyen műveletek lettek elvégezve a hiba megjelenése óta.

Ahhoz, hogy az incidens menedzsment ennyire automatizáltan tudjon működni és ki lehessen használni a chat javára felsorolt előnyöket ahhoz erős szervezeti támogatottságra van szükség és a chaten található adatok kell használni, mint az információ egyetlen forrása. Ezen felül minden rendszernek (incidens menedzsment szoftver, monitorozó rendszer, metrika/napló rendszer) rendelkeznie kell programozható interfésszel, melyen keresztül a robot képes az információkhoz hozzáférést biztosítani. Ebben a részben nem került tárgyalásra, de mindennek a működéséhez szükséges, hogy az alkalmazásokból a megfelelő adatok bekötésre kerülnek, a hibák a központi naplózó rendszerbe bekerülnek, a monitorozható végpontok nem fals negatív adatokat szolgáltatnak. Ezeknek a megfelelő működése a fejlesztési csapat felelőssége.

6. fejezet

A szervezeti integráció

6.1. Áttekintés

Ahhoz, hogy egy szolgáltatás vagy egy alkalmazás (hosszú távon) működtethető legyen, bemutatásra kerültek a szükséges részek:

Folyamatos adatgyűjtés és értesítés

Az alkalmazás és a környezetének adatainak és metrikáinak folyamatos gyűjtése, a jó és a jól működésre való törekedés végett.

Gyűjtött adatok elemzése

A nagymennyiségű adatgyűjtés miatt, a zaj csökkentése érdekében cselekvésre sarkalló elemzésre van szükség.

Automatizált folyamatok

Az automatizálás segítségével a hibák száma minimalizálható, a feladatok újra és újra megismételhetőek, hordozhatókká, újra felhasználható válnak növelve a tudásátadás lehetőségét.

A rendszerek közti kapcsolatok megértése

A szoftverfejlesztő látókör kibővítése, a működtetéshez szükséges környezet való hozzáférés csökkenti a reakcióidőt, a szakmai döntésekhez szükséges idő rövidül, gördülékenyebb lesz.

Transzparens kommunikáció

Transzparens kommunikáció, melynek segítségével a szervezet tagjai közvetlenül és azonnal elérhetőek, reszponzívabb eszmecserét és visszakereshető kronológiai tudástárat eredményez.

A szervezet célja nem az, hogy mindenki értsen mindenhez hanem, hogy az egyénileg és csoportosan birtokolt tudás megosztása, minél egyszerűbb legyen és mindez minél kevesebb kontextusváltást igényeljen.

Amennyiben a szervezet a 6.1 listában szereplő elemeket implementálja, akkor rendkívül fontos, hogy az ott elérhető adatok minél egyszerűbben elérhetőek legyenek. Ennek a legjobb módja, ha minden alkalmazás (dokumentáció-, verziókezelő rendszer) nem csak saját felülettel, hanem API-val is rendelkezik, ennek a fontosságát jelzi, hogy az Egyesült Királyság kormánya ennek külön fejezetet szentelt a szolgáltatás kézikönyvében (*Government Service Design Manual. APIs - Using and creating*

Application Programming Interfaces), amelynek minden az állam által előállított szoftvernek meg kell felelnie. Ennek köszönhetően az elosztottan tárolt adatok egy központi helyre tudnak bekerülni, viszont a lényeg nem az, hogy egy alkalmazás által nyújtott szolgáltatások a programozható interfész segítségével egy másik felületen megvalósíthatóak legyenek, hanem:

- más alkalmazással való integráció elősegítése
- előzetes bepillantás lehetősége az adatokhoz
- a műveletek egy-egy fázisának kiszervezése
- a változásokról értesítés küldés

A listában szereplő pontok kerültek szemléltetésre a 6.1 példában, itt egy szándékosan nem technikai feladattal kerül bemutatásra, ugyanis egy szolgáltatás működtetéséhez nem csak technikai problémákkal kell megküzdeni, hanem sok kommunikáció nem technikai-technikai emberek között zajlik. A beszélgetés Emily kérdésével kezdődik, aki azt szeretné tudni Gavintől, hogy előkészítette-e a számlát a példában szereplő cégnek, Gavin fejből nem tudja, viszont ahelyett, hogy megnyitná a számlázó programot, azonnal a beszélgetésből ellenőrizni tudja, hogy a számla már meg lett-e írva (más alkalmazással való integráció elősegítése). Érdemes megfigyelni, hogy a listázott számláknak nem csak az azonosító száma kerül listázásra, amely kattintható elemként jelenik meg, tehát azonnal megtekinthető egy klikkeléssel a számlázórendszerben. Hanem tartalmazza a legfontosabb információkat is (előzetes bepillantás lehetősége az adatokhoz).
Miután Gavin meggyőződik róla, hogy a számla nincs a rendszerben, anélkül hogy belépne a számlázó szoftverbe, a beszélgetésből közvetlenül könnyen elkészítheti az új számlát Természetesen nem cél a számla teljes elkészítése, csak az elkezdése, hiszen így a már létrejövő piszkozati számlára már lehet hivatkozni azonosítóval (a műveletek egy-egy fázisának kiszervezése).
Az idő előrehaladtával, amikor a számla esedékessége közeledik, a számlázó rendszer üzenetet küldve (például a 3.5 fejezetben bemutatott webhook segítségével) tudja értesíteni, a felelősöket arról, hogy teendő van a számlával. Gavin ezt észlelve, a számla számára klikkelve, a számlázó szoftverbe belépve a számlát be tudja fejezni, amelynek megtörténéséről ismét értesítés kerül küldésre, ezáltal tudatva a többi érintettel az számla állapotváltozását (a változásokról értesítés küldés), mindezt

```
                                2015. 03. 04.
Emily: Hey @Gavin, last week we signed the contract with
       the MicroCompany Ltd. company.
       Have you set up their first invoice with due date 03.31?
Gavin: Good question.
Gavin: @hubot search invoice for microcompany
hubot: There are 2 matches for search "microcompany":
       #32 is for MicroCompany with VOID status,
       its due date is: 2013. 04. 02. (almost 2 years ago)
       #33 is for MicroCompany with PAID status (USD 1300),
       its due date is: 2013. 04. 02. (almost 2 years ago)
Gavin: Oh it seems I haven't drafted the new one.
Gavin: @hubot invoice draft "MicroCompany Ltd." on 2015.03.31.
hubot: #DR98 has been drafted for MicroCompany Ltd. with due date 2015.03.31.
                                2015. 03. 31.
Accounting Software:
       There is one invoice which is dated for today:
       #DR98 for MicroCompany Ltd.
       created by @Gavin 27 days ago
Gavin: Oh almost slipped out of my mind, on it. cc @Emily
Accounting Software:
       Gavin added USD 1500 to #98 and set status to AUTHORISED.
Emily: Thanks @Gavin.
Emily: Fyi @Gavin I've just sent #98 to microcompany
hubot: #98 is an invoice in
AUTHORISED state, with amount USD 1500, its due date is 2015.03.31.
```

6.1. ábra. Szolgáltatás integráció, kontextváltás csökkentése

valós időben.
Érdemes megjegyezni, hogy az érintettek közti kommunikációt nagyban segíteni azáltal, ha valaki egy számlaszámot vagy egy számla linkjét említi a beszélgetésben, akkor a rendszer vagy éppen a robot automatikusan letölti az API-n keresztül a számlaszámhoz tartozó legfontosabb adatokat és megjeleníti azokat.

Érdemes továbbá megfigyelni a 6.1 példában, hogy miközben egy feladat elvégzésre került, a feladatban a többi érintett is folyamatosan értesülhetett az állapotról, így sokkal könnyen be tudtak csatlakozni a beszélgetésbe, meg lehet érteni a beszélgetés kontextusát, gyorsabban lehet reagálni egy-egy kérdésre. Amennyiben a szervezeten belüli központi kommunikáció aszinkron módon zajlik, akkor a kontextusváltások nincsenek ráerőltetve a szervezeti tagokra, hiszen az aktuálisan végzett feladat befejezése után is meg tudják válaszolni a számukra feltett kérdéseket. Végül, ha a központi kommunikációs csatornán keresztül kerülnek automatizálásra egy-egy feladat elvégzéséhez szükséges lépések, akkor az adatokhoz hozzáféréssel rendelkezők köre bővülhet, ezáltal csökkentve egy-egy feladat megoldásának idejét. Természetesen ez nem azt jelenti, hogy a szervezeten belül bárki bármihez hozzáférhet, legyen az a számlázi rendszer, vagy a chatrendszer egy szobája, a szervezetnek meg kell határoznia az adatokhoz való hozzáférhetőséget, legyen az írási vagy olvasási jog, de a dolgozatnak nem célja ennek meghatározása, ugyanis ez a szabályok mindig az adott szervezet kultúrájától, az adatok milyenségétől és az adott iparágtól is erősen függnek.

6.2. Hatásai a szoftverműködésre

A 6.1 fejezetben áttekintésre került a szervezet által használt eszközök integrációja hogyan javíthatja a munkavégzés minőségét. Most pedig megvizsgálásra kerül, hogy mire kell figyelni, egy szoftverműködés során, milyen lépések azok, amelyeknek végre kell hajtani, hogy a szolgáltatás felhasználói számára a működtetés szinte láthatatlan legyen.

Fejlesztési környezet.
A fejlesztés során a 2 és a 3 fejezetben bemutatott lépések a legfontosabbak, azaz

az adott fejlesztő csapat, az projekt menedzser és egyéb belső érintettek folyamatos tájékoztatása és a visszacsatolás idejének csökkentése, többek között a következő folyamatok során:

a kódátnézés (code review) során

A kód átnézés a használt szoftverfejlesztési metodologikától függetlenül fontos szerepet kap a fejlesztés során, ezért mind az átnézésért felelős szakember értesítése, mind a kód megírásáért felelős programozó részéről az igény jelen van a minél egyszerűbb és reszponzívabb kommunikációra, mert ugyanaz a kódrészlet újbóli megértése komoly erőforrásokat emészthet el, továbbá egyik szereplőnek sem jó, ha egy kód nem kerül beolvasztásra a főágba vagy esetleg csak hosszú idő elteltével kerül beolvasztásra.

a kódminőség javítása

A kódátnézés magas erőforrásigénye miatt (az átnézést végzőnek a minél jobb kódminőség elérése, a logikai hibák megtalálása a célja, a kód írójának pedig a saját munkája eredményének a megvédése a célja, ez gyakran szakmai vitához, egyet nem értéshez vezethet, amelynek feloldása hosszabb időt vehet igénybe) csökkenteni kell a kódátnézés iterációit, ezért a kódminőség automatizált ellenőrzésével (szintaktikai hibák, elérhetetlen kódrészletek) kizárhatóak azok a hibák, amelyek biztosan nem mennének át a kódátnézés folyamatán.

folyamatos tesztelés

A folyamatos tesztelés eredménye, amely automatizált teszteseténél a korábban megírt tesztek (egység, regressziós, integrációs) kimenetét és a hibás teszt azonosítóját elküldve értesíti a felelős egyént vagy csoportot. A folyamatos tesztelés másik típusa, a manuális tesztek, melyek érkezhetnek a minőségellenörző csapattól (QA) vagy egy az integrációt végző másik csapattól, ezért fontos, hogy a visszacsatolás minél egyszerűbben megoldható legyen.

automatizálás

Az alkalmazáshoz kapcsoló feladatok közül, nem csak a produkciós környezetben kell az automatizálást komolyan venni, és aktívan használni, hanem már a fejlesztés során is. Automatizálás magába foglalhatja a konfiguráció menedzsmentet (amelyet szintén tesztelni kell, tehát igaz rá minden a fejlesztési környezet minden pontja is), a tesztelést (egység, több környezetben is használható integrációs tesztek), performancia problémák detektálását.

Élesítés.

Az élesítés az a folyamat, amely során a kód a fejlesztési környezetből a produkciós környezetbe kerül, a dolgozat nem különíti el a kódélesítést (deploy) és a funkcióélesítést (release), mert bár a legtöbb szoftverrendszerben ezek elkülönítésre kerülnek, de a felmerülő pontok, amelyekre figyelmet kell fordítani mindkét esetben ugyanúgy jelen vannak. Az élesítés során a legfontosabb lépések, amelyeket figyelembe kell venni a következők:

Konfiguráció menedzsment A konfiguráció menedzsmentről már esett szó a fejlesztési környezet pontjai között, viszont érdemes itt is kiemelni, mert klasszikusan ennél a lépésnél használják, viszont ha két helyen is használatra kerül, az azt jelenti, hogy mindenki magabiztosabban nyúl hozzá, sokkal több kézen fordul meg, sokkal több tapasztalattal fog rendelkezni a szervezet és sokkal kiforrottabbá fog válni a konfiguráció menedzsment.

Folyamatos integráció A fejlesztési környezetnél már erről is szó esett, mégpedig az automizált tesztek és a kódellenőrzés lépeseknél, bár ez így nem lett kijelentve. Tulajdonképpen a folyamatos integráció lépései a kódminőség ellenőrzésével, a tesztek futtatásával kezdődik, míg a fejlesztés során ezek után az eredmény kerül publikálásra, addig az élesítés során egy további plusz lépéssel a kód az éleskörnyezetbe fog kerülni.

Változáskövetés A változáskövetést általában egy változás naplóban (change log) végzik, amely egy projekt megemlítendő változásait tartalmazza válogatott és kronológiailag rendezett formában (*Keep a CHANGELOG*).

Hibadetektálás.

A hibák detektálása az élesítés közbeni és utáni folyamatos feladatként jelentkezik, amelyek az érkező adatfolyamokból kerülnek kiszámításra, ezeknek két típusa van:

- külső
- belső

A belső adatok az alkalmazás által küldött naplóbejegyzésekből és számosítható metrikákból épülnek fel, míg a külső adatok független ellenőrzésekből (működik-e az alkalmazás, alkalmazás válaszidő értéke, megfelelő válasz érkezik-e a kérésre) érkeznek, amelyek nincsenek kapcsolatban az aktuálisan ellenőrzött rendszerrel. Rend-

kívül fontos, hogy a hibák detektálása adaptív legyen, mind a környezetre, mind a rendszer aktuális állapotára. Néhány példa ennek szemléltetésére:

- Egy alkalmazás sebessége függ a felhasználók számától, az aktuális időponttól (egy közösségi oldal életében este 6 és 8 óra között feltehetően magas lesz a felhasználók száma), az aktuális időjárástól (hideg esős idő esetén nőhet a felhasználók száma), ezért eltérően kell tudni kezelni a kedd este 6 órát és a vasárnap hajnali 4 órát, mert míg az egyik időpontban a sebességcsökkenés mindössze az aktív használat növekedést jelenti, addig a másik esetben felhetehetően hiba okozza. Ezetén érdemes több metrika együttesét figyelembe venni, illetve időablakokat alkalmazni.

- Hibák közti priorizálás az ügyeleteseket nem szakítja meg a már korábban jelzett hiba felderítése, illetve megoldása közben. Ilyen lehet, ha egy adatközpont hálózati problémákkal küzd, akkor nem érdemes értesíteni mindenkit arról is (vagy esetleg alacsonyabb priorizálással), hogy az adott adatközpontban futó alkalmazások is leálltak.

- Akcióra kész hibák használata. Annak ellenére, hogy külső adatfolyamból érkező hibák érthetőek, nem jelenti azt, hogy az azokat követő hibajavítás is egyértelmű lenne vagy esetleg meghatározható lenne. Ezért a több forrásból érkező adatfolyamokat (külső ellenőrzés, alkalmazás napló, konténer napló, hiba előidézésében résztvevő egyéb komponensek naplóüzenetei, metrikái) összegyűjtésével megérthető a hibák a kontextusa, a későbbi ismételt előidézhetőség esélye megnő, szükség esetén tesztesettel lefedhetővé válik.

Hibaelhárítás.

A hibaelhárítása a hiba detektálása után kezdődik el, melynek első lépése az incidens kezelő rendszerből érkező értesítés kezelése, majd a beérkező hibához kapcsolódó naplóüzenetek és elérhető grafikonok (hasonlóan, mint az 5.9 példában látható) megtekintése, a hiba okának detektálása, majd az előzetes szabálygyűjtemény (runbook) alapján történő végrehajtás elkezdése. Emellett publikus szolgáltatás vagy végfelhasználót is érintő probléma esetén szokás a szolgáltatás státusz oldalát is frissíteni, hogy reflektálja a fennálló hiba létét, az érintett komponseket és a várható javítás idejét, amennyiben lehetséges. A hiba javítása akár újabb emberek, csoportok bevonását is igényelheti. Gyakran előfordul, hogy a incidens megoldása magának

az alkalmazásnak a módosítását is igényli, amely ilyenkor ismét végigmegy az a fejlesztési környezet és az élesítés lépésein.
A hibajavítás befejeztével az incidens lezárásra kerül, a szolgáltatásért felelősök és amennyiben szükséges az érintett felhasználók is értesítésre kerülnek.

Hiba-utóélet.
Minden hiba után általában két-három nappal később érdemes egy úgynevezett post mortem analízist tartani. A nem közvetlenül a hiba után tartott analízis azért fontos, mert ilyenkor már a hibát övező érzelmek csökkennek (egy éjszaka három órakor érkező riasztás sok negatív érzelemmel fogja eltölteni az ügyeletest), viszont még nem került elfelejtésre a hiba oka és a javítás folyamata. A post mortem analízis egy írásos dokumentummal záródó megbeszélés, amely tartalmazza a hiba körülményeit, a megoldáshoz szükséges akciókat és a későbbi megelőzéshez szükséges lépéseket.
A modern szervezetekben gyakran alkalmazott metodológia a hibáztatásmentes post mortem (blamess post mortem), amely a hibák okozta szolgáltatáskieséseket a szervezet tanulási folyamataként látják, nem pedig a hibás személy kilétét próbálják meg felfedni és megbüntetni (Allspaw, 2012). A felelős személy meghatározása egy elosztott rendszerben szinte lehetetlen, ezért érdemes inkább a post mortem analízist egy tanulási lehetőségnek, mint a büntetés egy formájának tekinteni. Továbbá nagy skálázású alkalmazásoknál incidensek gyakran előfordulnak, ezért az üzemeltetőket érdemes támogatni abban, hogy nyíltan beszéljenek a post mortem alkalmával az általuk elvégzütt lépésekről ezáltal csökkentve az incidensek okozta stresszt.

7. fejezet

Összefoglalás

A szoftverek működtetése komplex dolog, de az adatalapú szemlélettel magabiztosan lehet őket működtetni. A magasfokú automatizálással a szoftverek külső beavatkozás nélkül tesztelhetőek, ellenőrizhetőek így csökkentve a hibázási lehetőséget. Az újrafelhasználással a komplex rendszerek is lebonthatóak apró alkotóelemekre, melyeknek megértése sokkal könnyebb és gyorsabb monolitikus társaiknál. Ez a két alkotóelem, amely meghatározza egy alkalmazás működtethetőségét technikai oldalról.

A másik oldalon viszont a szervezet tagjai vannak és a köztük történő kommunikáció. Az ismétlődő folyamatok automatizálásával csökken a humánerőforrás igénye, viszont a tervezésnél és a problémák megoldásánál nagyfokú kommunikációra és koordinációra van szükség. Ennek a problémának megoldása lehet a cset alapú, szobákra osztott csoportos kommunikáció, amely az aszinkronitásával, az eszköz- és helyfüggetlenségével nagyban hozzájárulhat a szervezetek egy egységes, központi és transzparens információcseréjéhez. Továbbá a központi cset lehet az a gócpontja a szervezetnek, ahová befutnak a külső forrásokból érkező adatok, ahol egy-egy üzenetre az akció azonnal végrehajtható kontextus váltás nélkül, növelve a szervezet tagjainak információellátottságát, megteremtve a tudásmegosztás platformját mindezt visszakereshetően és bárki számára elérhetően. A dolgozat bemutatta ennek a valósidőben történő akcióvégrehajtásnak az előnyét a szoftveres hibák menedzselésére, miként lehet a fejlesztők és az üzemeltetők tudásának keresztezését a szervezet javára fordítani, kitérve arra, hogy a hibautóélet kézelésének igenis a szoftverrendszerek részévé kell válnia és proaktívan kell segíteni a további fejlesztéseket.

Felhasznált irodalom

Allspaw John (máj. 2012). *Blameless PostMortems and a Just Culture.*
Letöltve: 2015. 04. 26. URL: https://codeascraft.com/2012/05/22/blameless-postmortems/.

Continuous Integration (original version) (2002).
Letöltve: 2012. december 25.
URL: http://martinfowler.com/articles/originalContinuousIntegration.html.

Government Service Design Manual. APIs - Using and creating Application Programming Interfaces.
Letöltve: 2015. 04. 25. URL: https://www.gov.uk/service-manual/making-software/apis.html.

Holman Zach (márc. 2014a). *How do you avoid the merge master trap.*
Letöltve: 2015. 04. 19. URL: https://github.com/holman/feedback/issues/519#issuecomment-38099365.

— (okt. 2014b). *How many .com teams does GitHub have?*
Letöltve: 2015. 04. 18. URL: https://github.com/holman/feedback/issues/597#issuecomment-58710920.

Johansen Christian (2011). *Test-Driven JavaScript Development.* Pearson Education, Inc, p. 12.

Keep a CHANGELOG.
Letöltve: 2015. 04. 26. URL: http://keepachangelog.com/.

Lindsay Jeff (máj. 2007). *Web hooks to revolutionize the web.*
Letöltve: 2013. április 2. URL: http://progrium.com/blog/2007/05/03/web-hooks-to-revolutionize-the-web/.

Mueller Ernest. *What Is DevOps?*
Letöltve: 2015. 03. 28. URL: http://theagileadmin.com/what-is-devops/.

Newland Jesse (febr. 2013). *ChatOps at GitHub*.
Letöltve: 2015. 04. 18. URL: `https://speakerdeck.com/jnewland/chatops-at-github`.