Vysoká škola ekonomická v Praze

Fakulta informatiky a statistiky

Katedra informačních technologií

DevOps – konec zavedeným pořádkům vývoje a řízení kvality softwaru?

SEMESTRÁLNÍ PRÁCE

Předmět: 4IT446 – Řízení kvality softwaru
Student: Bc. Luděk Veselý
E-mail: vesl00@vse.cz

2017

Obsah

Obsah **ii**

1 Úvod **1**

2 DevOps jako nový přístup k řízení vývoje a kvality software **2**
2.1 Tradiční způsob vývoje aplikací 2
2.2 Problémy vyskytující se u tradičního přístupu 3
2.3 Propojení vývoje a provozu – DevOps 4

3 Prostředky DevOps **5**
3.1 Architektura mikroslužeb 5
3.1.1 Velikost mikroslužeb 7
3.1.2 Problémy při implementaci mikroslužeb 8
3.2 Kontinuální integrace a nasazování 8
3.2.1 Nástroje pro kontinuální integraci 9
3.2.2 Implementace kontinuální integrace pomocí GitLab CI 10
3.3 Kontejnerizace 12
3.3.1 Docker 13

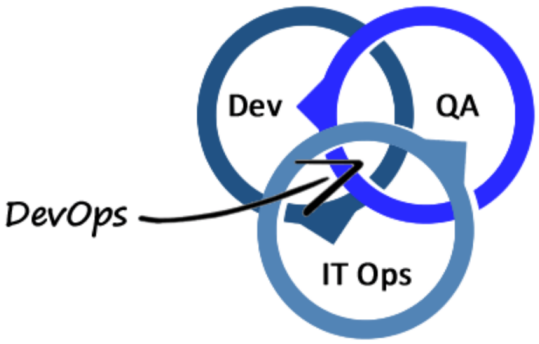
4 Závěr **15**

Literatura **16**

1 Úvod

Vývoj software prošel za svou dobu řadou změn, stále jsou však využívány přístupy, které vznikly v době, kdy byly požadavky na vývoj jiné než v současné době. Nyní je vývoj software rychlejší a dynamičtější, je třeba být schopen rychle reagovat na změny, umět doručit novou funkcionalitu uživateli co nejdříve. IT odvětví se totiž odlišuje od odvětví tradičních, k řízení je tedy třeba přistupovat jinak, než v případě tradičních odvětví. Přestože se i v rámci IT dílčí zaměření dosti odlišují (například požadavky na stabilitu systému jsou jiné u zdravotnického zařízení a jiné u webové fotogalerie), stále se toto odvětví od těch tradičních odlišuje.

Těmto problémům by měl pomoci čelit agilní přístup, který definuje jiné uvažování o vývoji software. Zákazník je v samém centru dění a primární je jeho spokojenost. S tím také souvisí pojem DevOps, který popisuje propojení světů vývojářů, správců a testerů a vede tak k odstranění bariér, které mohou blokovat uvedení software do provozu.



Obrázek 1.1: *Vztah vývoje, provozu, QA a DevOps (Zdroj: [2])*

Tato práce si klade za cíl provést světem DevOps – vysvětlit, proč může být přínosné tento koncept adoptovat a představit technické prostředky, kterými lze tohoto cíle dosáhnout. Cílem této práce není popisovat, co to vlastně DevOps je nebo jaká je jeho historie. Tato práce je spíše praktický průvodce, který čtenáři dodá přehled technik a nástrojů, které mu pomohou v adaptaci tohoto přístupu.

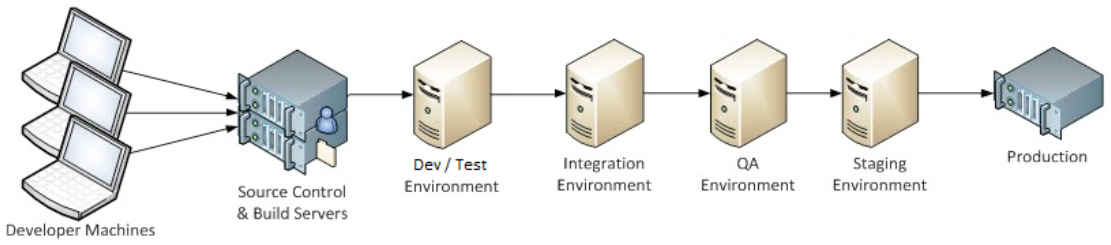
V úvodu práce je vysvětlen stav, kdy je aplikace implementována jako jeden celistvý systém a o její provoz se stará někdo jiný, než ji vyvíjí, což s sebou nese řadu problémů, zejména co se zodpovědnosti týče. Následně je zavedena filozofie DevOps, a vysvětleno, jak může pomoci čelit zmíněným problémům. Následně jsou popisovány vybrané prostředky pomáhající v naplnění DevOps ideálů – architektura mikroslužeb, kontinuální integrace a nasazování, kontejnerizace, monitoring.

2 DevOps jako nový přístup k řízení vývoje a kvality software

Za dobu vývoje a řízení kvality software se procesy s tím související ustálily. Jednotlivé role nebo pozice v IT světě jsou obecně známé včetně náplně jejich práce a souvisejících kompetencí. Jenže tento přístup je praktikován již nějakou dobu a v okamžiku svého vzniku rozhodně nebylo počítáno s bouřlivostí a rychlostí rozvoje IT, jak ji známe dnes. Stále častěji se tak mluví o DevOps jako nástupci zavedených pořádků v řízení vývoje a kvality software. Než bude vysvětleno, v čem tento přístup spočívá, je žádoucí krátce vysvětlit, v čem spočívá tradiční přístup k vývoji a jaká jsou jeho úskalí.

2.1 Tradiční způsob vývoje aplikací

Tradiční způsob řízení vývoje software vycházející z vodopádového modelu se přenáší i na způsob jeho nasazování do produkčního prostředí. Vývojáři (**Developers**) programují kód, který je třeba pro potřeby testování a akceptace zpřístupnit tak, aby byl veřejně dostupný pro danou skupinu uživatelů (testerů nebo zadavatelů). Typicky tak vzniká několik prostředí (vývojové, testovací, produkční nebo akceptační), v nichž je aplikace nasazena v různých verzích a konfiguracích, s různými daty. To s sebou nese problém správy jednotlivých prostředí a následného nasazování aplikace do těchto prostředí. Je to tak prostor pro vznik chyb, navíc je potřeba pracovník (nebo jejich skupina), který se o tato prostředí bude starat – těmto lidem se souhrnně říká **Operations**.



Obrázek 2.1: *Řada prostředí pro manuální schválení software (Zdroj: [3])*

Tento přístup má samozřejmě řadu důvodů, proč je využíván. V prvé řadě se každé činnosti věnuje specialista v daném oboru. Provozu aplikace se věnuje pracov-

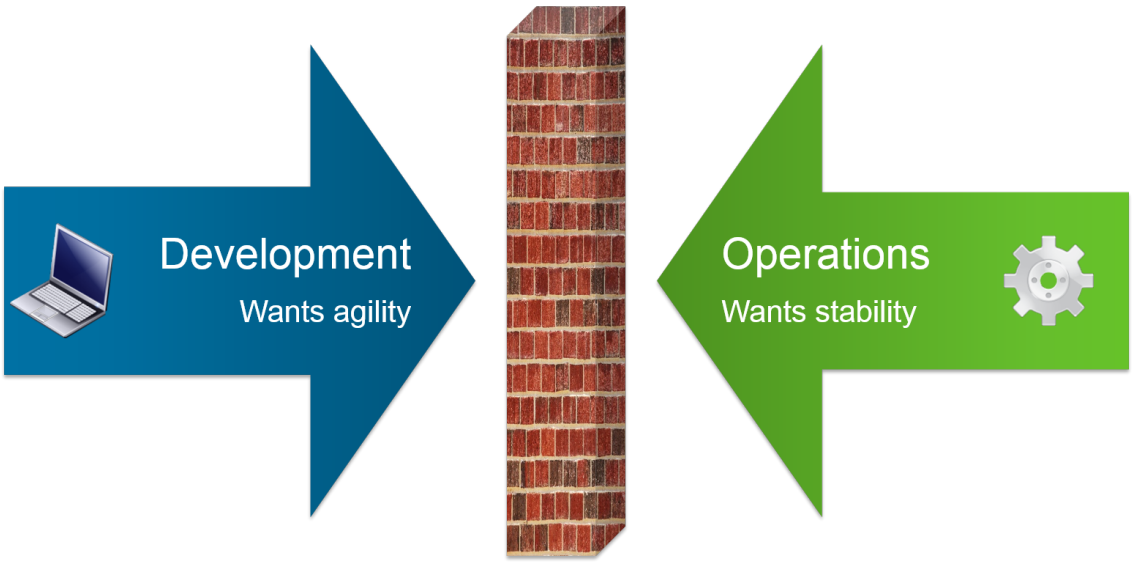
ník, který se může zabývat pouze touto prací a nemusí se zaobírat programováním, může se ve svém oboru více specializovat.

Dále má každý vývojář vždy k dispozici kompletní aplikaci, nemusí tak zprovozňovat více prostředí pro vývoj. Nemusí se ani starat o provoz testovacího prostředí, zjednodušeně řečeno pouze naprogramuje a odevzdá, co je po něm požadováno.

Zároveň je pomocí tohoto přístupu možné dosáhnout pořádku v nasazování v tom smyslu, že je dostatečně dlouhá doba do nasazení změny a při dodržení definovaných procesů je taková změna důsledně testována při jejím schvalování. Přestože se zdá, že takový systém musí bez problémů fungovat, v praxi může být situace jiná. Požadavky na změnu mohou být velmi časté s nutností jejich rychlého zapracování.

2.2 Problémy vyskytující se u tradičního přístupu

Tento přístup k nasazování aplikací se však potýká s řadou problémů. Vzhledem k oddělení prostředí trvá velmi dlouho, než se naprogramovaná změna dostane do produkčního prostředí. Na této cestě je příliš mnoho překážek, jako jsou jednotlivá schvalování a ruční zásahy. Díky tomu zadavatel vidí funkční artefakt až příliš pozdě a musí si tak umět včas představit, co by měl vyvíjený software umět, což je v případě IT projektů nesmírně obtížné [1, strana 73].



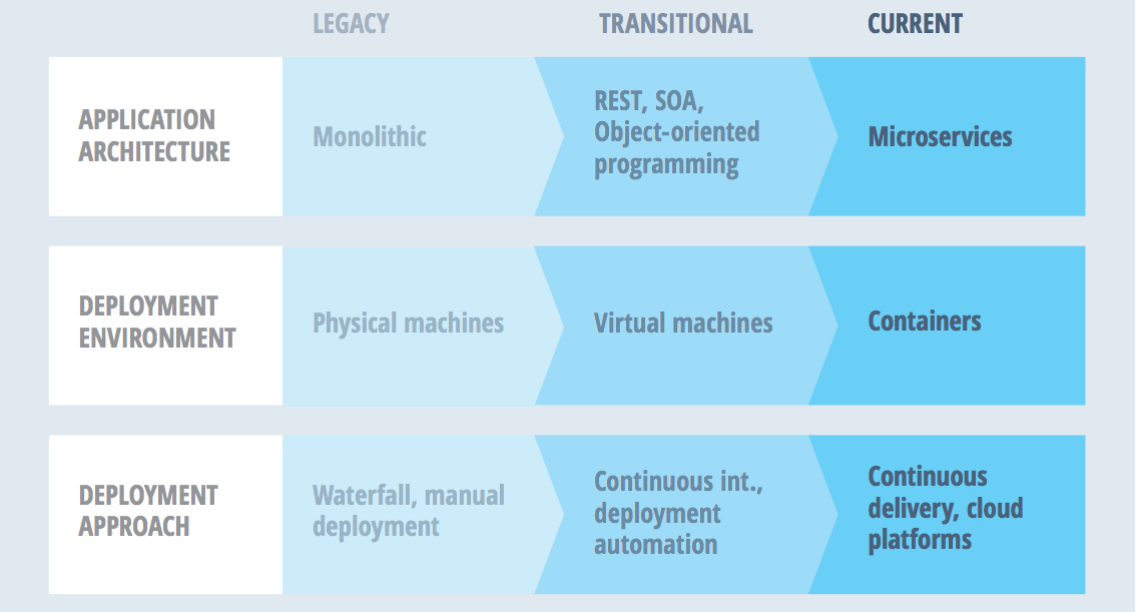
Obrázek 2.2: *Bariéra mezi vývojovým a provozním týmem (Zdroj: [6])*

Dále zde vzniká problém se zodpovědností za odvedenou práci, kdy programátoři chtějí co nejčastěji nasazovat své změny, naproti tomu provozní tým se snaží zachovat stabilitu a co nejvyšší dostupnost služby, což jsou protichůdné požadavky. V nejhorším případě si lze takovou situaci představit jako zeď, přes kterou je kód přehazován, což je znázorněno na obrázku 2.2. Pokud nastane v aplikaci chyba, může být obtížné identifikovat jejího autora, což vede k následnému upřesňování předávky kódu aplikace mezi týmy a k dalšímu zesložiťování celého procesu nasazování.

Pokud nejsou jednotlivé kroky při nasazování aplikace automatizovány, je problém rychlost jejich provádění a samozřejmě je také třeba počítat se zvýšeným rizikem vzniku chyb.

2.3 Propojení vývoje a provozu – DevOps

Pro čelení těmto problémům vznikla filozofie DevOps, jež si dává za cíl oba světy (vývoj i provoz) propojit. DevOps je primárně o kultuře – pokud není ve firmě zájem jej zavádět, je jeho použití prakticky nemožné. Týmy by měly být otevřené a chtít sdílet znalosti, spolupracovat na řešení problémů, být připraveni nést odpovědnost. K tomu je nutné počítat se zavedením automatizace a samozřejmě také měřit vhodné metriky pro ověření dosaženého zlepšení. DevOps je na rozdíl od Tradičního přístupu kontinuální proces, který se neustále opakuje.



Obrázek 2.3: *Vývoj architektury a nasazování software (Zdroj: [5, strana 109])*

Vzhledem k rozsahu této práce se z výše uvedeného dále zabývám jen zaváděním automatizace – nelze pokrýt veškerá témata dostatečně podrobně. Filozofie DevOps postupně promlouvá do přístupu k návrhu architektury software a způsobu provozu a nasazování software. Namísto monolitické architektury nabývá na popularitě architektura mikroslužeb, software je dodáván častěji po menších změnách a plně automatizovaně, což je znázorněno na obrázku 2.3.

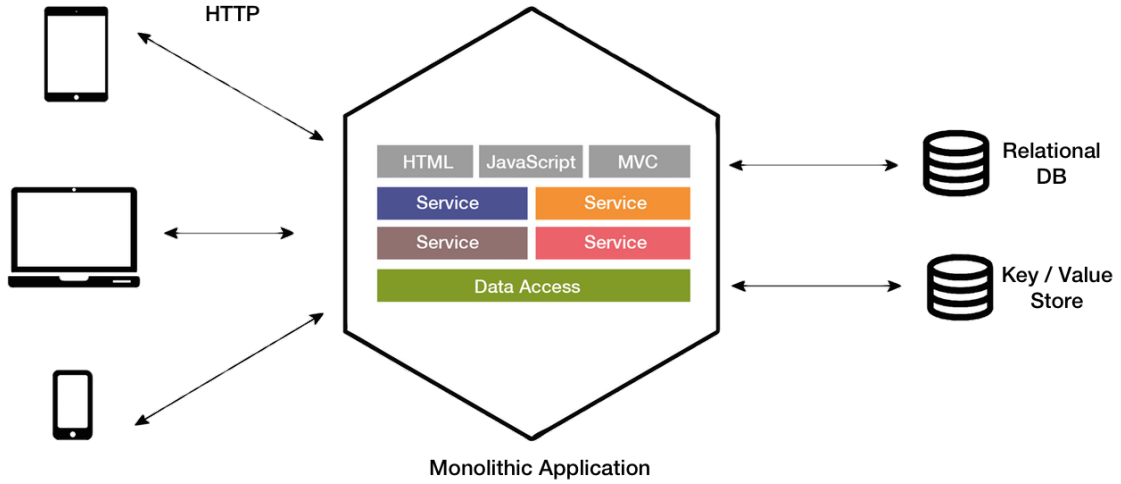
V této práci uvedené principy a techniky naleznou nejlepší uplatnění při vývoji a dodávce software poskytovaného jako služba. Je zřejmé, že některé druhy software buď není možné nebo naopak není žádoucí dodávat kontinuálně, agilním způsobem. Prvním příkladem může být software v kávovaru, ke kterému nemá vývojář po jeho prodeji přístup, druhým příkladem budiž software autonomního řízení vozidla, které musí procházet zcela odlišnými schvalovacími procesy.

3 Prostředky DevOps

Pro dosažení ideálu DevOps je kromě samotné změny přemýšlení nad dodávkou software také nutné se zamyslet nad technickými prostředky. Pokud je vyvíjená aplikace velká a její nasazení trvá příliš dlouhou dobu, je adaptace agilního přístupu téměř nemožná, protože samotné nasazení aplikace veškerou agilitu pohřbí. Je na místě přemýšlet o tom, jak změny doručovat častěji, po menších částech, až kontinuálně. Jako vhodné se jeví změna architektury tak, aby bylo možné nasazovat častěji a automatizace běžných a opakujících se činností v průběhu integrace změn a jejich doručování.

3.1 Architektura mikroslužeb

Jako první je při adaptaci konceptu DevOps nutné přizpůsobit architekturu samotné aplikace. Cílem je umožnění častého nasazování, což je problém v situaci, kdy je třeba sestavit a otestovat komplexní aplikaci.

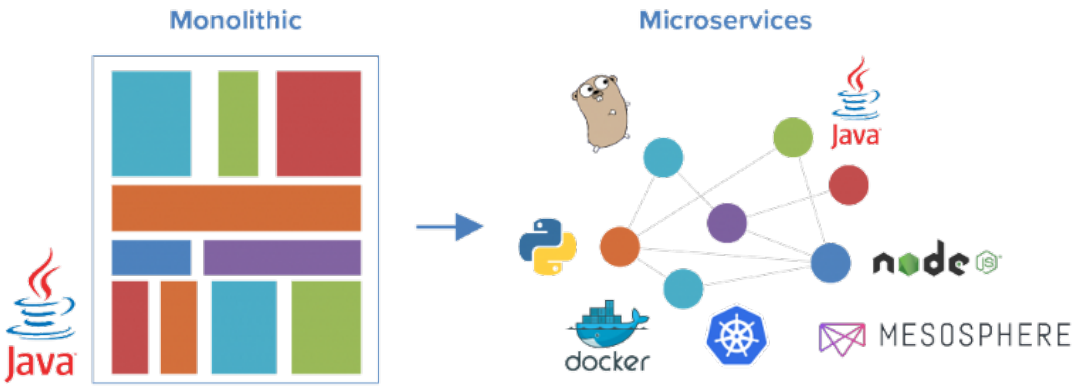


Obrázek 3.1: *Monolit (Zdroj: [7])*

Uvažujme elektronický obchod, který je implementován jako jedna celistvá aplikace, která umožňuje zákazníkům nakupovat, zákaznické podpoře pracovat s objednávkami, účetním pracovat s fakturami a zároveň také vedení firmy generovat reporty. Pokud by v tomto případě bylo třeba provést triviální opravu v jedné části aplikace, je nutné ji vždy otestovat a nasadit celou. Samozřejmě by bylo možné

testovat pouze tu část aplikace, které se změna týká, není pak ale možné odhalit, zda případná chyba neovlivní úplně jinou část aplikace [1, strana 77]. V takovou chvíli je třeba se rozhodnout, do jaké míry je třeba aplikaci testovat, zda případně není možné počkat, až bude nasaditelných změn více. V tento okamžik ale může být možnost pružně reagovat na požadavky obchodu dokonce i konkurenční výhodou. Problémem takové aplikace je také možnost jejího škálování. V okamžiku, kdy není možné navyšovat výkon stroje, na kterém je provozována (vertikální škálování), je možné škálovat horizontálně pouze přidáváním dalších instancí celé aplikace, přestože je fakticky třeba navýšit výpočetní kapacitu pouze pro jednu konkrétní část aplikace [1, strana 78].

Způsob návrhu, kdy jsou spolu všechny části aplikace provázány a musí být nasazovány společně se nazývá **monolitická architektura** [5, strana 7]. V praxi bych tento návrh architektury použil pouze u projektu, který je malý (co do množství zdrojového kódu) a nepředpokládá se u něj v budoucnu další výrazné rozšiřování. Další podmínkou pro jeho úspěšné použití by byla práce jediného programátora na celém projektu. Jinak docházelo spíše k odkládání nasazení a následným obavám, zda se něco nerozbije, protože případná oprava by trvala příliš dlouho. Dalším důsledkem byl problém aktualizace vývojových nástrojů a prostředí, v nichž aplikace běží, což by znamenalo výpadek celé aplikace. V případě, že se neaktualizovalo zase vzrůstalo riziko odhalení bezpečnostních chyb a v konečném důsledku byl také problém nalézt vývojáře, kteří jsou ochotni pracovat s neaktuálními technologiemi.



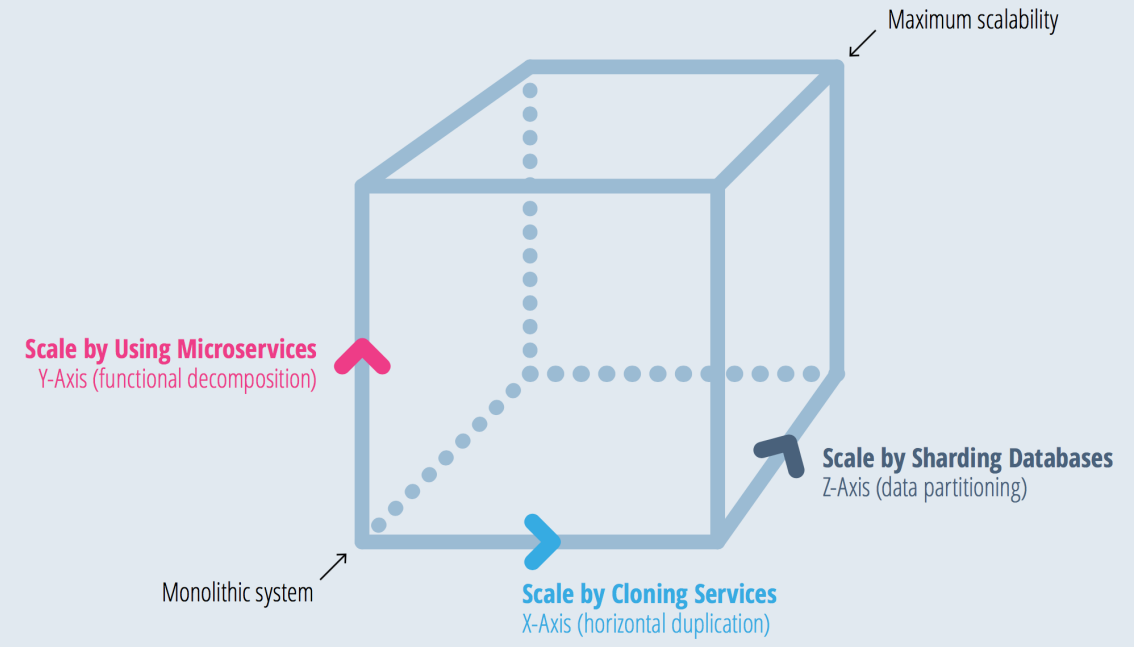
Obrázek 3.2: *Rozdíl mezi monolitickou architekturou a mikroslužbami (Zdroj: [8])*

Naproti tomu **architektura mikroslužeb** umožňuje rozdělení aplikace na několik samostatných celků. Každý díl (mikroslužba) je samostatná aplikace, která je samostatně nasaditelná a spustitelná a disponuje vlastním rozhraním, kterým s ní lze komunikovat [5, strana 8]. Konkrétním příkladem na případu elektronického obchodu je rozdělení na službu obsluhující objednávky, dále službu obsluhující produkty a tak dále. Primární výhodou tohoto přístupu je jednoduchost a samostatnost každé mikroslužby. Díky tomu, že řeší jeden izolovaný problém, je pro programátora snadnější ji pochopit a udržovat, je možné ji daleko rychleji otestovat a nasadit. Dále je také možné pro její vývoj zvolit optimální technologie – pro ukládání objednávek

bude vhodné použít databázi, která zajistí především bezpečné uložení dat, naproti tomu pro službu zobrazující produkty bude vhodnější použít databázi, která nabízí rychlé čtení a dostatek funkcí pro textové vyhledávání [1, strana 83].

3.1.1 Velikost mikroslužeb

Zde je třeba se pozastavit a rozhodnout, jak velké mají mikroslužby být, což je hojně diskutovaný problém [5, strana 13]. Pokud jsou příliš malé a je jich příliš mnoho, může to v konečném důsledku přinést spíše problémy. Provoz každé služby totiž obnáší určitou režii, kterou je třeba brát v potaz. Typicky má každá služba vlastní repozitář, vlastní konfigurace a další závislosti. Na druhou stranu však příliš malé služby nemusí dosáhnout kýženého přínosu. Je také třeba přemýšlet o možném budoucím škálování služeb. Obecně lze aplikace škálovat třemi způsoby: použitím mikroslužeb, jejich duplikací a shardováním databáze, což je znázorněno na obrázku 3.3. V praxi se mi osvědčila dvě pravidla, která mohou poukázat na optimální velikost služby, případně na způsob jejich návrhu.



Obrázek 3.3: *Dimenze škálování a mikroslužby (Zdroj: [5, strana 10])*

Prvně lze začít vyčleňováním samostatně funkčních celků. Je třeba identifikovat vhodné kandidáty, v případě e-shopu to může být například systém pro rozesílání novinek elektronickou poštou. Následně lze postupně oddělovat další části, které mohou být samostatně funkční. Ve výsledku jsou tak mikroslužby děleny dle logických částí aplikace, tak jak mají na starosti jinou část obchodu [5, strana 11]. Tento postup je spíše dlouhodobý, osvědčil se mi ale v případě, kdy byla aplikace technologicky zastaralá a její další rozšiřování (nebo jen aktualizace) je problematické.

Rozdělení aplikace na služby lze vztáhnout i k rozdělení týmů nebo lidí ve firmě. Pokud se jeden vývojář stará v rámci e-shopu o vyhledávání produktů, nabízí se,

aby právě vyhledávání bylo implementováno jako jedna služba s jasně definovaným rozhraním. Pokud pak tento vývojář potřebuje načíst informace o počtu prodaných kusů daného produktu, provede to voláním služby zastřešující objednávky – obdobně, jako by se ptal kolegy, který se o tuto oblast stará. Struktura služeb tak kopíruje strukturu firmy. K tomuto poznání často dojdou firmy samy, aniž by si to uvědomovaly, myšlenka je to však stará padesát let a je známá jako Conway's Law [7].

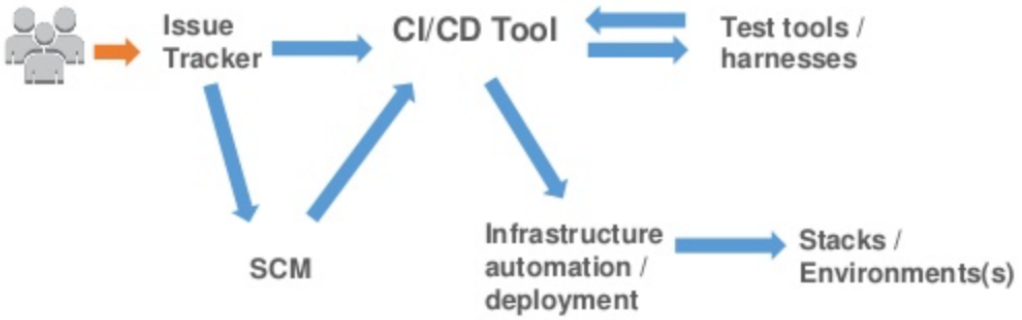
3.1.2 Problémy při implementaci mikroslužeb

V okamžiku, kdy přibývají další mikroslužby lze pozorovat, že se v určitých rysech shodují. Často využívají stejné vývojové nebo běhové prostředí, pracují s obdobnými zdroji. To může vést ke snaze opakující se činnosti nějakým způsobem standardizovat nebo vyčleňovat [5, strana 24]. Ať už se jedná o vytváření sdílených knihoven nebo standardizaci rozhraní a chování služeb, je to krok zpět, který degraduje využití potenciálu mikroslužeb a zavádění inovací do firmy [1, strana 27].

Nelze také opomenout to, že problémy, které existují při vývoji monolitických aplikací se objevují také u mikroslužeb, avšak na vyšší úrovni [8]. Pokud je špatně navržena funkce a ta je volána mnohokrát, je to jisté zdržení v provádění kódu. V případě mikroslužeb je však takové zdržení mnohonásobně delší, protože spolu služby komunikují po síti prostřednictvím definovaného protokolu, což je řádově náročnější, než prosté volání funkce.

3.2 Kontinuální integrace a nasazování

V této kapitole jsou popsány kroky, vedoucí k automatizaci testování a nasazování software. To je důležité pro to, aby byly vytvořené změny co nejdříve dostupné ostatním vývojářům a uživatelům. Nelze však pouze povolit všem veškeré pravomoci k nasazování software, je třeba dodržet několik pravidel, aby byl tento proces spolehlivý a přínosný.



Obrázek 3.4: *Průběh kontinuálního nasazování (Zdroj: [4])*

Nejprve je třeba si ujasnit základní pojmy související s kontinuální integrací [9]. V praxi se používají následující tři pojmy (bývají však často zaměňovány, nebo

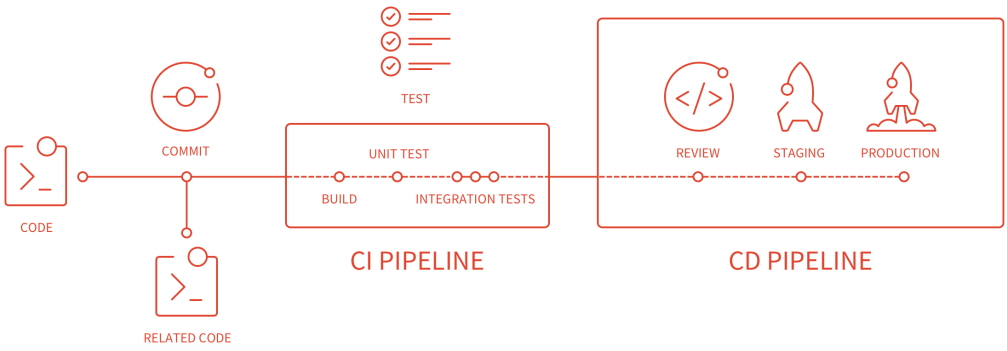
souhrnně označovány jako CI/CD, podrobnější popis rozdílů je například v [10]):

- **Kontinuální integrace** (continuous integration): Kód je několikrát denně ukládán do centrálního repozitáře a je prováděna kontrola tohoto kódu
- **Kontinuální doručování** (continuous delivery): Navíc k integraci je sestaven artefakt, který je nasaditelný
- **Kontinuální nasazování** (continuous deployment): navíc k doručování je tento artefakt nasazen do provozu

Postup integrace je zpravidla následující: Uživatel nahraje změny v zdrojových kódech do sdíleného repozitáře, ten oznámí integračnímu nástroji detail této změny. Integrační nástroj zkontroluje zdrojový kód a pokud je v pořádku, vytvoří artefakt, který je možné nasadit. Posledním krokem je samotné nasazení vytvořeného artefaktu do provozu, přičemž o celém průběhu je vývojář průběžně informován.

3.2.1 Nástroje pro kontinuální integraci

Nástrojů, které umožňují kontinuální integraci je celá řada, jejich komplexní výčet je k dispozici v [5, strana 139]. Výběr vhodného nástroje je závislý na požadavcích a možnostech dané společnosti. Základní členění je podle toho, zda je daný nástroj poskytován jako služba, nebo si jej musí uživatel spravovat sám. Každé řešení má své výhody a nevýhody, každý podnik je schopný akceptovat něco jiného, obecně však je viditelná tendence firem spíše využívat služeb. Dalším kritériem výběru je otevřenost nástroje a jeho cena. Konečně je také kritériem pro výběr nástroje jeho funkčnost, rychlost implementace a podpora napojení na další systémy.



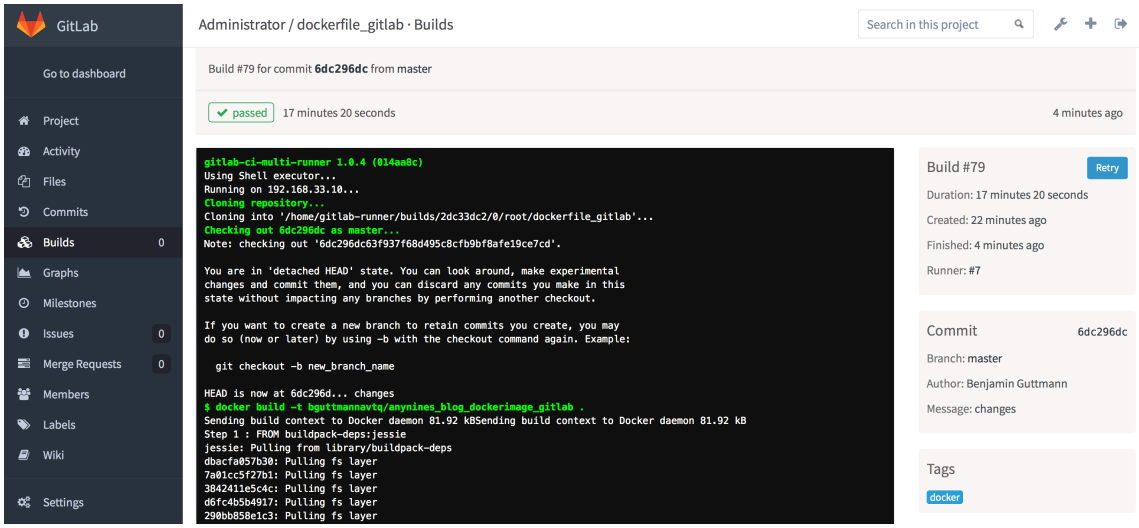
Obrázek 3.5: *Dílčí kroky integračního serveru (Zdroj: [11])*

V praxi jsem používal více nástrojů - GitLab CI, CircleCI, Drone, Jenkins a Bamboo. Z těchto nástrojů je jediný kompletně placený nástroj Bamboo [13], který byl použit z důvodu dostupné integrace s dalšími nástroji firmy Atlassian. Bohužel tento nástroj optimálně nepodporuje agilní přístup a tak celý proces zpomaluje.

Zajímavý je nástroj Drone [12], který je extrémně štíhlý a rychlý, dostupný včetně zdrojových kódů. Jeho nevýhodou může být nutnost jej zprovoznit a spravovat na své vlastní infrastruktuře, na druhou stranu ale nedisponuje žádnými omezeními, může tak jít při častějším používání o cenově výhodnější řešení. Na stejném principu je k dispozici Jenkins [15], který je jedním z nejrozšířenějších nástrojů v oboru. Populární je zejména díky dostupné řadě pluginů, které umožňují nástroj přizpůsobit téměř jakýmkoli potřebám. Je však třeba počítat s vyšší složitostí, zejména ve srovnání s Drone. Naproti tomu CircleCI [14] je poskytován jako služba, přičemž je možné provádět souběžně právě jednu integraci, další navyšování výpočetní kapacity je zpoplatněno. Výhodou této služby je řada předinstalovaných a předkonfigurovaných nástrojů, které lze použít. Automaticky je tak možné detekovat například použitý framework pro spuštění testů. Posledním nástrojem je GitLab CI [11], který je dostupný všemi možnými způsoby - jako Open Source, jako placený, hostovaný i jako služba. Vzhledem k jeho dostupnosti a snadnosti použití budu dále uvádět ukázky právě pro GitLab CI.

3.2.2 Implementace kontinuální integrace pomocí GitLab CI

Základem je definovat jednotlivé kroky, které má integrační server vykonat. Ty jsou zpravidla rozděleny do fází (stages), které seskupují příkazy zajišťující jeden druh činnosti. Může jít o instalaci závislostí, provedení statické analýzy, spuštění jednotlivých testů (jednotkových, integračních, zátěžových...), vytvoření artefaktu, jeho zveřejnění a následné nasazení do odpovídajícího prostředí, případně otestování po tomto nasazení – tyto fáze jsou znázorněny na obrázku 3.5. Následně jsou v rámci těchto sekcí prováděny jednotlivé příkazy, kterými může být konkrétní spuštění skriptu, programu nebo utility.



Obrázek 3.6: *Sledování průběhu integrace v GitLab CI (Zdroj: [11])*

Například konfigurace pro projekt psaný v jazyce PHP, který je třeba sestavit, otestovat, zveřejnit a nasadit by vypadala následovně: nejprve by bylo v souboru

`.gitlab-ci.yml` třeba definovat sekce, kterými je třeba projít v sekci `stages`. Následně jsou pro každou sekci uvedeny příkazy, které se mají vykonat – pokud je daná sekce vykonána úspěšně (příkazy končí návratovým kódem `0`), postupuje se ve vykonávání k další sekci. Kupříkladu druhá sekce má na starosti spuštění testů, přičemž nejprve je provedena statická analýza kódu pomocí nástroje `phpcs`, následně jsou spuštěny jednotkové testy nástrojem `phpunit` a nakonec také akceptační nebo integrační pomocí nástroje `codeception`, který na pozadí spouští webový prohlížeč. Celá konfigurace je zobrazena ve výpisu 3.1, přičemž dále popisuje stažení závislostí a sestavení balíčku v sekci `build`, jeho zveřejnění v sekci `release` a nakonec také nasazení v sekci `deploy`.

Jediným limitem je zde nutnost použití integrace současně s repozitářem GitLab. Ten je nabízen zdarma jako služba a s integrační částí sdílí společné grafické rozhraní. Přímo v kódu je tak vidět, co již bylo integrováno a s jakým výsledkem. Průběh integrace může sledovat každý, kdo má k repozitáři přístup, na obrázku 3.6 lze vidět průběh právě probíhající integrace. Vhodné je také provést propojení s chatem (například Slack nebo HipChat) – poté je v případě neúspěchu ihned jasné, kdo celou pipeline rozbil a může se tak ihned začít pracovat na nápravě.

Výpis 3.1: *Ukázková konfigurace pro GitLab CI*

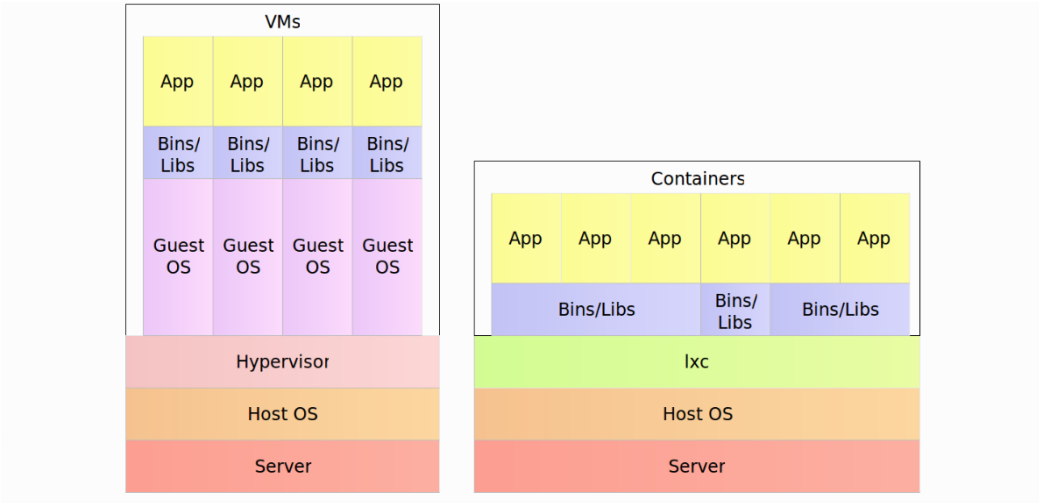
```
stages:
    - build
    - test
    - release
    - deploy
build:
    stage: build
    script:
        - composer install
        - docker build -t my-app .
test:
    stage: test
    script:
        - vendor/bin/phpcs
        - vendor/bin/phpunit
        - vendor/bin/codeception run
release:
    stage: release
    script:
        - docker push my-app
deploy:
    stage: deploy
    script:
        - kubectl apply -f kube-config.yaml
```

Adaptace kontinuální integrace je pro mě v dnešní době automatická a je to většinou první krok po vytvoření repozitáře pro sdílení zdrojových kódů. Sice jde o práci navíc, zrychlení dalšího vývoje ale investovaný čas navrátí. Navíc je mnohem snazší nastavit přísné kontroly kódu u nového projektu, než u projektů existujících.

U již rozběhlých projektů je implementace složitější, zejména pokud je cílem zavést kontrolu kódu tam, kde v minulosti nebyla prováděna. V tomto případě se mi osvědčilo zavést určitou hranici (například počet tolerovaných chyb mess detektoru) s tím, že se nesmí zvyšovat a o volných chvílích se nabízí práce na snížení této hranice. Jako důležité také shledávám možnost konfigurace integračního nástroje souborem, který je součástí zdrojových kódů – díky tomu jsou mohou být veškeré změny v konfiguraci schvalovány a verzovány.

3.3 Kontejnerizace

S mikroslužbami a kontinuální integrací souvisí také to, že je třeba nějakým způsobem vytvořit takový artefakt, který bude možné snadno distribuovat a spustit. K tomu by bylo možné vytvářet binární soubory, JAR nebo DLL balíčky, nicméně ne každý programovací jazyk toto umožňuje – například skriptovací jazyky zpravidla vyžadují nějaké běhové prostředí. To není úplně ideální situace ve chvíli, kdy pracujeme s mikroslužbami, a s každou z nich se pracuje jinak, což komplikuje jejich provoz.



Obrázek 3.7: *Porovnání virtálního stroje a kontejneru (Zdroj: [1, strana 68])*

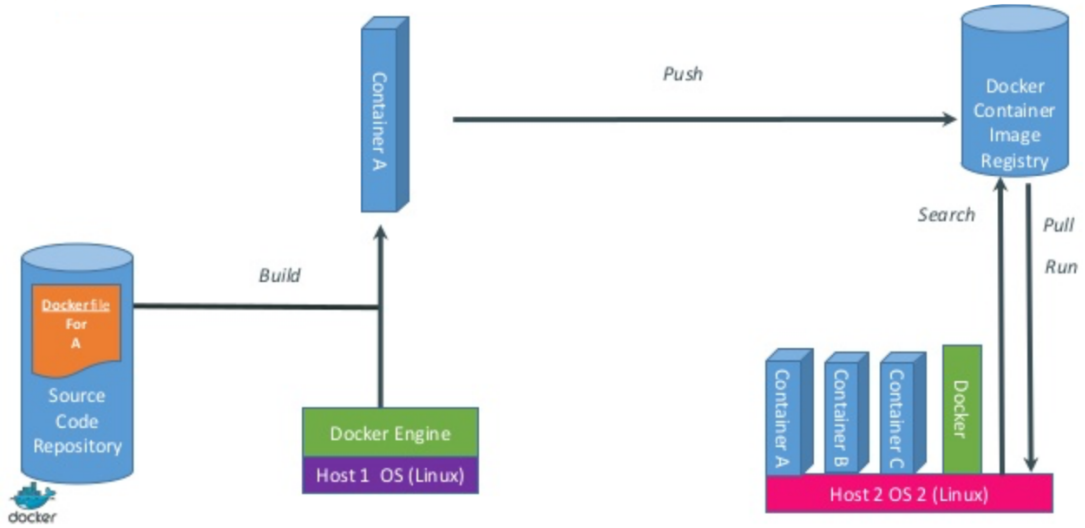
Pro zefektivnění provozu je přínosné vycházet z metodologie The Twelve-Factor App [16], která přináší sadu doporučení, jak těmto problémům čelit. Tato doporučení lze shrnout do několika bodů:

- Používání deklarativního formátu pro veškeré nastavení – nemělo by být nutné cokoliv ručně konfigurovat veškeré konfigurační soubory by měly být součástí kódu (stejně jako příklad konfigurace GitLab CI v kapitole 3.2).

- Jasně definovat rozhraní, kterým je možné se spustitelnými balíčky pracovat – ať už jde o jejich spouštění a ovládání, monitoring nebo nastavení.
- Minimalizaci rozdílů pro spouštění v různých prostředích včetně cloudu, s čímž souvisí připravenost pro bezproblémové škálování.

3.3.1 Docker

Tomuto přístupu jde naproti použití kontejnerů. Ty jsou k dispozici již řadu let, jejich popularizaci však způsobil až Docker [17], s jehož pomocí je práce s kontejnery snadná. Kontejner si lze představit jako virtuální stroj, který obsahuje spouštěnou aplikaci se všemi jejími závislostmi, který je však snadné distribuovat a spouštět. Přesnější je však o kontejneru mluvit jako o izolovaném procesu, což lépe vystihuje jeho rychlost spouštění a výkonnost, protože kontejner sdílí řadu prostředků s operačním systémem, v němž je spouštěn.



Obrázek 3.8: *Kroky při práci s nástrojem Docker (Zdroj: [18])*

Práce s nástrojem Docker probíhá následovně: Nejprve je napsán `Dockerfile` (příklad je ve výpisu 3.2), což je soubor, který definuje kroky, kterými vzniká tzv. obraz (Docker image). Prvním krokem je zpravidla zvolení základního obrazu, z kterého se vychází. To může být jak štíhlá linuxová distribuce typu Busybox, tak kompletní distribuce jako Debian nebo CentOS. Přestože se může zdát vhodné použít známou plnohodnotnou distribuci obsahující všechny myslitelné nástroje, je daleko vhodnější použít co nejjednodušší distribuci. Jednak se tím sníží požadavky na propustnost sítě a velikost diskového prostoru a s obrazy je tak možné lépe manipulovat (rychleji je distribuovat a spouštět), především je však zvýšena bezpečnost díky použití pouze nutných závislostí. Sestavit obraz je možné pomocí příkazu `docker build -t my-app` spouštěném ve složce obsahující `Dockerfile`. Po provedení tohoto příkazu je vytvořen obraz pojmenovaný jako `my-app`. Ten je možné příkazem `docker push my-app` nahrát do veřejného úložiště, kterému se říká

Docker Registry, přičemž pokud není řečeno jinak, tak je použito výchozí dostupné na `https://hub.docker.com`. Na serveru je možné tento obraz spustit (a vytvořit tak kontejner) příkazem `docker run my-app`. Ten, pokud obraz nenalezne lokálně, stáhne obraz z registru.

Výpis 3.2: *Příklad souboru Dockerfile*

```
# Volba základního obrazu - linuxové distribuce Alpine Linux
FROM alpine:3.1.0

# Instalace NGINX (webserver a proxy)
RUN apk --update add nginx

# Kontejner zveřejní port 80
EXPOSE 80

# Při spuštění kontejneru je spuštěn NGINX na popředí
ENTRYPOINT nginx -t
```

Bylo by samozřejmě možné vzniklý obraz nahrát rovnou na server a tam ho spustit – výhodou použití registru je však to, že obraz může být stažen kdykoli i na jiný server, striktně tento přístup odděluje krok zveřejnění artefaktu od kroku jeho nasazení. Dále jsou aplikace v kontejnerech nastavovány pomocí proměnných prostředí, mají přesně definované zveřejněné porty, produkují log jako proud událostí, což ctí principy *The Twelve Factor Apps*. Největší výhodou kontejnerů je však to, že umožňují dosáhnout totožného prostředí pro běh aplikace bez závislosti na použitém stroji. Díky tomu lze předejít chybám, kdy se aplikace chovala korektně v testovacím prostředí ale ne v produkčním, protože se zapomnělo na aktualizaci jedné knihovny. Dále značně zjednodušují lokální vývoj – vzhledem k nezávislosti na operačním systému, v němž jsou provozovány je schopen vývojář zprovozni prostředí pro vývoj během okamžiku. Po dokončení vývoje pouhým odstraněním stažených obrazů může uvést svůj stroj do stavu před stažením veškerých závislostí.

Docker sám o sobě však neřeší veškeré problémy, které se vyskytují při použití mikroslužeb a kontejnerů v produkčním prostředí. V praxi je třeba řešit také dostupnost aplikace, monitorovat a vyhodnocovat její chování, což je součástí cyklu DevOps. S rostoucím množstvím provozovaných služeb a serverů je třeba řídit jejich provoz. Ve světě kontejnerů se nástroje umožňující správu kontejnerů označují jako orchestrátory, přičemž jich existuje celá řada [19, strana 123]. Ty umožňují nasazení kontejnerů napříč celými clustery, kontrolu stavu jednotlivých kontejnerů, jejich dostupnost. Dokáží se tak vyrovnat se selháním některého z serverů nebo škálovat konkrétní mikroslužbu při jejím přetížení. Zároveň také umožňují agregaci logů, které kontejnery produkují a jejich zpřístupnění správci.

4 Závěr

V této práci byly popsány problémy vznikající při vývoji a provozu software tradičním způsobem vycházejícím z vodopádového modelu. Byl zaveden pojem DevOps, který přináší nový přístup k vývoji a provozu software, který přináší otevřenost a zodpovědnost do těchto procesů, přičemž cílem je dodávání software tak, aby byl zákazník spokojenější. Toho je dosahováno jak změnou kultury a uvažováním vývojářů a správců, tak zavedením řady technických opatření nebo nástrojů.

Práce popisuje především technickou stránku zavádění DevOps, zaměřuje se na automatizaci procesů sovisejících s nasazováním software. Podrobně je popsána architektura mikroslužeb, která umožňuje s aplikací pracovat prostřednictvím menších částí. Dále jsou popsány kontinuální integrace a nasazování jako způsob neustálého automatického doručování software do produkčního prostředí. Nakonec jsou také popsány kontejnery jako způsob efektivní distribuce jednotlivých služeb.

Vzhledem k rozsahu práce nebyla vyčerpána veškerá témata související s automatizací, ani nebyla všechna témata probrána do dostatečné hloubky – pro další prozkoumání problematiky lze doporučit literaturu uvedenou v závěru práce, zejména knihu The DevOps 2.0 Toolkit [1] a knihy z série The New Stack ([5] a [19]).

Z témat, kterým by bylo vhodné se dále podrobněji věnovat je jistě orchestrace (práce s vysokou dostupností aplikace) nebo monitoring (agregace logů, jejich vyhodnocování a alerting). Jako teoretického nástupce kontejnerů pak lze v určitých případech uvést architekturu Serverless [20], která vývojáře plně odstiňuje od procesu sestavování balíčků nebo kontejnerů, ale je distribuován pouze kód, který je následně spouštěn pouze na infrastruktuře poskytovatele. Díky tomu není třeba při návrhu infrastruktury uvažovat nad výkonností jednotlivých serverů a době jejich použití - v případě Serverless je infrastruktura poskytovatele využita maximálně efektivně a zpoplatněna pouze za její faktické využití.

Literatura

[1] FARCIC Viktor. *The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices*. Packt Publishing, 2016 462 s. ISBN 978-1-78528-031-3.

[2] SCHAEFFER Chuck. *DevOps & CRM Software Come Together* [online]. 2017 [vid. 2017-05-01]. Dostupné z: http://www.crmsearch.com/devops.php.

[3] VERMA Ravi. *Target Deployment Servers/Environments* [online]. 2017 [vid. 2017-05-01]. Dostupné z: http://scmquest.com/target-deployment-servers-environments.

[4] WHITE Adrian. *DevOps Culture – Continuous Integration & Continuous Deployment on the AWS Cloud* [online]. 2014 [vid. 2017-05-03]. Dostupné z: https://www.slideshare.net/AmazonWebServices/day-3-devops-culture-continuous-integration-continuous-deployment-on-the-aws-cloud.

[5] THE NEW STACK. *Applications & Microservices with Docker & Containers* [e-kniha]. 2017 [vid. 2017-05-01]. Dostupné z: https://thenewstack.io/ebookseries/.

[6] SHARMA Akshat. *Enabing DevOps in an SDN World* [online]. 2014 [vid. 2017-05-02]. Dostupné z: https://www.slideshare.net/CiscoDevNet/enabing-devops-in-an-sdn-world.

[7] MANCUSO Emiliano. *Microservices 101* [online]. 2015 [vid. 2017-05-03]. Dostupné z: http://bits.citrusbyte.com/microservices/.

[8] NETSIL. *Microservices, DevOps and Production Complexity* [online]. 2016 [vid. 2017-05-02]. Dostupné z: https://blog.netsil.com/microservices-devops-and-operational-complexity-be98cb01b660.

[9] FOWLER Martin. *Continuous Integration* [online]. 2006 [vid. 2017-05-03]. Dostupné z: https://www.martinfowler.com/articles/continuousIntegration.html.

[10] GOLUB Sergiy. *Continuous Delivery vs Continuous Deployment vs Continuous Integration: Key Definitions* [online]. 2012 [vid. 2017-05-03].

Dostupné z: https://blog.assembla.com/AssemblaBlog/tabid/12618/bid/92411/Continuous-Delivery-vs-Continuous-Deployment-vs-Continuous-Integration-Wait-huh.aspx.

[11] GITLAB CE. *GitLab Continuous Integration (GitLab CI)* [online]. 2017 [vid. 2017-05-03]. Dostupné z: https://docs.gitlab.com/ce/ci.

[12] DRONE.IO. *Drone Continuous Delivery* [software]. [přístup 2017-05-03]. Dostupné z: https://github.com/drone/drone.

[13] ATLASSIAN. *Bamboo* [software]. [přístup 2017-05-03]. Dostupné z: https://www.atlassian.com/software/bamboo.

[14] CIRCLE CI. *CircleCI* [software]. [přístup 2017-05-04]. Dostupné z: https://circleci.com.

[15] JENKINS. *Jenkins* [software]. [přístup 2017-05-04]. Dostupné z: https://jenkins.io.

[16] WIGGINS Adam. *The Twelve-Factor App* [e-kniha]. 2017 [vid. 2017-05-02]. Dostupné z: https://12factor.net/12factor.epub.

[17] DOCKER INC. *Docker* [software]. [přístup 2017-05-05]. Dostupné z: https://www.docker.com.

[18] DOT CLOUD. *Why Docker* [online]. 2013 [vid. 2017-05-05]. Dostupné z: https://www.slideshare.net/dotCloud/why-docker.

[19] THE NEW STACK. *Automation & Orchestration with Docker & Containers* [e-kniha]. 2017 [vid. 2017-05-02]. Dostupné z: https://thenewstack.io/ebookseries/.

[20] ROBERTS Mike. *Serverless Architectures* [online]. 2016 [vid. 2017-05-06]. Dostupné z: https://martinfowler.com/articles/serverless.html.