Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Automatické testování cloud native aplikací

Bc. Martin Sládek

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Martin Sládek

Systémové inženýrství a informatika
Informatika

Název práce

Automatické testování cloud native aplikací

Název anglicky

Automated testing of cloud native applications

Cíle práce

Diplomová práce je tematicky zaměřena na automatizované testování softwaru tzv. cloud native aplikací. Hlavním cílem je analyzovat procesy automatizovaného testování softwaru z pohledu životního cyklu vývoje softwaru přístupem CI/CD v prostředí cloud native aplikací.

Dílčí cíle diplomové práce jsou:
- vytvořit přehled řešené problematiky v oblasti konceptů testování software,
- analyzovat současný stav a požadavky pro návrh a optimalizaci testování aplikace vyvíjené na míru v podnikové praxi,
- modelovat podnikový proces automatizovaného testování software v CI/CD prostřednictvím notace BPMN,

- navrhnout zefektivnění procesu testování v životním cyklu vývoje softwaru.

Metodika

Metodika řešené problematiky diplomové práce vychází ze studia a analýzy odborných informačních zdrojů. Praktická část práce je zaměřena na vypracování případové studie analyzující materiály dostupné a získané z vybrané společnosti, ve které probíhá implementace, vývoj a testování frontendového systému aplikací pro call centra, kterého se student účastní. Modelovaný proces bude vytvořen s využitím Business Process Model and Notation (BPMN). Na základě syntézy teoretických poznatků a výsledků praktické části práce budou formulovány závěry diplomové práce a provedena diskuze na téma zefektivnění testování softwaru v životním cyklu jeho vývoje.

Doporučený rozsah práce

60-80

Klíčová slova

Automatizované testování, cloud native, CI/CD, software, BPMN

Doporučené zdroje informací

BALALAIE, Armin; HEYDARNOORI, Abbas; JAMSHIDI, Pooyan. Microservices architecture enables devops: Migration to a cloud-native architecture. Ieee Software, 2016, 33.3: 42-52.

BOROVCOVÁ, Anna. Kvalita a testování v České republice. Systémová integrace. 2011, 18(1), 16. ISSN 1804-2716.

BUCHALCEVOVÁ, Alena a Jan KUČERA. Hodnocení metodik vývoje informačních systémů z pohledu testování. Systémová integrace. 2008, 15(2), 13. ISSN 1210-9479.

BUREŠ, Miroslav, Miroslav RENDA, Michal DOLEŽEL, Peter SVOBODA, Zdeněk GRÖSSL, Martin KOMÁREK, Ondřej MACEK a Radoslav MLYNÁŘ. Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu. Praha: Grada, 2016. Profesionál. ISBN 978-80-247-5594-6.

CANO, Patricia Ortegon, et al. A Taxonomy on Continuous Integration and Deployment Tools and Frameworks. In: International Conference on Software Process Improvement. Springer, Cham, 2020. p. 323-336.

KRATZKE, Nane; QUINT, Peter-Christian. Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. Journal of Systems and Software, 2017, 126: 1-16.

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

Ing. Jan Tyrychtr, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 19. 11. 2020

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 11. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 29. 03. 2021

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Automatické testování cloud native aplikací" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 29.03.2021 _____

Poděkování

Rád(a) bych touto cestou poděkoval(a) svému vedoucímu práce panu doktoru Janu Tyrychtrovi za odborné vedení mé práce a jeho cenné rady. Dále bych rád poděkoval společnosti ZOOM Intenational a.s. za poskytnutí zázemí pro vytvoření praktické části této práce.

Automatické testování cloud native aplikací

Abstrakt

Tato diplomová práce vznikla za účelem analyzování a modelování procesů v rámci vybrané společnosti vyvíjející softwarový produkt pro své zákazníky, kde dochází k automatizovanému testování tohoto produktu. Během analýzy v praktické části této práce bylo zjištěno, že ačkoliv vybraná společnost využívá a aplikuje jak návrh architektury, tak i nástroje typické pro cloud native architekturu, nemá zatím plně dokončený cloud native produkt. Nicméně je aktuálně ve fázi těsně před jejím spuštěním a proces vývoje a testování zde již nějakou dobu probíhá přístupem jako vývoj cloud native aplikací.

Procesy jsou modelovány za využití metodiky BPMN a v této notaci pak zaneseny do BPD diagramů. Vybrány byly procesy, které se přímo týkají automatizovaného testování z vnějšího pohledu, ale je zde modelováno i několik pohledů zevnitř zejména v metodice CI/CD. Všechna teoretická východiska byla analyzována studiem odborných informačních zdrojů v teoretické části této práce. Na základě syntézy těchto teoretických poznatků, výsledků praktické části, ale i praxe ve vybrané společnosti byly následně modelovány procesy obsahující návrhy na zefektivnění dvou takových procesů.

Klíčová slova: Automatizované testování, cloud native, CI/CD, software, BPMN

Automated testing of cloud native applications

Abstract

This diploma thesis has been made for purposes of analyzing and modeling processes inside the chosen company which has been developing software product to deliver it to their clients, where the automated testing is implanted. During the analysis within practical part of this thesis, it has been discovered, that even if the company does not have stable released cloud native product, it uses and implementing tools and approaches tied up with cloud native architecture. However, the company is in a stage just before releasing and processes of development and testing with cloud native architecture has been implemented yet.

These processes have been modeled with BPMN and BPD diagrams has been made afterwards. Processes that directly relate to the automated testing either from the outside or the inside, especially in CI/CD methodology have been chosen. Theoretical part of this thesis has been analyzed by studying of technical and academical. By the end, there have been modeled two processes containing proposals of increasing efficiency on the basis of synthesis of theoretical knowledge, results of practical part and practice gained within the chosen company.

Keywords: Automated testing, cloud native, CI/CD, software, BPMN

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

O tom, jak je rychlý a dynamický vývoj informačních a komunikačních technologií, odvětví označované zkratkou ICT (Information and Communication Technologies) již dnes není pochyb. Vlastně se z tohoto trendu stala jedna z charakteristik dnešní doby. Rychlý, bezpečný a kvalitní software pomáhá podnikům a společnostem uspět v současných tržních podmínkách. Pokud se ale v softwarovém produktu vyskytnou vady, může to nevyhnutelně vést k selhání informačního systému a důsledkem mohou být velké až zdrcující škody. Je tedy v zájmu těchto společností software otestovat, tedy ověřit, zda výsledný softwarový produkt odpovídá definovaným požadavkům. Testování softwaru není samotný proces, ale jedná se o součást procesu vývoje softwarového produktu. Metodik vývoje a testování existuje velké množství a rozdíly nalezneme jak v přístupu, tak v samotných procesech vývoje softwaru. [1]

Migrace do prostředí cloudu byla populárním tématem několika posledních let nejen napříč průmyslem zabývajícím se informačními technologiemi, ale i napříč akademickým prostředím. I přes mnoho výhod, které představuje cloudové řešení, jako je například vysoká míra dostupnosti a škálovatelnosti, není většina tzv. on-premise aplikací připravena a ani schopna výhody tohoto prostředí naplno využívat. Jejich přesun a přizpůsobení do tohoto prostředí cloudu je úkol netriviální. V prostředí cloudu jsou deklarovány tzv. mikroslužby (angl.: microservices), které se objevily v posledních letech a jsou považované za nové architektonické styly, které jsou pro cloud prostředí nativní. Mohou tak výrazně usnadnit migraci on-premise architektury aplikací tak, aby mohly plně využívat cloudové prostředí, jako například škálovatelnost. I přes jasné výhody nejsou ale mikroslužby vhodné pro všechny případy. Přináší totiž do systému nová komplexní řešení a jejich implementace by měla být zvážena z hlediska mnoha faktorů, jako například složitost distribuce. [2]

Počátek cloudové přístupu je datován do roku 2006, kdy při první spuštění byla široké veřejnosti představena cloudová služba dnešním největším provozovatelem cloud služeb, který je Amazon Web Services (AWS). V průběhu let až do současnosti se přidali další a další firmy, které pomohly definovat pojmy jako cloud computing a cloud native. [3]

2 Cíl práce a metodika

2.1 Cíl práce

Diplomová práce je tematicky zaměřena na automatizované testování softwaru tzv. cloud native aplikací. Hlavním cílem je analyzovat procesy automatizovaného testování softwaru z pohledu životního cyklu vývoje softwaru přístupem CI/CD v prostředí cloud native aplikací.

Dílčí cíle diplomové práce jsou:

- vytvořit přehled řešené problematiky v oblasti konceptů testování software,
- analyzovat současný stav a požadavky pro návrh a optimalizaci testování aplikace vyvíjené na míru v podnikové praxi,
- modelovat podnikový proces automatizovaného testování software v CI/CD prostřednictvím notace BPMN,
- navrhnout zefektivnění procesu testování v životním cyklu vývoje softwaru.

2.2 Metodika

Metodika řešené problematiky diplomové práce vychází ze studia a analýzy odborných informačních zdrojů. Praktická část práce je zaměřena na vypracování případové studie analyzující materiály dostupné a získané z vybrané společnosti, ve které probíhá implementace, vývoj a testování frontendového systému aplikací pro call centra, kterého se student účastní. Modelovaný proces bude vytvořen s využitím Business Process Model and Notation (BPMN). Na základě syntézy teoretických poznatků a výsledků praktické části práce budou formulovány závěry diplomové práce a provedena diskuze na téma zefektivnění testování softwaru v životním cyklu jeho vývoje.

2.2.1 Procesní modelování

Tvorba procesních modelů reflektuje abstraktní podobu obchodního procesu. Pomáhá tak zachovat jeho přehlednost pro všechny účastníky procesu a usnadňuje další práci s takovými procesy. Anglický výraz „business process“ nemá v češtině jasně definovaný překlad. V praxi je možné setkat se s pojmy obchodní proces, podnikový proces nebo s počeštěnou verzí *„byznys proces“*.

2.2.2 **BPMN**

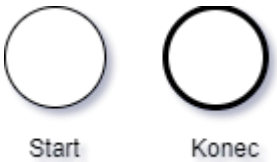
Standard Business Process Modeling Notation (BPMN) byl vyvinutý skupinou Business Process Management Initiative (BPMI) v roce 2004. Primárním cílem BPMN je poskytnout notaci, která je jednoduše pochopitelná všemi firemními uživateli (od firemních analytiků, přes vývojáře až po pracovníky, kteří budou firemní procesy řídit a monitorovat). BPMN vytváří standardizovaný přechod, který vyplňuje mezeru mezi analýzou procesu a její následnou implementací. BPMN dále definuje Business Process Diagram (BPD), který je použitý k modelování procesů vybrané společnosti v praktické části této práce. BPD je tvořen základními komponentami a jejich sub-komponentami:

1. Tokové objekty (Flow Objects)
    a. Událost (Event)
    b. Aktivita (Activity)
    c. Brána (Gateway)
2. Spojovací objekty (Connecting Objects)
    a. Sekvenční tok (Sequence Flow)
    b. Tok zpráv (Message Flow)
    c. Asociace (Association)
3. Plavecké dráhy (Swimlanes)
    a. Bazén (Pool)
    b. Dráha (Lane)
4. Artefakty (Artifacts)
    a. Datový objekt (Data Object)
    b. Seskupení (Group)
    c. Poznámka (Annotation)

Všechny autorské diagramy a komponenty této práce byly vytvořeny v nástroji **draw.io** [4]**.**

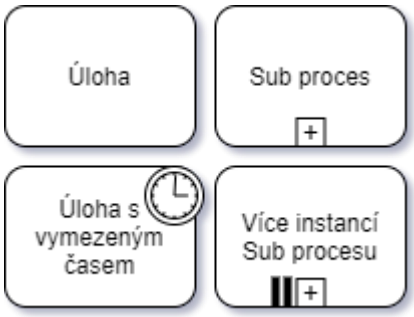
Tokové objekty – Událost

Událost je reprezentovaná pomocí kruhu. Události ovlivňují tok procesu a obvykle mají příčinu, která ji spustí. Podle vztahu k procesu rozlišujeme několik typů událostí například Počáteční (Start) a Koncovou (End).



Tokové objekty – Aktivita

Aktivita je reprezentovaná pomocí obdélníku se zaoblenými rohy. Je to obecný grafický prvek představující nějakou práci. Typy aktivit jsou: Úloha a Sub proces. Proces může mít například více instancí nebo být mít vymezený časový rámec.



Tokové objekty – Brána

Brána je reprezentovaná pomocí čtverce či kosočtverce, stojícím na špici a je používaná pro kontrolu rozdělení a sloučení procesního toku. Interní značky indikují typ chování brány.

Spojovací objekty – Sekvenční tok

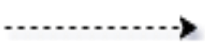
Sekvenční tok je reprezentovaný plnou čárou s vyplněnou šipkou a určuje pořadí v jakém budou jednotlivé aktivity procesu vykonávány.

Spojovací objekty – Tok zprávy

Tok zpráv je reprezentován pomocí přerušované čáry s prázdnou šipkou. Používá se pro zobrazení toku zpráv mezi různými účastníky (procesními rolemi), kteří si mezi sebou posílají a přijímají zprávy.

Spojovací objekty – Asociace

Asociace je reprezentovaná tečkovanou čárou. Používá se k propojení objektu s nějakou dodatečnou informací (data, text nebo jiný artefakt). Jsou také používané pro ukázaní vstupů a výstupů aktivit.



Plavecké dráhy – Bazén

Bazén reprezentuje účastníky v procesu. Taktéž se chová jako grafický kontejner pro oddělení množiny aktivit z jiného bazénu (procesu).

Plavecké dráhy – Dráha

Dráha je podúrovní bazénu. Rozprostírá se v rámci celého bazénu, a to buď vertikálně nebo horizontálně. Dráhy se používají pro organizaci a kategorizaci aktivit.

Artefakty – Datový objekt

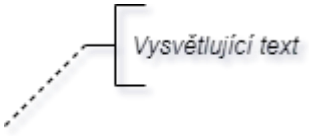
Datové objekty představují mechanizmus jakým způsobem ukázat, že určitá data jsou požadovaná, nebo produkovaná určitou aktivitou. Jsou připojené k aktivitám pomocí asociační vazby.

Artefakty – Seskupení

Seskupení je reprezentováno obdélníkem kresleným přerušovanou čárou. Seskupování může být použito při dokumentaci nebo k účelové analýze, ale nemá žádný vliv na sekvenční toky procesu.

Artefakty – Poznámka

Poznámky představují způsob, jakým může uživatel poskytnout doplňkový informační text pro čtenáře procesního diagramu.



3 Teoretická východiska

3.1 Testování

V publikaci s názvem ***Efektivní testování softwaru: Klíčové otázky pro efektivitu testovacího procesu*** se v úvodu nachází tato deklarace, podle níž se dá usuzovat, že se jedná o záležitost netriviálního charakteru [5]:

„*Testování představuje průměrně 20 % až 40 % pracnosti v projektech vývoje softwaru. Pro vývoj řídicího softwaru s vysokými požadavky na spolehlivost může pracnost testování často přesáhnout pracnost vývoje. Pokud má být testování efektivní, musíme k němu přistupovat jako k samostatné odborné disciplíně – stejné, jako je analýza, návrh, architektura, vývoj nebo projektové řízení. Klíčem k úspěchu je vyvážená kombinace metodiky a praktických zkušeností*.“

Pro většinu lidí, kteří nejsou zběhlí v problematice vývoje softwaru, a tudíž nemají dostatek relevantních zkušeností, se bude testování vždy jevit jako „pouhé klikání“. Pragmatičtěji založení lidé v něm však spatřují komplexní činnost a disciplínu determinovanou kontextem, ve kterém je prováděna. Nejčastěji je kontext vztažen k prostředí konkrétní společnosti a konkrétního projektu, jelikož každá společnost funguje v daném odvětví lidské činnosti, které je vždy určující z hlediska charakteristik pro vývoj softwaru. Těchto charakteristik je bezpochyby nespočet, ale mezi základní může patřit například to, jestli se firma zabývá vývojem typového software, jestli vyvíjí software pro tisíce zákazníků či pro jednoho, jestli špatně otestovaný software nemůže mít vliv na zdraví koncového uživatele nebo například jestli je potřeba uvést na trh produkt co nejrychleji v minimální životaschopné formě neboli minimal viable product. [5]

Procesem testování se rozumí sled a vzájemné vztahy činností, které s testováním souvisí. Takové činnosti jsou prováděny většinou specializovanými rolemi, ale mohou se na nich samozřejmě podílet a podílejí i role jiné. Například vývojářské testování, při kterém vývojáři provádí jednotkové testování, je rozlišeno jako testování závislé. Takové testování má tedy specializaci jinou než testovací. Naproti tomu nezávislé testování je prováděno rolí na testování specializovanou, ale může se jednat i zákazníka provádějící tzv. akceptační nebo beta testování. Tyto role tedy testují výsledky práce rolí ostatních. [1]

Otázkou ale není jak se v té a té dané společnosti testuje nebo jakou užívají metodiku, nýbrž jak se u nic vyvíjí software? Dříve bylo testování z hlediska vývoje velmi často vnímáno jako činnost, co se dělá až „úplně na konci“, což je přístup, který vede k poměrně velké neefektivitě. Dnes ale známe modernější přístupy k testování, které sází na úzkou provázanost testování, vývoje a dalšími činnostmi prováděnými vývojovými týmy. Na oblast testování již tedy není nahlíženo jako na samostatnou a je tak potřeba uvědomit si, že sice určité činnosti jsou jen a pouze na rolích testovacích, ale je zde i několik činností, kde se testovací role neobejde bez vstupů od projektových manažerů, od oddělení řízení rizik nebo od vedoucího vývoje či vývojového týmu. Je tedy nasnadě parafrázovat původní otázku do formy „Jaký model či metodika je využívána v životním cyklu vývoje systému či softwaru?“. Odpověď na tuto otázku totiž jasně předurčuje faktory vedoucí k sestavení testovací filozofie společnosti. [5]

3.1.1 Úrovně testování

Alena Buchalcevová a Jan Kučera ve svém článku s názvem ***Hodnocení metodik vývoje informačních systémů*** *z pohledu testování* charakterizují určité úrovně testování [1]:

„*Úroveň testování je definována jako sada testů, kterými je softwarový produkt testován na definované úrovni podrobnosti. Rozeznáváme jednotkové testování, které testuje nejmenší části systému, integrační testování, které se zaměřuje na testování vazeb a propojení částí systému, systémové testování, tedy testování systému jako celku, a akceptační testování, jehož cílem je prokázat splnění definovaných akceptačních kritérií. V rámci tohoto kritéria se zjišťuje, které úrovně testování metodika pokrývá, jaké role se podílejí na testování na dané úrovni a jaká je intenzita testování.*“

Zde si dovolím v návaznosti na předchozí odstavec autocitovat svou bakalářkou práci s názvem ***Automatizace testovacích scénářů***, kde jsem za pomocí zdrojů definoval výše zmíněné úrovně testování takto [6] [7] [8]:

- **Testování programátorem** – prováděno hned po naprogramování úseku kódu. Vždy jej však nemusí provádět sám programátor, který úsek vytvořil, ale může být vykonáváno například programátorem seniorem.

- **Jednotkové testování** – po předchozí fázi přichází na řadu ověření jednotek. Při objektově orientovaném programování (OOP) jsou těmito jednotkami samostatné třídy a metody. Testování jednotek pak obnáší otestování každé metody či třídy zvlášť tak, aby nebyla ovlivněna okolním prostředím.
- **Integrační testování** – při této fázi nastupuje tester či testovací tým. Ověřuje se bezchybná komunikace mezi komponentami systému. Integrační testy mohou být manuální, a to v případě nových funkcionalit systému nebo automatické v případě stávajících funkcionalit.
- **Systémové testování** – po otestování jednotlivých komponent se testuje systém jako celek. Systémové testování je tedy stěžejní pro předání aplikace zákazníkovi. Testy jsou prováděny v pozdější fázi vývoje a také jsou připraveny testovací scénáře, které obsahují reálné průchody aplikací.
- **Akceptační testování** – pokud se během předchozích fází nevyskytnou žádné problém nebo jsou opraveny, dojde k předání aplikace zákazníkovi. Ten pak na své straně realizuje testování svým testovacím týmem, který testuje podle scénářů vytvořených v kooperaci s dodavatelem aplikace. Testování aplikace probíhá na straně klienta a při nalezení chyby se reportuje zpět dodavateli, který musí chybu opravit. Princip je potom podobný jako u systémového testování, tedy ve více kolech, kdy jsou reportovány chyby a jsou nasazovány jejich opravy v pravidelných časových intervalech.

3.1.2 Typy testů

Dále je vhodné rozdělit testy samotné. Testy kategorizujeme podle oblastí, které testují z hlediska kvalitativních ukazatelů [1] [6] [9]:

- **Testy funkčnosti** – tyto testy ověřují, že požadované funkce byly správně naimplementovány. Jsou využívány nejčastěji v integrační, systémové a akceptační úrovni testování.
- **Testy použitelnosti** – zaměřují se pouze na přívětivost uživatelského rozhraní, na jeho dokumentaci, kvalitu výukových materiálů nebo oblast nápovědy.

- **Výkonnostní testy** – vystavují systémy situacím s velkou zátěží a testují jejich správný a stabilní průběh.
- **Testy spolehlivosti** – zaměřují se na schopnost systémů zvládat nestandartní podmínky jako jsou chybné vstupy nebo stavy obnovy po havárii systému.
- **Testy podpory** – testují oblasti jako je možnosti konfigurace systému, možnost nasazení na různých platformách, možnosti rozšíření nebo údržby.
- **Regresní a progresní testy** – progresní testy se zaměřují na oblasti nových funkcí softwaru a je potřeba přesná specifikace k jejich exekuci. Regresní testy se využívají při opětovném testování vlastností a funkcí. Jejich účelem je zjistit, zdali nové funkce neměly nežádoucí vliv na ostatní části softwaru.
- **Smoke testy** – neboli zahořovací testy; jedná se o určité vybrané podmnožiny vytvořených nebo plánovaných testů, často mají i regresní povahu. Jejich cílem je ověření správné funkčnosti všech kritických vlastností systému.

3.1.3 Artefakty testování

Alena Buchalcevová a Jan Kučera ve svém článku srovnávají metodiky z hlediska několika kritérií. Dokumenty, které definuje *standard IEEE 829* [10], jsou jedním z kritérií. Jelikož se některé pojmy vyskytují napříč více zdroji, ale i praxí, rozhodl jsem se je v této práci také zmínit. Jejich výčet a stručnou definici obsahuje následující tabulka [1]:

Test Plan	Popisuje, co bude testováno, použité testy, techniky, nástroje, data a požadavky na kvalitu a harmonogram
Test Design Specification	Shrnuje a blíže vymezuje co má být testováno a kritérium pro posouzení úspěšnosti testování
Test Case Specification	Popis testovacího případu; obsahuje vstupní a očekávaná výstupní data a jednotlivé kroky případu
Test Procedure Specification	Popisuje jednotlivé kroky při exekuci test casu
Test Item Transmittal Report	Obsahuje popis změn proti předchozí verzi softwaru
Test Log	Zachycuje průběh test casů a jejich výsledků
Test Incident Report	Popisuje pozorované neshody, kroky a okolnosti
Test Summary Report	Souhrnná zpráva o průběhu celého testování

Tabulka 1: Přehled dokumentů dle standardu IEEE 829

3.1.4 Metodiky testování, vývoje a provozu

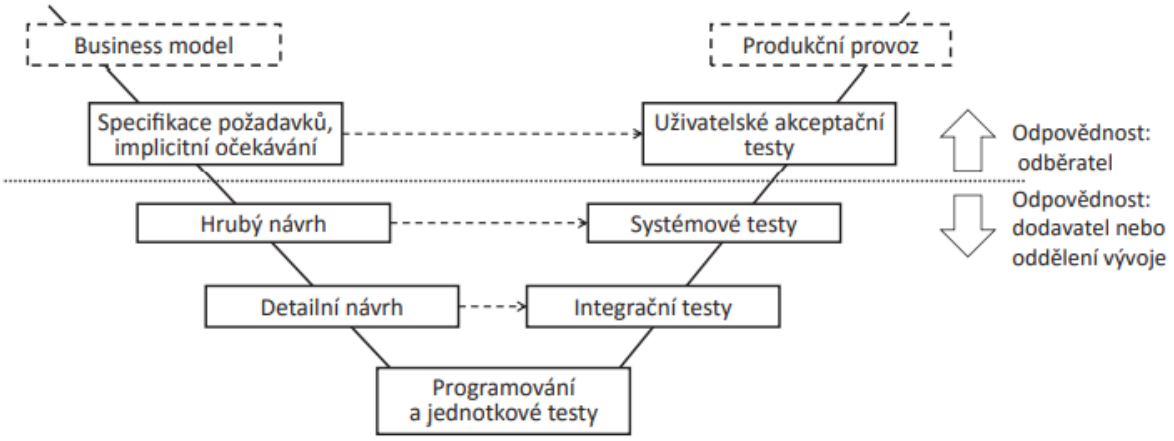
Slovo metodika či model v souvislosti s vývojem a testováním zde již bylo zmíněno. Co to tedy znamená? Formální definice dle zdroje [5] zní takto:

„Metodika je souhrn vodítek a principů, které mohou být přizpůsobeny a aplikovány na řešení určité situace. Může jít jak o jednoduchý soupis akcí, které se mají vykonat, tak o specifický přístup, šablony, formuláře, popřípadě kontrolní seznamy.“

O metodice můžeme tedy hovořit jako o konkrétním postupu či návodu, který můžeme použít v určité oblasti činností v průběhu životního cyklu systému. [5]

3.1.4.1 V-model a W-model

Středobodem mnoha metodik a strategií užívaných společnostmi je V-model. Jedná se o model životního cyklu systému s důrazem na oblast testování. Jeho schématické vyobrazení vypadá takto:

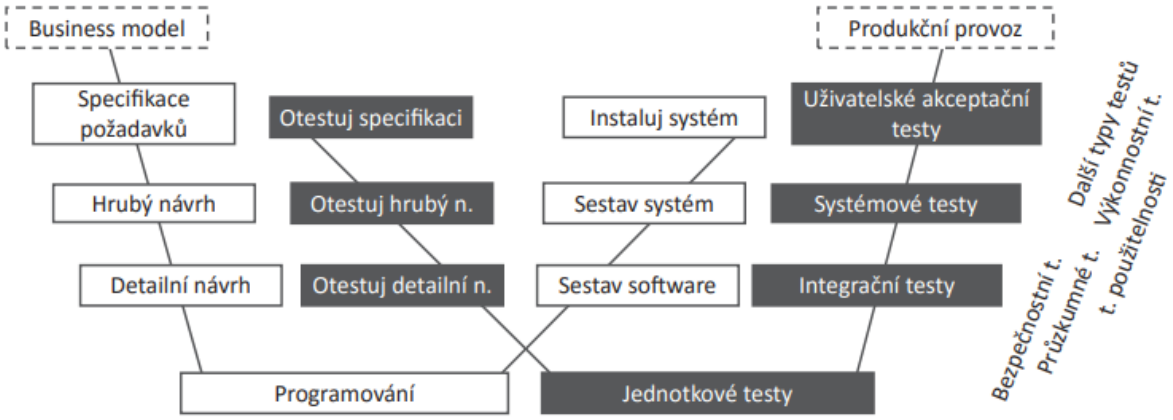


Obrázek 1: V-model [5]

Princip fungování V-modelu je velice podobný všeobecně známému vodopádovému modelu. V model tvoří několik dvojic aktivit jako například *specifikace požadavků → uživatelské akceptační testy, hrubý návrh → systémové testy, detailní návrh → integrační testy* a *programování → jednotkové(unit) testy.* Levá strana tedy čítá aktivity spojené se specifikací a návrhem, případně implementací a pravá testování. Tím je zdůrazněná role testování v životním cyklu. Naznačená hranice odpovědnosti je pouze orientační, nikoliv pevně stanovená. Jedná se však o nejčastější rozdělení, nikoliv jedinou možnost.

V souhrnu V-model definuje, že nestačí pouze správný návrh a správná implementace, ale v určitém okamžiku je třeba zařadit nástroj testování pro ověření postupu při návrhu a realizaci. Z obrázku jsou pak patrné jednotlivé úrovně testování, tedy od horní úrovně směrem ke spodní – *akceptační testování, systémové testování, integrační testování a jednotkové testování.* V-model je ale značně idealizovaný, jelikož v praxi ne zřídkakdy dochází k rychlému spádu až do fáze implementace a pravou část už se mnohdy ani nedostane z časových či jiných důvodů nebo dostane, ale až se zpožděním. Nejedná se však přímo o problém modelu, ale spíše o uvažování některých manažerů a vývojářů. S V-modelem se ale nesou další nedostatky, například značně idealizované vztahy mezi jednotlivými úrovněmi návrhu a testování. Druhý nedostatek se váže na nedoceňování potenciálního přínosu testerů už v procesu vývoje svým nezávislým názorem například na kvalitu kódu.

Možným řešením v mnoha případech může být W-model, který tyto nedostatky V-modelu eliminuje. Tento model klade důraz na včasné zapojení testerských rolí již v raných fázích vývoje. Také nabízí přímé souvislosti vývojové a testovací aktivity a zásadně tak rozšiřuje chápání a pojetí testování ve smyslu zkoumání softwaru při běhu. W-model také obsahuje typy testů, které nejsou vztažené k žádné konkrétní úrovni testování (zobrazené po pravé straně obrázku 2). Schéma W-modelu ukazuje obrázek 2:



Obrázek 2: W-model [5]

Model typu W ale také není bez nedostatků. V praxi může být argumentováno proti jeho zavedení náklady nebo nedostatkem času a vcelku oprávněně. Častějším nedostatkem je případ, kdy jsou požadavky zadány pouze mlhavě ať už záměrně či nikoliv. [5]

3.1.4.2 **Agilní metodiky**

V ICT odvětví existuje poměrně velké množství agilních metodik, které mohou jednotlivé společnosti používat. Přinesly s sebou velkou změnu v procesu testování díky vlastní podstatě agilních metodik, ale rozhodně to nelze chápat tím způsobem, že dostaneme jasný návod na agilní testování. Zásadní změny však nastávají v oblasti vztahů mezi vývojáři a testery, kdy se středobodem veškerého zájmu stává kvalita a tím pádem se stává i zodpovědností celého týmu. To má pochopitelně dopad i na vzorce chování týmů, jejich zájmy a vztahy. Společnosti dnes již volí spíše testery-univerzály, kteří zastávají procesy automatizace, ale i původní činnosti zastávané testerem či manažerem testování. Nejlepším modelem v oblasti agility se ukazuje forma a model sdílení informací, znalostí a zkušeností a tím pádem se z týmů stávají čím dál tím častěji držitele pomyslné moci ve společnostech, což se nutně nemusí líbit ani vyhovovat všem společnostem. [5]

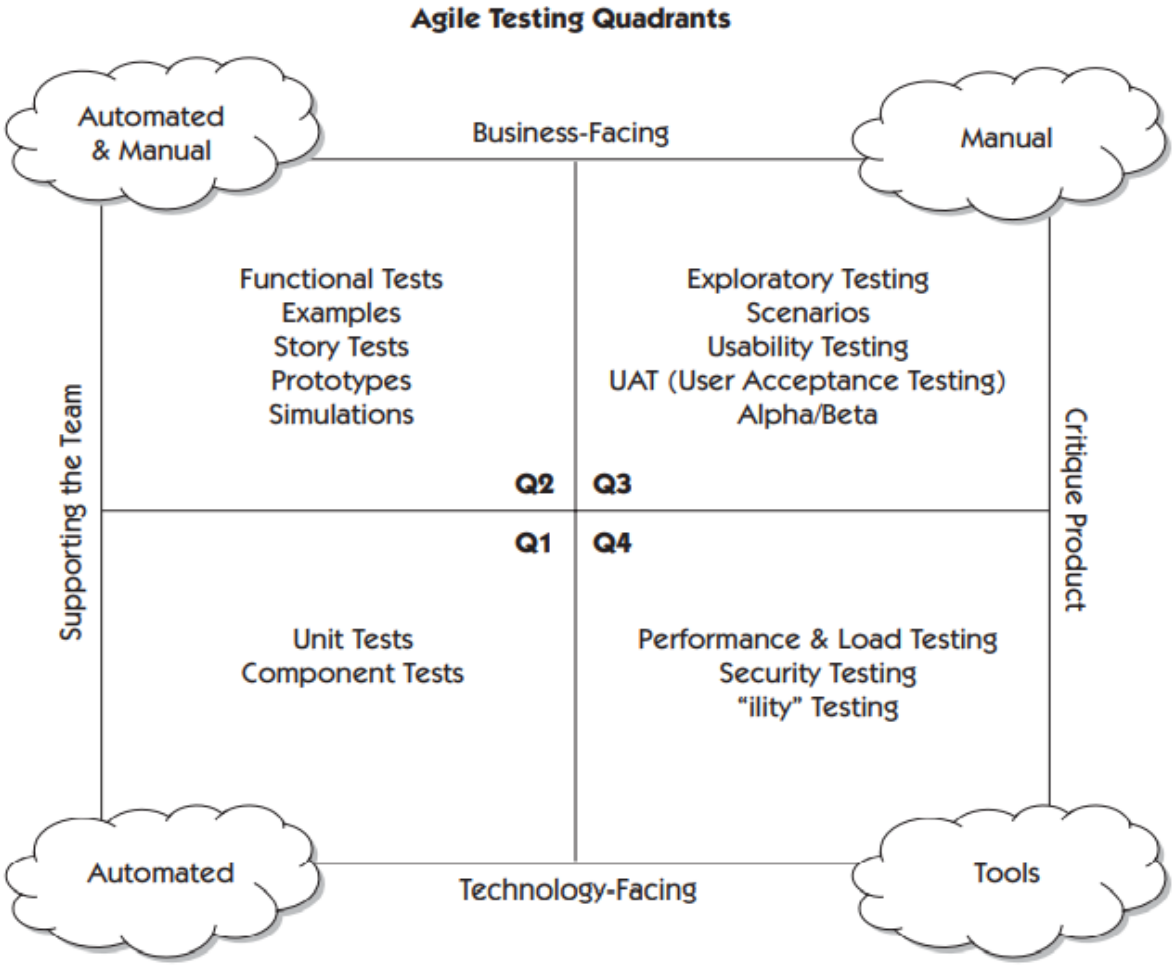
Následující tabulka poskytuje souhrn základních charakteristik mezi tradičními metodikami (nejčastěji vodopádový životní cyklus) a agilními metodikami [5]:

	Tradiční metodiky	**Agilní metodiky**
Logika fungování vývojového týmu	Předem pevně definovaná	Empirická (ad-hoc modifikovatelná)
Stupeň formalizace procesů	Vysoký	Nízký
Zdroj procesního know-how	Standardizace Dobré praktiky v odvětví jako celku	Každý tým si najde nejlépe vyhovující individuální přístup Neformální sdílení napříč týmy v organizaci
Profesní tradice	Inženýrství	Řemeslo
Použití	Zejména komplexní projekty a distribuované týmy	Zejména malé týmy nacházející se na jednom místě

Metafora	Stavba kancelářské budovy s využitím několika dodavatelských firem	Stavba rekreační chaty svépomocí
Klíčové sociální faktory	Plánování a kontrola	Zmocňování (empowerment) a důvěra
Obecná rétorika	Plán, proces, trasovatelnost, standardizace, bezpečnost	Improvizace, inovace, hodnota pro uživatele, rychlost, spolupráce
Typický pohled na časovou stálost	Projekt = časově ohraničený úsek	Produkt = dlouhodobý rozvoj
Požadavky uživatelů	Předem pevně dané (změny podléhají složitému řízení)	Modifikovatelné před každou iterací
Zapojení uživatelů	Začátek a konec vývojového procesu → nároky na promyšlenost hned od počátku	Průběžné, např. po každé iteraci → nároky na čas uživatelů a přijetí osobní odpovědnosti za rozhodování
Zásadní riziko	Vyvíjíme něco, co uživatel vůbec nechce, a zjistíme to příliš pozdě	Zbrklý přístup může způsobit nákladnost řešení a nesoulad s podnikovou architekturou
Organizace vývoje a testování	Hierarchická	Samoorganizující se tým
Organizační filosofie	Management	Leadership každého člena (např. role Scrum Mastera)
Úloha testerů	„Strážci brány – ochránci uživatelů“	„Průvodci programátorů v džungli kvality“
Specializace testerů	Specialisté (test manažer, test analytik, tester, vývojář automatizovaných testů…)	Univerzalisté nazývaní např. inženýři testování/kvality
Externí vlivy posilující pozici	Tradice a zákonné regulace a audit (motivovány požadavky na bezpečnost a spolehlivost)	Tradice Zákonné regulace a audit (motivovány požadavky na bezpečnost a spolehlivost)

Tabulka 2: Srovnání určitých aspektů tradičních a agilních metodik

Lisa Crispin a Janet Gregory definují ve své publikaci „Agilní testování“ určité kvadranty agilního testování. Diagram zobrazuje kvadranty popisující různé důvody k testování. Na jedné ose jsou vyznačeny testy, které jsou kritické pro produkt a které podporují celý agilní tým. Na druhé ose jsou rozlišené dle obchodní roviny a dle roviny technologické. Kvadrant **Q1** reprezentuje tzv. Test-Driven Development (TDD neboli programování řízené testy), což je stěžejní praktika agilního vývoje, kdy unit testy testují funkcionality malých částí systému a jsou vytvořeny programátorem jako první. Teprve pak vzniká kód samotný. Kvadrant **Q2** také reprezentuje podporu vývojového týmu, ale z vyššího pohledu. Jedná se také o formu TDD, ale na vyšší funkční úrovni z pohledu původních obchodních požadavků a jejich účelem je tak potvrdit požadované chování systémů. Jedná se tedy o akceptační testování. Kvadrant **Q3** je čistě obchodně orientovaný a testy mají za úkol určit, zda se povrchně potkávají požadavky s výslednou realizací. Taktéž se jedná o úroveň akceptačního testování. Do kvadrantu **Q4** spadají technologicky orientované výkonnostní testy jako jsou zátěžové, robustní či bezpečnostní testy. [11]

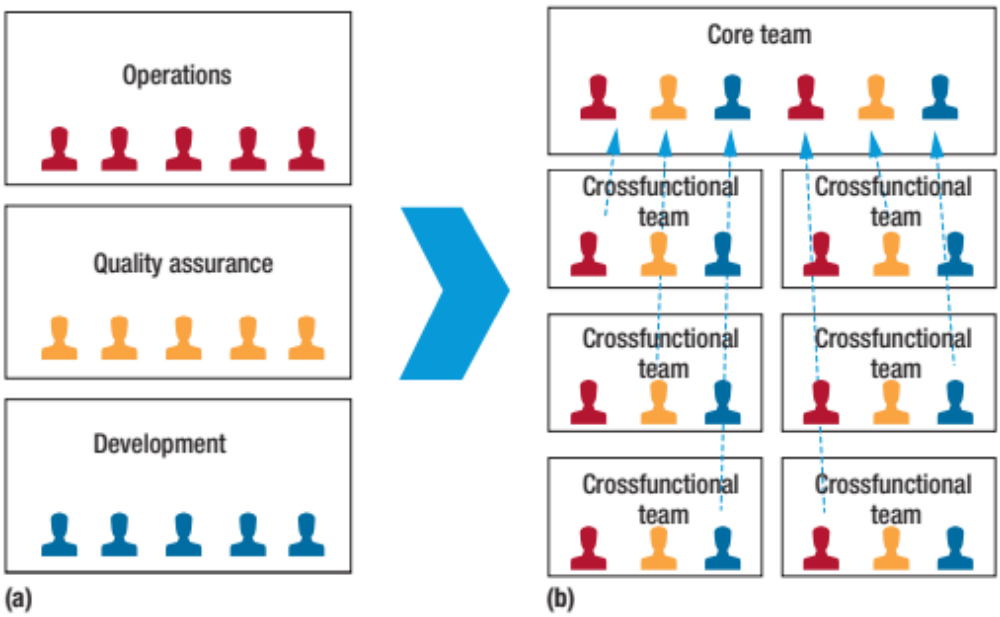


Obrázek 3: Kvadranty agilního testování [11]

3.1.4.3 **DevOps**

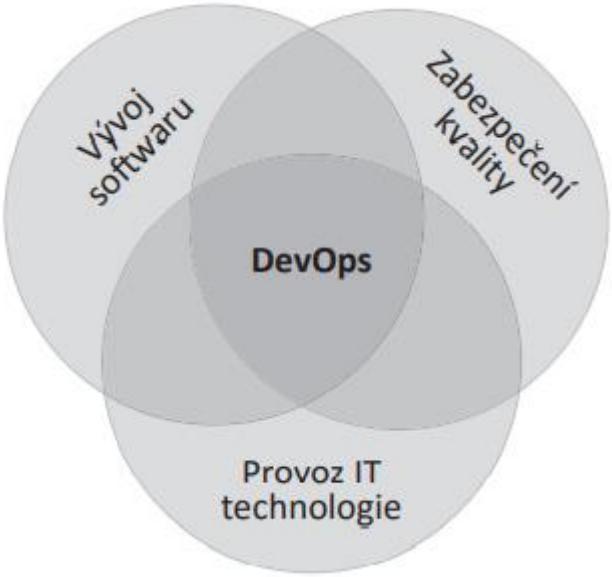
Způsob práce zvaný **DevOps** je metodika profesního a kulturního hnutí, která zdůrazňuje nutnost odstranění všech bariér mezi oddělením vývoje a provozu a testování ve společnosti. Zkratka DevOps totiž vychází ze slov ***Dev****elompent* a ***Op****erations* neboli vývoj a provoz v jednom. Jedná se vlastně o přirozené rozšíření konceptů agilních metodik do oblasti provozu systémů. Při procesu transformace směrem k této metodice je tzv. CAMS neboli Culture – Automation – Measurement – Sharing (Kultura – Automatizace – Měření – Sdílení) ve významu Kultura = změna organizační struktury, Automatizace = maximální automatizace vývojových, testovacích a provozních prostředí, Měření = důraz na průběžná měření a Sdílení = důraz na sdílení odpovědnosti ale i problémů. [5]

Následující obrázek ukazuje tvoření týmů a změnu organizační struktury do metodiky DevOps. Složení a) představuje tradiční „horizontálně“ stavěné týmy a b) zachycuje „vertikální“ týmy dle DevOps, kde je každý tým zodpovědný za své služby a tým je složen z lidí mající různé schopnosti. Členové týmů spolupracují na projektu od začátku s cílem vytvořit více hodnot pro specifického koncového uživatele. Na vrcholu modelu se nachází tzv. Core tým, který má přehled nad všemi interakcemi v systému a je rozhodujícím článkem v kritických otázkách architektonického rozhodování. [12]



Obrázek 4: Formování týmů dle DevOps [12]

Jak tedy napovídá i samotný název DevOps, jde o provázání vývoje softwaru a provozu IT technologií. Vzhledem k tomu, že vývoj je neustálým zdrojem změn už jen ze své podstaty a provoz usiluje o co nejméně změn, aby dosáhl co nejstabilnějšího prostředí pro své uživatele, je třeba hledat kompromisní řešení. DevOps má snahu tyto protikladné požadavky co nejvíce optimalizovat a nalézt jejich řešení. Aby bylo dosaženo stability, je nutné zajistit dostatečnou kvalitu dodávaného řešení. Následující obrázek ukazuje klasickou provázanost signifikantní právě pro DevOps. Průnik okruhů symbolizuje odstranění komunikačních hranic a nastolení spolupráce mezi těmito okruhy. Díky efektivnější komunikaci je možné rychleji a lépe reagovat na změnové požadavky, ale i na situaci na trhu. [5]



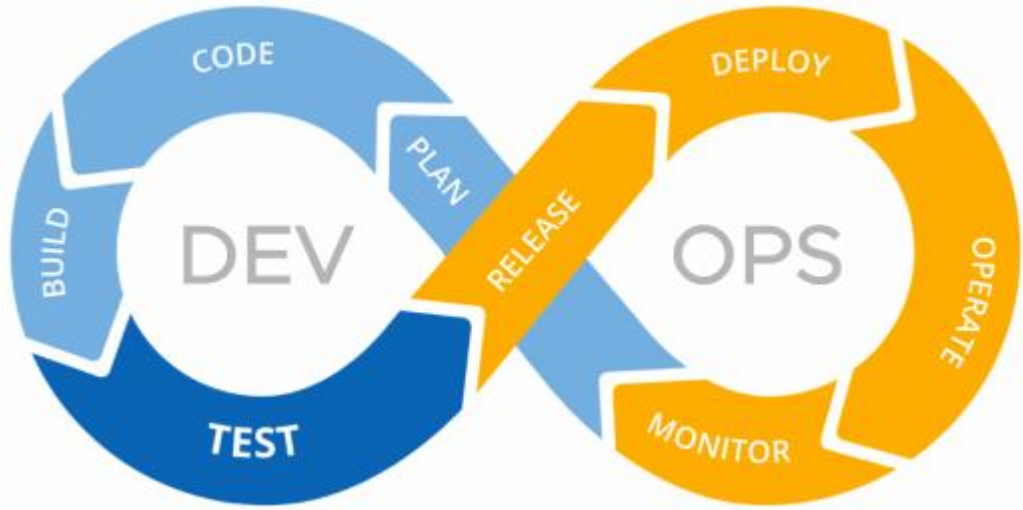
Obrázek 5: Průnik složek definujících DevOps [5]

V rámci zajištění spolupráce k DevOps metodice neodmyslitelně patří prvek zpětné vazby mezi jednotlivými kroky či praktikami. Takovými praktikami jsou zpravidla [5]:

- **Plánování funkcionalit –** dřívější modely, kdy bylo třeba analyzovat všechny změny najednou jsou nahrazeny plánováním dílčích funkcionalit
- **Průběžná integrace (CI) –** způsob řešení vývoje komplexních systémů různorodými týmy. (Tomuto tématu je věnována jedna z kapitol této práce společně s (CD) průběžnými dodávkami)

- **Průběžné dodávky** (CD) – princip průběžné integrace je podmíněn schopností realizace průběžných dodávek. (Tomuto tématu je věnována jedna z kapitol této práce společně s (CI) průběžnou integrací)
- **Průběžné testování –** kromě samotných testů zahrnuje ještě v ideálním případě automatizované zajištění testovacího prostředí s ideálně automatizovanou správu testovacích dat.
- **Průběžný monitoring, zpětná vazba a zlepšování –** Je kladen důraz na monitorování a interpretaci výsledků monitoringu z obchodního pohledu.

Následující obrázek zachycuje řetězec jednotlivých DevOps praktik. Tato DevOps smyčka zachycuje průběžný tok mezi částí vývoje (plánování, tvorba kódu, kompilace, testování) a provozní částí (uvolnění verze, nasazení, provoz, monitorování). [13]:



Obrázek 6: Řetězec DevOps praktik [13]

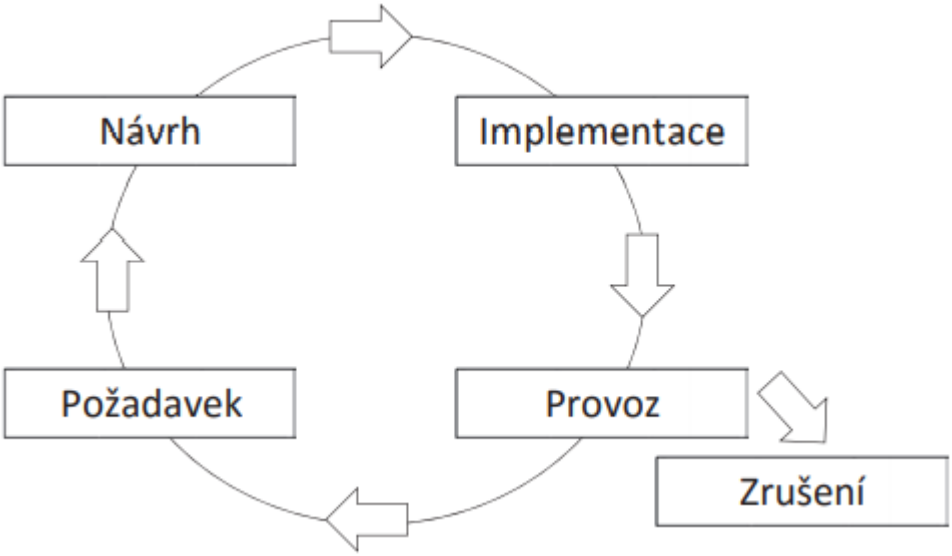
3.1.5 Testovací Prostředí / Typy prostředí

Pojem testovací prostředí zde již byl zmíněn a pro účely této práce je nezbytné zmínit i další pojmy vysvětlující různé druhy prostředí. Publikace „Efektivní testování softwaru“ uvádí tuto definici [5]:

„Jak můžeme najít ve slovníku ISTQB, pojmem „testovací prostředí“ je míněno „prostředí“ obsahující hardware, software, simulátory, nástroje a další vybavení potřebné pro vykonání testů. Toto pojetí je všeobjímající.“

Skrze tuto definici si tedy testovací prostředí můžeme představit jako soubor veškeré IT infrastruktury potřebné pro simulaci klientského prostředí jako jsou počítače, mobilní zařízení, periferie, operační systémy, databázové servery, síťové komunikační prvky atd. Zkrátka jde o sestavení co nejvíce odpovídajících podmínek z komponent, které se blíží reálné situaci v provozu. Jsou zde ale také přítomné nástroje, které se do prostředí provozu nehodí či nejsou žádoucí. Takových nástrojů existuje celá řada, ale pro představu se může jednat o různé generátory požadavků nebo dat, nástroje pro automatizované funkční testy, nástroje pro zátěžové testy, nástroje pro přístup k databázím atp. [5]

Životní cyklus testovacího prostředí vypadá následovně:



Obrázek 7: Fáze životního cyklu testovacího prostředí [5]

Testovací prostředí můžeme rozlišovat hned podle několika obecných atributů jako jsou **Název**, **Zkratka**, **Funkce**, **Shodnost s produkčním prostředím**, **Výkonnost**, **Správa prostředí** a **Integrovanost.** Následující tabulka představuje přehled typů prostředí a jejich

hodnoty těchto atributů. Problematika prostředí je značně komplexní, ale pro účely této práce postačí pouze tento základní přehled [5]:

Název prostředí	Zkratka	Funkce	Shodnost s produkčním prostředím	Výkonnost	Správa prostředí	Integrovanost
Pískoviště	–	Prototypování, ověřování vývojových a testovacích nástrojů…	Ne	Ne	Oddělení vývoje	Ne
Vývojové (Development)	DEV	Vývojové testy – jednotkové testy, jednotkové integrační testy, testy integrity verze	Ne	Ne	Oddělení vývoje	Ne
Systemtest (Systemtest)	SYS	Vývojové testy integrace, systémové testy – finální testy vývojového týmu	Ano (SW platformy)	Ne	Oddělení vývoje	Ano (pro vývojové testy integrace)
Integrační (Integration)	INT	Systémové integrační testy a uživatelské akceptační testy	Ano (SW platformy)	Ne	Oddělení provozu	Ano
Předprodukční (Preproduction)	PRE	Zátěžové testy, technické akceptační testy, testy nasazení	Ano (SW i HW platformy)	Ano	Oddělení provozu	Ano
Podpora produkce (Production support)	PRS	Simulování produkčních chyb, testování jejich oprav, testy drobných (konfiguračních) požadavků	Ano (SW platformy)	Ne	Oddělení provozu	Ano
Školící (Education)	EDU	Školení uživatelů	Ne	(Ano)	Oddělení provozu	Ano
Produkce a záloha (Production and backup)	PRO	Produkční prostředí	–	Ano	Oddělení provozu	Ano
	BAC	Záložní prostředí	–	Ano	Oddělení provozu	Ano

Tabulka 3: Typy prostředí a jejich atributy

3.1.6 Zajištění kvality (Quality Assurance)

Koncepty testování a řízení kvality je třeba rozlišovat a správně pojmenovávat. Testování totiž není to samé, co zajištění kvality (angl. QA – Quality Assurance) a mezi těmito pojmy existují jasné významové rozdíly.

„*Podle glosáře ISTQB je zajištění kvality chápáno jako součást konceptu řízení kvality, což je plánovaná a koordinovaná aktivita zasahující celou organizaci a zahrnující koncepty, jakými jsou např. politika kvality, cíle kvality, plánování kvality, kontrola kvality, zajištění kvality a zlepšování kvality. Koncept zajištění kvality jednoduše zahrnuje daleko více očekávání od role QA specialistů než od role testerů. Aby bylo možno kvalitu skutečně zaručit, je třeba mít kontrolu nad vývojovými procesy jako celkem, oprávnění činit nepopulární rozhodnutí (např. zastavit vývoj či nasazení určitého systému),*

(spolu)rozhodovat o personálních opatřeních v případě nedodržení definovaného procesního standardu apod. Roli správně fungujícího QA oddělení lze výstižně přirovnat k roli hlídacího psa: hlídací psi nejen štěkají, když se děje něco nepravého; mohou rovněž kousnout.“ [5]

V praxi se můžeme setkat se třemi běžně užívanými způsoby fungování oddělení testování, které můžeme charakterizovat následujícími názvy [5]:

- **Oddělení kontroly kvality –** Jedná se o čistě reaktivní přístup. Inklinuje k chápání vývoje jako výroby, na jejímž konci stojí kontrolor kvality a teoreticky má možnost zastavit celou „výrobní linku“. Pro oblast softwaru je tento způsob chápání ale krajně nevhodný.
- **Oddělení zajištění kvality –** Jedná se o proaktivní přístup. Definuje standarty, nastavuje strategii, stará se o trénink zaměstnanců v problematice prevence defektů.
- **Oddělení testovacích služeb –** Slouží jako podpora projektového manažera. Nemá rozhodovací schopnost, pouze poskytuje dokumentaci manažerovi, který na základě ní může či nemusí jednat.

Anna Borovcová se ve svém článku na téma „**Kvalita a testování v České republice**“ zabývala hledáním odpovědí na zásadní otázky typu [14]:

„Jak je vnímáno testování softwaru v České republice?“,
„Jakou pozici testování, potažmo řízení kvality, zastává ve firmách?“,
„Jak je prováděno testování softwaru v České republice?“,
„Zejména, jaké techniky jsou voleny a jak vypadá testovací tým?“

V článku je zkoumán a analyzován stav testování ve společnostech v České Republice, které se zabývají vývojem softwaru formou dotazníkového šetření mezi celkem 300 firmami. Z celkem 84 respondentů bylo 41 členy testovacího týmu, 16 projektových amanžerů a 27 pracovníků v rolích vývojářů a analytiků. Důležitá je také optika velikosti dotazovaného podniku. Zhruba 48 % dotazovaných byli zaměstnanci menších firem do 100 zaměstnanců a zbylých 52 % potom zaměstnanci firem nad 100 zaměstnanců.

Položena jim byla stěžejní otázka *„Která z následujících definic nejvíce odpovídá vašemu chápání testování?“*. Odpovědi dotazovaných ukazuje následující tabulka [14]:

Která z následujících definic nejvíce odpovídá vašemu chápání testování?	
Testování je proces hledání chyb	9,5 %
Testování je proces zjišťování informací o kvalitě	16,7 %
Testování je kontrola produktu oproti jeho specifikaci	26,2 %
Testování je proces sloužící k zajištění kvality	47,6 %

Tabulka 4: Definice testování dle účastníku dotazníku

Výsledky ukazují že téměř polovina dotazovaných se domnívá, že testování je proces sloužící k zajištění kvality, což je poměrně zajímavé, protože testování samo o sobě kvalitu nezajišťuje. 35,7 % dotazovaných si ale na základě zvolení nejednoduché odpovědi uvědomuje, že je testování součástí komplexnější disciplíny. Aby však otázka byla úplná, je potřeba zeptat se i z druhé strany. Dotazovaným byla položena následující otázka. Odpovědi na ni ukazuje tabulka 5 [14]:

„Která z následujících definic nejvíce odpovídá vašemu chápání zajišťování kvality?

a) Nevím přesně co si pod tímto pojmem představit

b) Zajištění kvality je honosnější název pro testování

c) Zajišťování kvality je přístup k procesům vývoje, který se snaží předcházet nedostatkům a minimalizovat tak náklady na jejich odstranění

d) Zajišťování kvality je komplexní přístup k testování, který se snaží zajistit všestranné protestování produktu a dohlédnout na správnost oprav nalezených chyb“

Zvolená odpověď	Procentuální zastoupení		
	(za firmy do 100 zaměstnanců)	(za firmy nad 100 zaměstnanců)	Celkem
a)	0 %	0 %	0 %
b)	7,5 %	0 %	3,6 %
c)	50 %	79,5 %	65,5 %
d)	42,5 %	20,5 %	30,9%

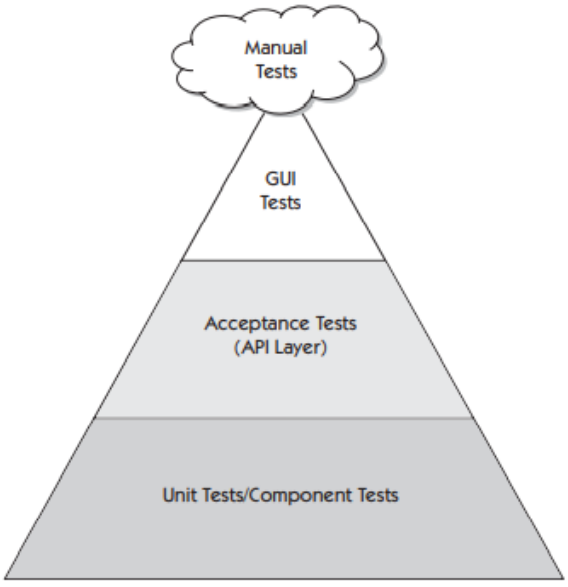
Tabulka 5: Procentuální odpovědi na dotaz ohledně definice zajišťování kvality

Z výsledků lze usuzovat, že zaměstnanci větších firem dokáží správně definovat pojem zajištění kvality. Polovina zaměstnanců menších firem se s tímto pojmem nesetkává nebo jim nebyl vysvětlen rozdíl mezi těmito dvěma disciplínami. [14]

3.1.7 Automatizace testování

Automatizovaný test se od manuálního liší tím, že je prováděn nějakým programem. Ten simuluje například kroky uživatele při funkčním testování nebo simuluju důsledky chování určitého počtu uživatelů v případě zátěžových testů. Automatizované testy je možné vytvářet v každé úrovni testování, nejčastější je ale úroveň integrační či akceptační z pohledu UI front-end aplikace nebo End-to-End testů (E2E testy). Automatizovaný test vykonává testování „zdarma“ a relativně rychle a je ho možné opakovat mnohem častěji. Navíc není náchylný na únavu a lidský faktor chyby. Akce provedené takovým programem jsou vždy stejné a je možné tedy očekávat stále stejné výsledky. Pokud se ale chování systému liší, automat se s tím sám nevypořádá, pokud na to není naprogramován. Druhá rovina je že má automat omezenou možnost analýzy chybného chování testovaného systému, a tak je v takových případech nezbytná přítomnost specializovaného pracovníka. Proto i negativní výsledky musí ještě následně vyhodnotit člověk. Vlastní testování ale započíná už při samotném vývoji automatizovaného testu. Pro takovou implementaci ale už potřebujeme funkční uživatelské rozhraní ideálně nasazené na testovacím prostředí. Rozdíl je pak pochopitelně i v kvalifikaci, kdy manuální tester nevyžaduje hlubší programovací znalosti, automatizační tester již ano. [5]

V agilním prostředí je snaha o automatizaci velká, a tudíž je i snaha co nejvíce testů zautomatizovat. O typickém rozložení typů testů z hlediska stability a podpory systému v agilním prostředí vypovídá následující obrázek 8 [11]:



Obrázek 8: Pyramida automatizovaných testů [11]

Co vše můžeme automatizovat? [5] [11]

- Jednotkové testy / testy komponent – Pokud vývojáři používají metodiku TDD, vytváří mechanismus regresní povahy pro automatizované jednotkové testování. Pokud jsou automatizované unit testy, má tým a jeho produkt mnohem větší šanci na dlouhodobou udržitelnost
- Integrační testy / testy API a Webových služeb – Jedná se o testy založené na datech. Jsou většinou jednoduše udržitelné, většinou mají End-to-End povahu či povahu testování „*černé skříňky*".
- Testování GUI front-end aplikace – Ve většině případů se jedná o prostou simulaci aktivit manuálního testera. Ne všechny testy jsou vhodné automatizovat, a proto se volí nejčastěji varianta testů regresní povahy.
- Zátěžové testy – Některé typy zátěžových testů ani nemohou být provedeny bez automatizovaného přístupu, jelikož je zapotřebí najednou provádět akci nad aplikací a zároveň sledovat její výkon a chování.
- Tvorba dat nebo konfigurace – V některých případech, zejména v případech virtualizace je potřeba nejprve prostředí automatizovaně připravit. Proto je vhodné automatizovat konfigurační skripty a zároveň v některých případech můžeme rovnou i připravit testovací data.
- Průběžná integrace, sestavení aplikace a nasazení – Důležitost automatizovat tyto úlohy v agilních metodikách a v cloud přístupu je nezměrná a správná implementace kritická. Blíže je tomuto tématu jedna z kapitol této práce.

Co bychom automatizovat neměli? [5] [11]

- Testování použitelnosti – Samotná podstata testování použitelnosti už tento fakt potvrzuje, jelikož k řádnému testování použitelnosti potřebujeme fyzické uživatele, na kterých pozorujeme, jak si počínají.
- Průzkumové testování – Neboli exploratory testing, které vyžadují přítomnost zkušeného testera, který navrhne a následně provede testy nad systémem. Maximální hranice automatizace může být participace na vytvoření testovacích dat.
- Testy, které nikdy neselžou – Zde se například jedná o testy částí systému, kde požadavky k jejich vytvoření tvoří triviální úlohu.
- Testy, které se provedou pouze jednou či jen párkrát

Automatizovaný test by měl být hlavně dobře navržen, aby byl co nejefektivnější. Většina vlastností je pro všechny úrovně téměř stejná. Vlastnosti a jejich popisy obsahuje následující tabulka [5]:

Vlastnost	FE	API	Unit	Popis
Dostatečnost a zároveň stručnost	X	X	X	Test má být pokud možno stručný, ale zároveň má dostatečně pokrýt testovanou funkci nebo charakteristiky. Jinými slovy: jeho kód nemá obsahovat zbytečné části a samotný test kroky, které nejsou z logiky testu potřeba.
Co nejsnazší kontrola výsledku	X	X	X	Test by měl reportovat výsledky tak, aby následná manuální kontrola a analýza výsledku byla co nejsnazší. Co se týče jednoznačnosti výsledku, závisí na typu automatizovaného testu. U jednotkových testů by nás měl report z jejich běhu co nejpřesněji navést na místo kódu a datovou kombinaci, při které byl defekt odhalen. V případě defektu nalezeného front-end testem je ale manuální kontrola často efektivnější než složité programování kódu, který nalezený defekt kompletně automaticky vyhodnotí.
Opakovatelnost	X	X	X	Test by mělo být možné častokrát opakovat bez potřeby manuálního zásahu. Sníží se tím režie při spouštění automatizovaných testů.
Stabilita	X	X	X	Test by měl při stejném stavu testovaného systému, z pohledu přítomnosti defektů, produkovat vždy stejné výsledky. Na první pohled se zdá, že taková podmínka musí být splněna automaticky, ale není tomu tak. Co například v situacích, kdy běh testu závisí na nějaké konfiguraci systému nebo průběhu jiného testu (viz dále kritérium „Nezávislost")?
Srozumitelnost	X	X	X	Z kódu testu by mělo být na první pohled zřejmé, co dělá. Lze toho dosáhnout například komentáři, vhodným strukturováním, jmennými konvencemi, případně vhodnou externí dokumentací k testům.
Časová efektivita	X	X	X	Test by měl běžet po dobu, jež je přiměřená vzhledem k celkovému času, který budeme mít k dispozici na spuštění celé sady automatizovaných testů. Pokud tomu tak není, je vhodné testy optimalizovat, nebo jejich běh paralelizovat.
Nezávislost	–	–	X	V stejném stavu testovaného systému z pohledu přítomnosti defektů by test měl vrátit vždy stejný výsledek, bez ohledu na to, zda byl spuštěn samostatně nebo spolu s jinými testy. Princip nezávislosti by měl být dodržován všude, kde jej není potřeba z dobrého důvodu porušit. Takovým důvodem jsou třeba automatizované testy realizované jako sekvence jednotlivých akcí procesu za sebou, kde se řešení návaznosti jednotlivých kroků nevyhneme. Ale i v těchto situacích bychom se měli o nezávislost testů v rámci daných podmínek co nejvíce snažit.
Udržovatelnost	X	X	X	Test by mělo být možné snadno udržovat aktuální, pokud se testovaný systém mění. Je to prakticky jedna z nejdůležitějších vlastností automatizovaných testů.
Trasovatelnost	X	X	X	Pokud máme možnost k testu rychle najít, jaký požadavek a funkci (případně část kódu v případě jednotkových testů) testuje, nezanedbatelně nám to sníží režii celého testovacího procesu.

Tabulka 6: Základní charakteristiky dobrého automatizovaného testu

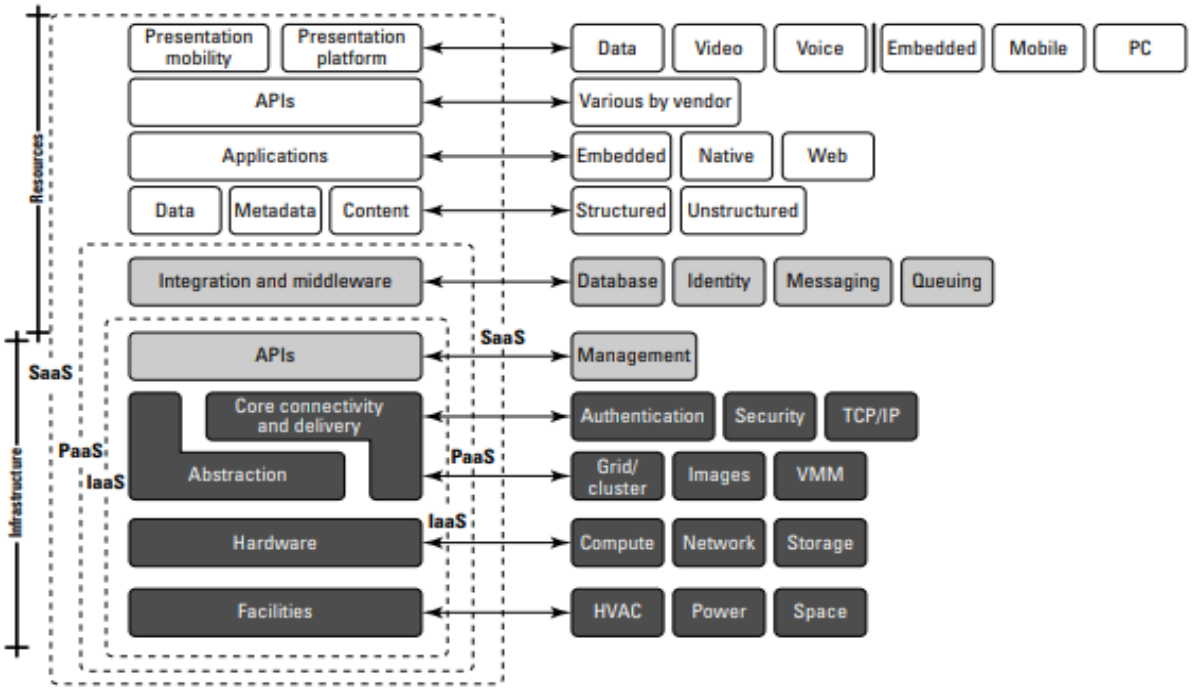
Největší smysl má automatizovat testy na nejnižších úrovních, jak ukazuje pyramida na obrázku 8. Automatizované testy mají řadu určitých charakteristik, které je potřeba znát pro zvýšení jejich efektivity. Nespornými výhodami je jednoznačnost automatizovaných testů, možnost častého opakování na různých platformách s různými daty. Nevýhodou je zase jejich přesnost, tedy že jsou naprogramované absolutně přesně ne jediný účel.

3.2 Cloud computing

Cloud computing (zkráceně cloud) ve zkratce znamená poskytování služeb a aplikací uložených na serverech uživatelům. K takovým službám a aplikacím uživatel může přistupovat například přes webový prohlížeč nebo klient elektronické pošty. V praxi se pak jedná o poskytování různorodých služeb jako jsou aplikace, operační systémy, ale třeba také celé infrastruktury. Pokud se jedná o placenou službu, uživatel platí za její užívání, ale za jeho vlastnění nikoliv. V základu je rozlišováno několik modelů, například distribuční (services model), který popisuje cloud z hlediska toho, jaké služby poskytuje, anebo model nasazování (deployment model), který cloud definuje podle způsobu poskytování služeb. [15]

3.2.1 Distribuční model (services model)

Distribuční model definuje druhy cloudů podle služby, kterou poskytují. Napříč literaturou se můžeme dočíst o docela různých modelech, všechny ale následují formu „*XaaS*“ nebo „*<Název> as a Service*“. Tři nejuznávanější modely se nazývají **IaaS** (Infrastructure as a Service – Infrastruktura jako služba), **PaaS** (Platform as a Service – Platforma jako služba) a **SaaS** (Software as a Service – Software jako služba).



Obrázek 9: Vyobrazení distribučního modelu [15]

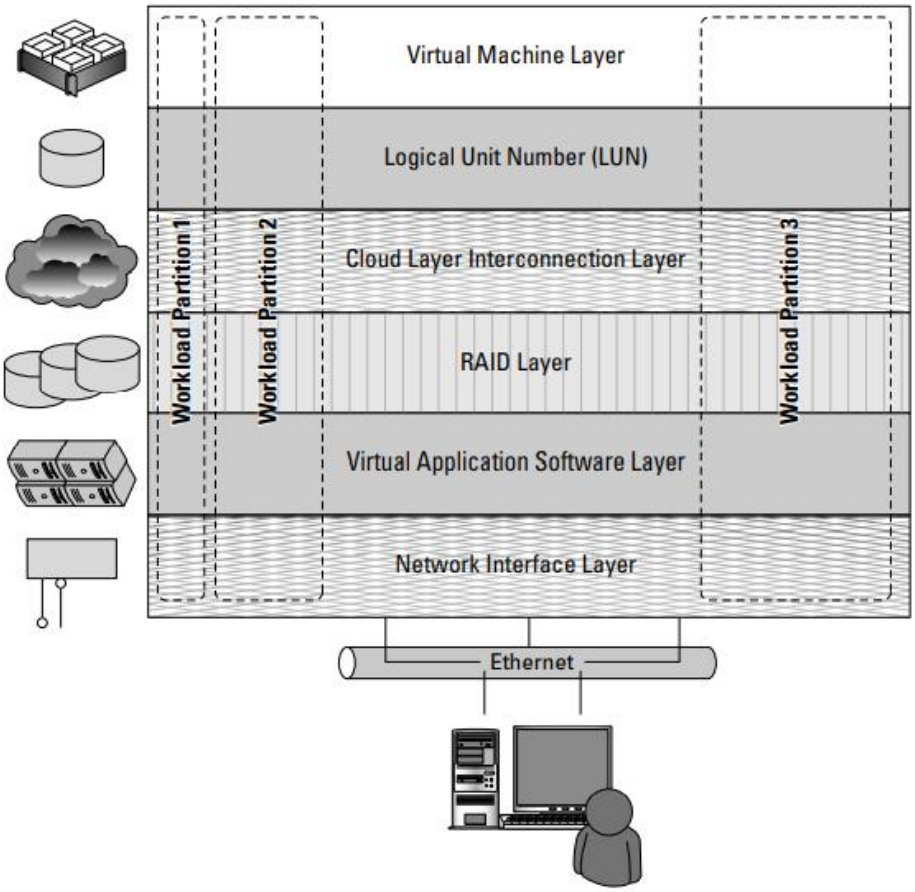
Dalšími často používanými modely jsou například IDaaS – Identity as a Service, který poskytuje autentizaci a autorizaci v distribuční síti nebo CaaS – Compliance as a Service, který zasahuje a zpracovává práva a vztahy v cloudu napříč zeměmi. [15]

1. **Infrastructure as a Service (IaaS)**

Model Infrastruktura jako služba nabízí virtualizaci hardwarových prostředků, kterými jsou například počet serverů, počet procesorů, operační paměť, datový prostor, konektivita atd. Konektivitou je pak myšleno nejen například připojení do internetové sítě, ale i hledisko bezpečnosti řešené firewallem nebo případně systémem na rozdělování šířky pásma (tzv. shaper). Všechny tyto prostředky jsou škálovatelné z hlediska zvýšení či snížení výkonu dle potřebných požadavků. Díky tomuto modelu tedy dochází k úsporám v oblasti firemní infrastruktury a lidských zdrojů potřebných na její správu. Také je snížena pravděpodobnost výpadku a nedostupnosti služeb požadovaných od koncového klienta. [16]

Vývojáři využívají IaaS model, aby vytvořili virtuální privátní servery, virtuální privátní úložiště, virtuální privátní sítě atd. Poté tyto virtuální systémy plní aplikacemi a službami, které jsou potřeba pro kompletaci daného řešení. V IaaS modelu jsou virtualizované prostředky namapované na reálné systémy. V momentě, kdy klient zahájí interakci s IaaS službou a požádá ji o zdroje z virtuálního systému, jsou tyto požadavky přesměrovány na reálné servery, které provedou požadovanou operaci. [15]

Základní jednotkou virtualizace klienta je *workload* neboli pracovní zátěž. Workload simuluje schopnost určitého typu reálného nebo fyzického serveru provést určitý objem práce. Objem práce je typicky determinován počtem transakcí za minutu (TPM – transactions per minute). Při službách jako je hosting běží klientská aplikace na dedikovaném serveru, který je uložen v serverové skříni nebo na vlastním serveru v místnosti plné takových serverů. V cloudovém přístupu je požadovaný objem výkonu nazván instancí a potřebný výkon je pak rezervován na fyzickém serveru přesně podle požadavků klienta. Obrázek níže zobrazuje tři virtuální privátní instance rozdělující si výkon v IaaS modelu a každý z nich má rozdílný požadavek na objem práce. [15]



Obrázek 10: Rozdělení virtuálního privátního serveru v IaaS modelu [15]

2. Platform as a Service (PaaS)

Platforma jako služba je model poskytující služby klientovi mimo jeho vlastní infrastrukturu. Vývoj a běh klientovo vlastních aplikací je realizován bez potřeby stahování či instalace jakéhokoliv softwaru. Klient k takové aplikaci přistupuje skrze internet či VPN. Nejčastěji nabízenými službami jsou návrh, vývoj, testování, implementace, škálovatelnost, bezpečnost, implementace a hosting aplikace, úložiště, správa verzí atp. Naopak jeho nevýhodou pak je nemožnost konverze například mezi poskytovateli takových služeb čili dochází ke ztrátě dat či celé aplikace. [17]

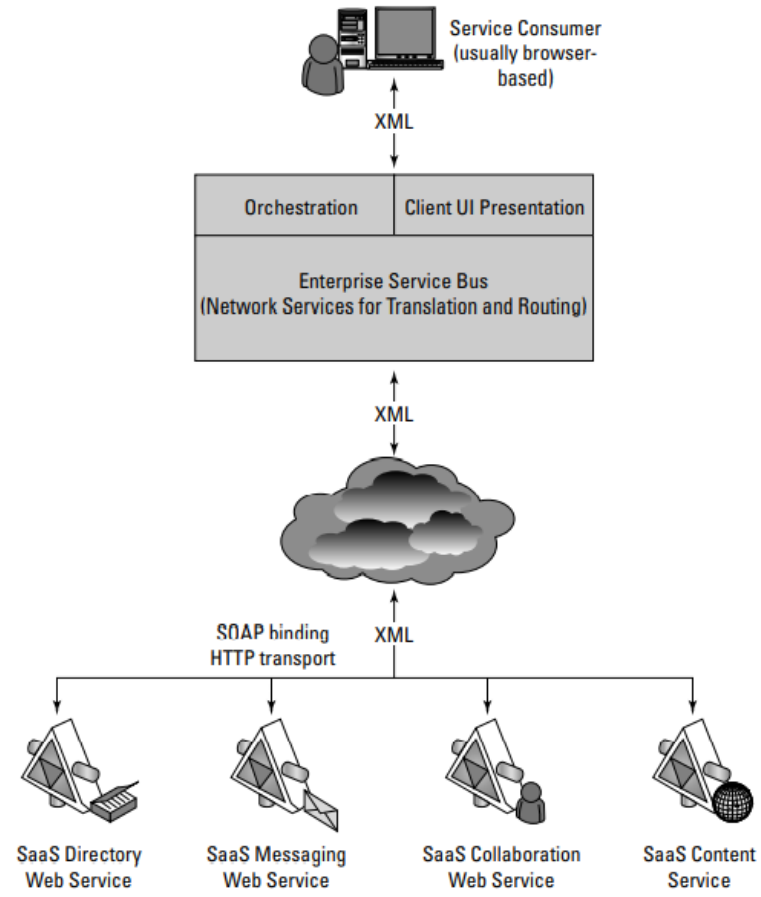
Klient nepřebírá žádnou zodpovědnost za údržbu hardwaru, softwaru nebo vývoj aplikací a zodpovídá tak pouze za interakci s danou platformou. Za operační aspekty služeb, údržbu a řízení životního cyklu produktu je zodpovědný poskytovatel IaaS. Aby taková služba byla výhodná, je plně integrovaná s potřebnými technologiemi a podporuje vývoj uživatelského rozhraní pomocí HTML, JavaScript, CSS a jiných. [15]

3. Software as a Service (SaaS)

Software jako služba je nejkompletnější model z hlediska poskytovaného hardwaru a softwaru. Stejně jako samotné softwarové řešení, tak i hardwarové potřeby jsou provozované na straně poskytovatele jako kompletní řešení dostupné zejména přes internet skrze webový prohlížeč. Kromě uživatelské interakce se softwarem je jakákoliv další interakce se systémem čistě abstraktní a ve většině případů nemožná. Neznamená to ale, že je takový software nelze přizpůsobit potřebám uživatele. V některé případech lze opravdu pouze o úpravu povrchních uživatelských parametrů, ale spoustu dalších SaaS řešení poskytuje vývojářům možnost vytvoření vlastního kompozitu aplikací skrze vystavené *API (Application Programming Interface)* neboli rozhraní pro programování aplikací. Pomocí takového API je možné například upravovat bezpečnostní model, datová schémata, charakteristiku procesů. [15] [17]

Všechny SaaS aplikace sdílí následující charakteristiky [15] :

- Software je dostupný globálně přes internet skrze webový prohlížeč
- Typické licencovaní na základě předplatného nebo na základě kapacity využívání. Splatné je typicky na opakující se bázi
- Software a služby jsou monitorované a spravované poskytovatelem nehledě na to, kde dané komponenty běží
- Cenově výhodnější díky sníženým nákladům na distribuci či údržbu
- Aplikace obsahují automatické aktualizace a záplaty. Změny jsou zaváděny v rychlejším sledu
- Mají nižší přístupovou bariéru
- Všichni uživatelé mají stejnou verzi, takže jsou navzájem kompatibilní
- Podporují přístup více uživatelů při sdílení datového modelu



Obrázek 11: Implementace SaaS s využitím architektury ESB pro integraci služeb [15]

4. Identity as a Service (IDaaS)

Identita jako služba zprostředkovává uchování informací o uživatelích, které mohou být využité například pro elektronický podpis nebo jiné elektronické transakce. Vzhledem k povaze fungování cloud computing služeb můžeme hovořit o zvýšeném riziku útoků, takže jedním ze základních principů této služby je zajištění ochrany před vstupem do sítě, či krádeží dat. Nejrozšířenějším zabezpečovacím postupem přihlašování uživatelů bývá uživatelské jméno a heslo a jedná se o tzv. single-factor authentication neboli jedno faktorová autentizace. Kvůli výše zmíněným aspektům je vhodnější zavedení two-factor authentication neboli autentizace dvou fázové. Ta spočívá v ověřování navíc pomocí náhodně vygenerovaného klíče, tzv. *token*. Dále můžeme hovořit o multi-factor authentication, kdy dochází k ověření identity pomocí nějakého biometrického údaje, například otisk prstu a podobně. [17] [15]

5. **Compliance as a Service (CaaS)**

Cloudové služby ze své podstaty zasahují na různá území s různými jurisdikcemi. Znamená to, že například zákony země původu služeb nemusí odpovídat zákonům koncových uživatelů či zemím, kde jsou požadavky procesovány. Compliance as a Service je tedy hlavně o poskytování anonymizovaných služeb a tokenů identitám, aby jim byl zajištěn přístup ke zdrojům. Přesto se ale jedná o velmi komplexní problematiku vyžadující expertízu. Compliance as a Service aplikace musí důvěryhodným poskytovatelem třetích stran, jelikož se jedná o jakéhosi prostředníka v komunikaci. Mnohem jednodušší je pak situace v privátním cloudovém řešení, kde jsou data pod správou jedné entity, stejně tak jako bezpečnost. Pokud je tato služba naimplementovaná do systému správně, může se ukázat jako velice cennou. [15] [17]

3.2.2 Modely nasazení (deployment model)

V modelech nasazení nebo nasazování je cloudový přístup rozdělen podle kritérií jako jsou oprávnění, bezpečnost, přístupnost nebo okruh uživatelů. Každý model je určen pro danou cílovou skupinu klientů. V současné době velké společnosti dávají přednost vybudování vlastních infrasktur před těmi veřejnými, což ale neplatilo v minulosti. [16] [17]

1. **Veřejný cloud (public cloud)**

Veřejný cloud je všeobecně dostupný model širé veřejnosti nebo případně pro velké skupiny například v prostředí dané společnosti. Je založen na přístupu pronájmu za využití od poskytovatele, který takovou službu spravuje ve svých datových centrech. Běžně dostupné služby v takovém cloudu jsou IaaS, SaaS a PaaS. [16] [17]

2. **Soukromý cloud (private cloud)**

Infrastruktura soukromého cloudu je spravována zejména organizací, pro níž je určena. V takovém prostředí se nacházejí pouze data daného klienta bez přístupu k datům veřejným a naopak. Běžnými službami jsou služby IaaS z důvodu distribuce pouze k jednomu klientovi, a tak většinou není třeba další modely služeb, jelikož bezpečnostní rizika jsou výrazně menší. Pro zefektivnění infrastruktury, snížení nákladů a využívání vlastností jako je škálovatelnost je toto řešení poměrně oblíbené. Součástí může také

být potřebný hardware pro monitoring interakcí napříč systémem. Pro svoji uzavřenost bývá využíván převážně velkými společnostmi. [16] [17]

3. **Komunitní cloud (community cloud)**

Komunitní cloud je takový, který je navržen tak, aby složil běžným účelům buď jedné organizace nebo několika organizacím. Ty pak sdílí účely, politiku, bezpečnost atp. Je vhodný pro skupinu podniků, mezi kterými se sdílí data. Ve většině případů je spravovaný třetí stranou, ale také komunitou samotnou. [16] [17]

4. **Virtuální privátní cloud**

Virtuální privátní cloud je unikátní tím, že má uvnitř cloudu vlastní privátní síť. Komunikace mezi klientem a službami je pak zabezpečena pomocí Virtual Private Network neboli VPN. Jelikož využívá zdroje soukromého, ale i výhody veřejného modelu, hovoříme o nich jako o hybridních cloudech. Každý z nich si ale zachovává svoje jedinečné entity, které jsou ale propojené skrze standardizované a patentované rozhraní, které zaručuje adaptabilitu cloudů navzájem. [16] [17]

3.2.3 Dostupná řešení

Současný trh je determinovaný velkým počtem poskytovatelů cloudových služeb. Výhody cloudu a přechod na cloudová řešení implementuje čím dál tím více společností. Otázkou budoucnosti zůstává, jak menší poskytovatelé dokáží držet krok s velkými společnostmi na poli cloud služeb. Otázka konkurenceschopnosti vyvstává hlavně z hlediska ceny, kvality a kapacity. Mezi aktuálně největší poskytovatele patří například Amazon Web Services, Windows Azure Services Platform, Google Web Services, IBM Smart Cloud a další. [15] [16]

1. **Amazon Web Services**

Amazon Web Service neboli AWS jsou cloudové produkty od společnosti Amazon. Ta je považována za největšího poskytovatele služeb na bázi cloud. AWS spustila společnost Amazon v roce 2006, kdy se postupně stala největším hráčem na poli poskytování IaaS služeb. [15] [16]

V dnešní době (přelom roků 2020 a 2021) je portfolio služeb Amazonu opravdu široké. Ty v současnosti nejpoužívanější popisuje následující tabulka [18] [19]:

Amazon EC2	Elastic Compute – flexibilní výpočetní kapacita poskytovaná cloudem. Správa skrze webové rozhraní, kde se jednoduše nastaví parametry výkonu nebo kapacity
Amazon RDS	Relation Database Service – uživatelsky přívětivá infrastruktura databází s implementací SQL
Amazon Simple Storage Service (S3)	Datové úložiště pro klienty. Lze pomocí webového rozhraní zcela bezpečně manipulovat s velkým objemem dat
Amazon Cloud Front	Zvyšuje rychlost při načítání webových stránek
Amazon VPC	Vytváří infrastrukturu pro privátní virtuální síť
Amazon SNS	Reaguje na události a na jejich základě spouští automaticky úlohy o čemž pak uživatele upozorní
Amazon Beanstalk	Pomáhá vývojářům řídit infrastruktury webů a automaticky aktualizuje software stejně tak jako jej škáluje
AWS Lambda	V momentě, kdy je server zaplaven přílivem požadavků, hrozí jeho zhroucení. AWS Lambda řeší tuto problematiku a zvládne jakoukoliv zátěž požadavků
AWS Autoscaling	Poskytuje automatickou škálovatelnost pro celé flotily serverů
AWS IAM	AWS Identity and Access Management – opevnění pro citlivá data a jejich zabezpečení

Tabulka 7: Nejpoužívanější Amazon cloud produkty 2020

2. Windows Azure Services Platform

Společnost Microsoft můžeme znát jako jednu z největších společností na trhu s operačními systémy. Jakožto takový gigant nemůže chybět ani na poli se službami modelu cloudu. Windows Azure Services Platform je platforma, která poskytuje svým klientům síťovou a serverovou infrastrukturu a napojenou na data centra společnosti. Platforma sestává z několika komponent, jsou jimi například [15] [16] [20]:

- **Windows Azure –** operační systém založený na bázi cloudu. Svým uživatelům nabízí kompletní prostředí na vývoj a správu produktů řady

Azure. Jedná se tedy o plně přizpůsobitelné a škálovatelné prostředí. Díky *Windows Azure Compute* platforma podporuje více druhů aplikací, primárně však prostředí .NET, ale také Java, C++ nebo PHP

- **SQL Azure –** tato komponenta umožňuje zpracovávání a ukládání relačních dat v prostředí Windows Azure. Je déle složen ze 3 částí, a to:
    - *SQL Azure* – Databázový systém v cloudu. Umožní aplikacím v lokální síti, ale i v cloudu ukládat relačně orientovaná data na servery datových center. Náklady se škálují přesně podle potřeb zákazníka, tzn. Zákazník zaplatí jen za to, co reálně využívá
    - *SQL Azure Data Sync* – slouží pro synchronizaci dat mezi databází SQL Azure a databázemi SQL v síti
    - *SQL Azure Reporting* – umožňuje databázi SQL Azure generování a publikování standardních sestav služby SSRS (SQL Server Reporting Services)
- **Windows Azure AppFabric –** AppFabric zastává obchodní logiku, která je závislá na fungování a infrastruktuře, kterou využívá produkt od svého nasazení. Poskytuje tvůrcům aplikací specifické druhy infrastruktur. Sestává z komponent jako je servisní sběrnice (Service Bus), řízení přístupu (Access Control) a vyrovnávací paměť (Caching).
- **Windows Azure Marketplace –** jedná se o online obchod pro data a aplikace v prostředí cloudu

3. Google Web Services

Google je prototypem cloud computing společnosti. Jádrem obchodních aktivit společnosti Google je její vlastní automatizovaná technologie, pomocí jíž indexuje prostředí webu. Nejdůležitější částí reklamní části obchodních aktivit jsou pak AdWords a AdSense služby pomocí kterých může zákazník efektivně cílit obsah. Aplikace Googlu jsou na bázi cloudu a nabízí širokou škálu účelů jako například produktivita a kolaborace, mobilní aplikace, úložiště, sociální interakce, média a zábava, umělá inteligence a spoustu dalších. Google věnoval přes 12 let vývoji zdrojů pro **Kuberenetes** a pak z něj učinil kompletně open-source řešení. Ve zkratce se jedná o systém nasazování, škálování a řízení kontejnerizovaných aplikací. Kubernetu je věnována i jedna z částí této práce. [15] [21]

3.2.4 Výhody a rizika Cloud řešení

Jako každé technické řešení, i cloudová řešení mají své limitace, své výhody a nevýhody. Některé jsou zjevné již ze samotné funkční podstaty, jiné vyžadují zvážení a hlubší znalost problematiky. Dalším důvodem pro využití cloudových řešení může být i problematika legislativy, kterou nutně musí řešit nadnárodní společnosti. Omezuje například v jaké geografické lokaci mohou či nemohu být ta či ona data uložena atp. [16] [22]

1. **Výhody Cloud řešení**
    - **Moderní a flexibilní infrastruktura** – Dnešní doba vyžaduje po společnostech IT strukturu držící krok s rychle se vyvíjejícími technologiemi. Zde vznikají první překážky, a to sice vysoké náklady na pořizování aktuálních prostředí a náročná správa nových technologií. S přechodem na cloud řešení se tyto náklady redukují, jelikož klesne požadavek na vlastní hardware. Tím se redukují i náklady na personál potřebný k údržbě takové infrastruktury. Flexibilita, která již byla zmíněna dříve, je také důležitým benefitem. Je možné kdykoliv navýšit parametry infrastruktury bez nutnosti větších nákladů.
    - **Dostupnost a spolehlivost systémů** – Spolehlivé systémy, které eliminují riziko ztráty dat například při výpadku systému jsou nákladné a pokud by společnosti měly držet krok s dobou, musely by neustále sahat po nejnovějších technologiích. Opět se projevuje faktor ceny za služby. Poskytovatelé většinou nabízí komplexní portfolio služeb, které neustále aktualizují a několikanásobně zálohované.
    - **Přístupnost řešení**_– Společně s pokrokem technologií a dostupností internetového připojení roste požadavek na přístupnost. Proto se některé firemní procesy začaly přesouvat mimo prostředí společnosti. Postupem času začala mobilní zařízení zvládat mnohem náročnější úkon, a tak se na jejich integraci do infrastruktur společností začal klást mnohem větší důraz. Samozřejmě i tento přístup skýtá určitá rizika, a tak je nutné klást důraz na vysokou míru bezpečnosti. [16] [22]

2. Rizika Cloud řešení

- **Stabilita připojení** – Hlavním předpokladem pro cloudové služby je stabilní a rychlé internetové připojení. Výpadek nebo snížení kapacity připojení či přenosové kapacity by mohl způsobit velké problémy při nasazování. Je důležité zvážit všechny rizika, pokud společnost zvažuje přechod na model cloudových služeb, protože výpadky či nestabilita připojení může být opravdový problém.
- **Výběr poskytovatele služeb** – Jelikož je celá společnost a infrastruktura závislá právě na poskytovateli, je třeba provést důkladnou studii trhu a analýzu potřeb, ale i možností poskytovatele. Výběr nesprávného poskytovatele může způsobit společnosti problémy například pokud poskytovatel není schopen dodat služby v potřebné kvalitě. Je nezbytné zvážit výběr poskytovatele, jelikož i z důvodu, že přechod k jinému poskytovateli může být vysoce náročný a v některých případech i nemožný.
- **Bezpečnost informací** – Je zcela zřejmé, že pokud společnost využívá vlastní infrastrukturu, je riziko ztráty informací nízké, respektive nižší než v případě, kdy jsou informace přístupné i z vnějšku. Právě při přechodu na cloudové služby ztrácí společnost kontrolu nad kompletním systémem, a tak je vhodné ukotvit ve smlouvě s poskytovatelem patřičné kroky a podmínky, které zamezí nebo minimalizují rizika spojená s přesuny a ukládáním informací. [16] [22]

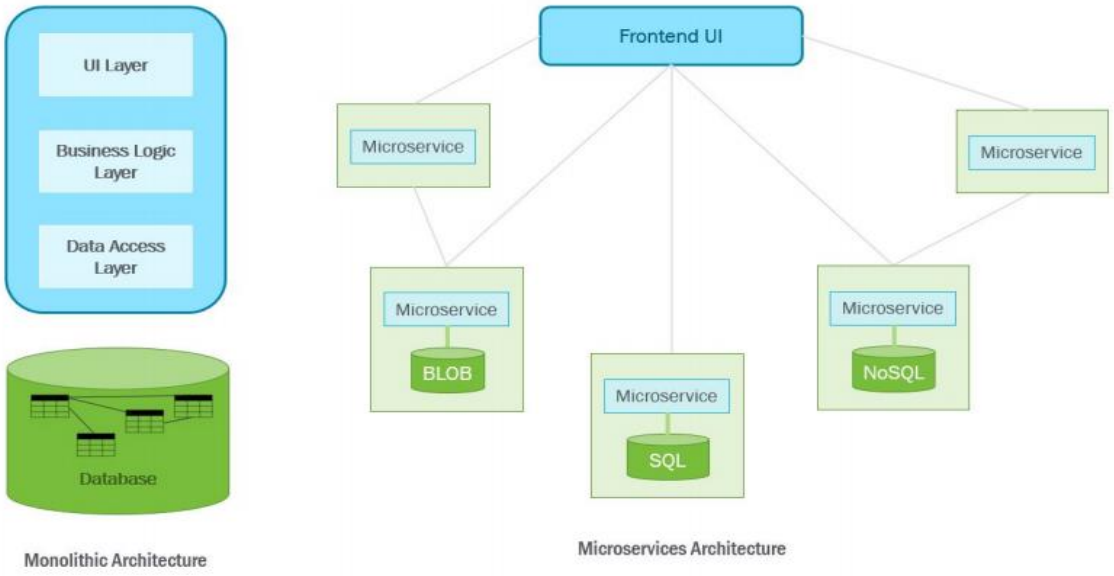
3.3 Cloud-Native

Cloud-Native aplikace jsou designovány tak, aby běžely na základě cloud computing služeb a infrastruktur. Nejedná se pouze o prosté nástupce cloud computing aplikací, nýbrž o aplikace, které se plně zaměřují na celý životní cyklus produktu. Jejich záběr čítá architekturu aplikace, nasazování aplikace, škálovatelnost aplikace, doručování nových verzí, ale také monitoring a statistiky užívání. Součástí a smyslem Cloud-Native aplikací je přechod od velkých neforemných a monolitických aplikací k aplikacím složeným z menších mikroslužeb, ale zároveň tento přístup umožnuje správu a řízení velkých a komplexních aplikací přizpůsobených prostředí cloudu. Pokud by se však vzala monolitická aplikace a pouze by se virtualizovala v podobě jednoho virtuálního serveru, nebyla by odolná například vůči výpadkům či nárokům na škálovatelnost. Tím pádem by plně nevyužívala prostředí a možnosti cloudového prostředí. Je to právě Cloud-Native přístup, který se snaží tyto nevýhody a rizika eliminovat. [23]

Definice Cloud-Native aplikace dle Kratzkeho a Quinta [3] :

„Cloud-native aplikace je distribuovaný, pružný a horizontálně škálovatelný systém složený z (mikro)služeb, který izoluje stav v minimu stavových komponent. Aplikace a každá soběstačná nasazovací jednotka aplikace je navržena podle cloud-orientovaného návrhového vzoru a provozována na samoobslužné pružné platformě.“

Jak můžeme vidět na obrázku 4, v přístupu mikroslužeb jsou jednotlivé komponenty oddělené. To dovoluje například, aby každá komponenta byla vytvořená v jiném jazyce a aby byl použitý nejvhodnější technologický postup pro daný účel. Mikroslužby pak mezi sebou komunikují pomocí protokolu jako je například http a zprávy a data jsou zasílána ve formátu jako je JSON nebo XML, což dává vývojářům spoustu volného prostoru. [24]



Obrázek 12: Porovnání monolitické architektury a architektury mikroslužeb [24]

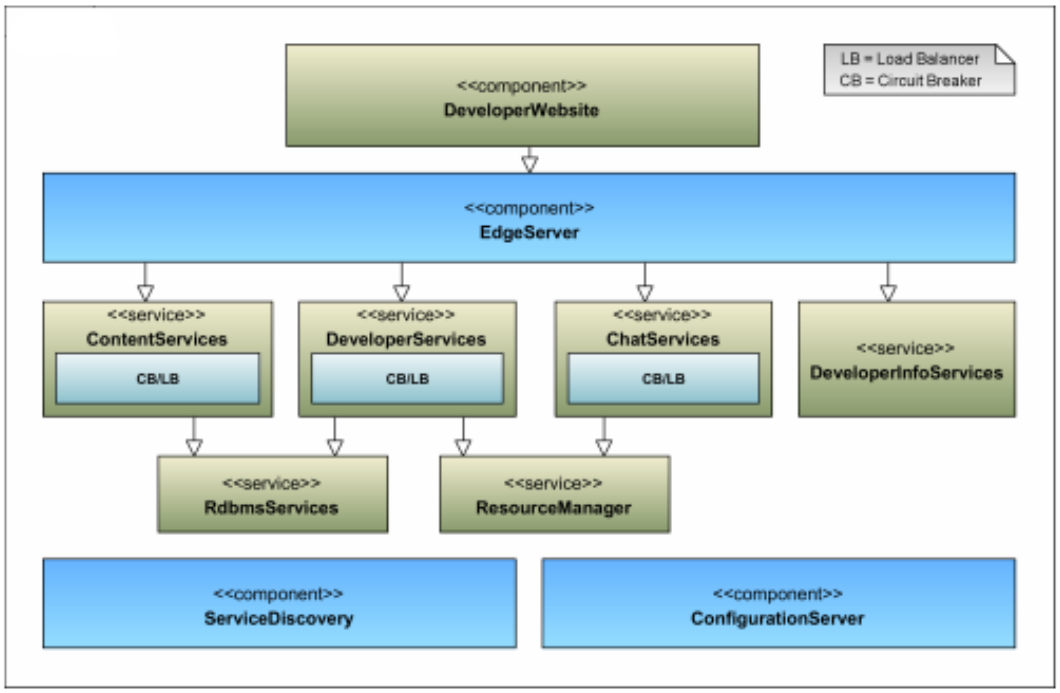
3.3.1 Microservices (mikroslužby)

Microservices neboli mikroslužby jsou architektonickým stylem, který má za úkol doručovat softwarové systémy jako hotové balíčky rozdělené na malé služby. Tyto malé služby je pak možné nasazovat každou zvlášť na jinou platformu, přičemž každá běží ve svém vlastním procesu. Komunikace pak probíhá například pomocí mechanismů jako je REST API (Representational State Transfer). [12]

Systém je implementovaný podle architektury mikroslužeb, pokud je zkomponován z několika služeb bez přítomnosti centralizovaného řídícího prvku. Dalším důležitým charakteristickým rysem je odolnost vůči selhání, jelikož každý požadavek poslaný do systému znamená několik volání napříč službami a tím pádem je potřeba zajistit, aby všechny služby byly přítomné a v dobré kondici. Aby bylo dosaženo plně funkčního systému dle architektury mikroslužeb a aby bylo možné plně využívat všech benefitů které nabízejí, je potřeba využívat a udržovat několik komponent. Většina z těchto komponent řeší složitosti distribucí byznys logiky skrze všechny integrované služby. [2]

Výše zmíněnými komponentami jsou [2]:

- **Konfigurační server (Configuration Server)** – Jedním z principů Continous Delivery (CD – je mu věnována jedna z dalších kapitol této práce) je oddělení zdrojového kódu od jeho konfigurace. Toto pojetí umožňuje měnit konfiguraci aplikace bez nutnosti ji znovu nasazovat. Architektura mikroslužeb má mnoho jednotlivých služeb a jejich přenasazování by bylo jistě velmi nákladné, je mnohem výhodnější držet konfiguraci oddělenou a nechat služby pouze stahovat konfigurační data.
- **Propagace služeb (Service Discovery)** – V návrhu mikroslužeb se může vyskytovat dokonce několik instancí stejné služby o čemž vypovídá jejich škálovatelnost. V takovém případě je téměř nemožné držet jejich přesné statické adresy. Ty se totiž službám mění podle toho jak se škálují, vypínají a zapínají či selhávají. Propagace adres služeb pak funguje buď na straně klienta nebo na straně serveru. Tak či tak, při nastartování instance služby se její adresa spolu s portem propagují a při jejím vypnutí se zase odstraní či přepíše.
- **Vyvažovač zátěže (Load Balancer)** – V závislosti na škálovatelnosti by měl aplikace být schopna distribuovat zátěž požadavků na služby na všechny její instance. Přesně to má na starost Load Balancer a jedná tak v závislosti na informacích, které obdrží od komponenty Propagace služeb.
- **Jistič (Circuit Breaker)** – Implementací jističe se zabrání tomu, že selže celý systém v závislosti na selhání jedné nebo několika služeb. Taková tolerance chyb by měla být zabudována v každém Cloud-Native návrhu, jelikož služby spolu úzce spolupracují.
- **Okrajový server (Edge Server)** – Veškerá komunikace zvenčí je směrována skrze tento server. Jeho úkolem je chránit před nechtěným vystavením API širému světu a také odklánět komunikaci směrem ke službám. V tomto případě pak klienti nejsou ovlivnění, pokud dojde ke změnám struktury systémových služeb.



Obrázek 13: Vyobrazení architektury s výše zmíněnými komponentami [2]

I přes to, že mikroslužby řeší mnoho problémů a vytváří potenciál pro nové příležitosti, přináší s sebou také i rizika spojené s novými komplexitami. Například při návrhu designu architektury založené na mikroslužbách jsou zapotřebí specializované a odborné znalosti. Navíc vyvstává úskalí komplexity při integraci služeb, jejich testování a nasazování. Také monitorování a podpora mikroslužeb za běhu je značně náročnější než v případě monolitické aplikace s jednoduchými procesy. [25]

3.3.2 Proces přechodu na architekturu mikroslužeb

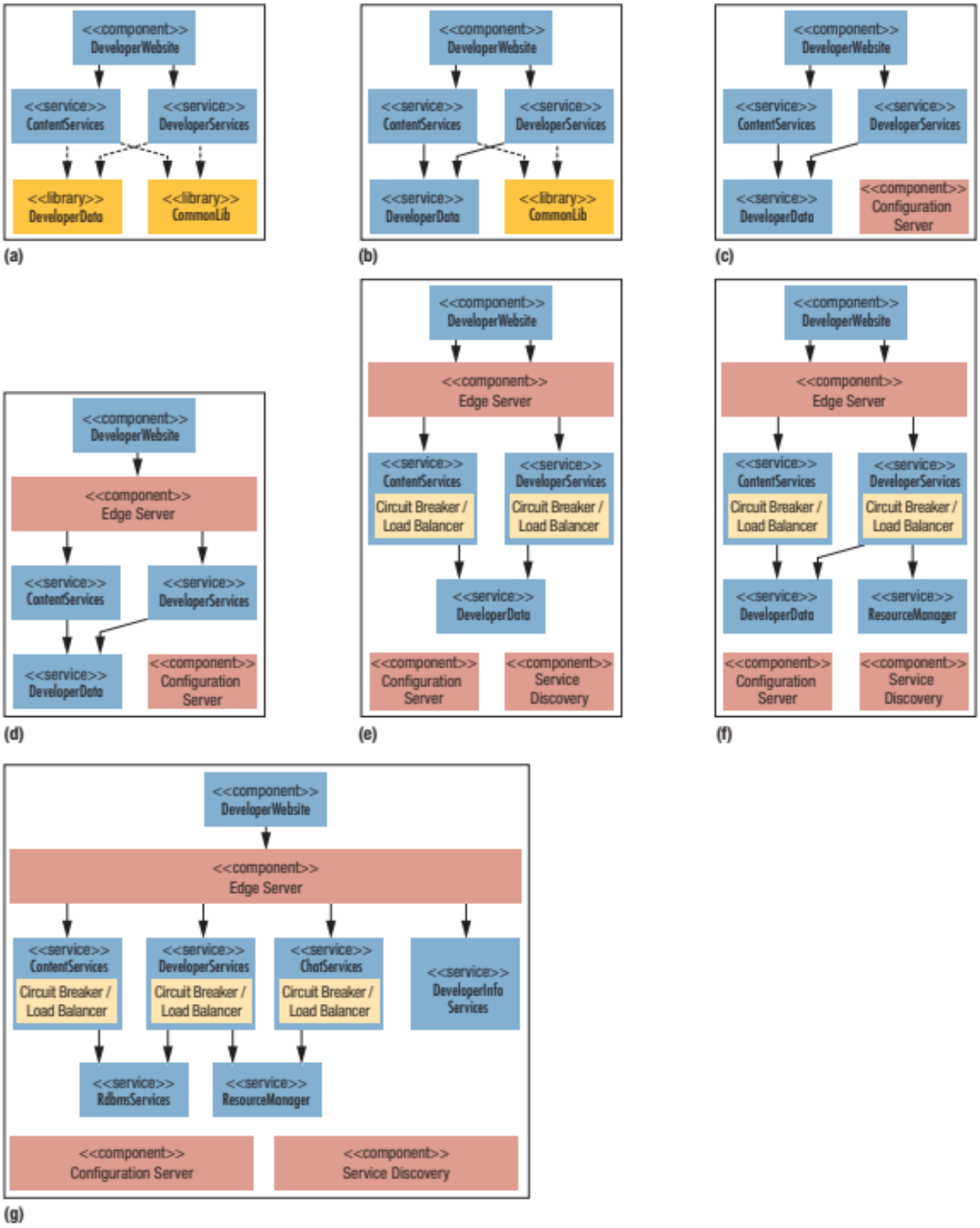
Proces přechodu z monolitické architektury na architekturu mikroslužeb s sebou přináší mnoho výhod, jak už zde bylo zmíněno. V zásadě nám poskytuje přizpůsobivost vůči technickým změnám a tím se může předejít technologickému uzamčení. Dále také umožňuje zlepšit a zefektivnit strukturu vývojového týmu a tím pádem snížit čas na uvedení na trh (angl. Time-to-market). Abychom těchto výhod mohli využít a zároveň abychom snížili rizika zmíněná v předchozí kapitole, je vyžadováno co nejvíce procesů automatizovat. Toho může být dosaženo například aplikováním infrastruktur automatizace softwarů jako je **průběžná integrace (CI – Continuous Integration), postupné dodávání (CD –**

Continuous Delivery) nebo **programování řízené testy (TDD – Test Driven Development).** Tématu CI/CD je v této práci věnována kapitola dále. [12] [25]

Modelový proces přechodu implementovaný ve společnosti *PegahTech* (*www.pegahtech.ir*) zobrazuje následující *obrázek 6*. Systém, nazvaný *Backtory*, poskytuje back-endové služby vývojářům mobilních technologií, kteří neznají a ani nemusí znát žádné serverové technologie. Původně se jednalo o systém správy relační databáze, který fungoval jako jednoduchá služba. Vývojáři definovali databázová schémata a systém pak poskytoval softwarový vývojářský balíček pro požadované platformy (například Android nebo iOS). [25]

Celý proces zobrazuje schéma jednotlivých kroků na obrázku 6. Plné šipky a čáry zobrazují volání služeb, pruhované indikují závislosti knihoven. Kroky v procesu jsou [25]:

a) Architektura systému *Backtory* před migrací
b) Transformace *DeveloperData* na službu
c) Implementace konfiguračního serveru (Configuration Server)
d) Implementace okrajového serveru (Edge Server)
e) Implementace jističe, vyvažovače zátěže a dynamické kolaborace služeb
f) Implementace *ResourceManager* (Správce zdrojů)
g) Architektura systému *Backtory* po migraci

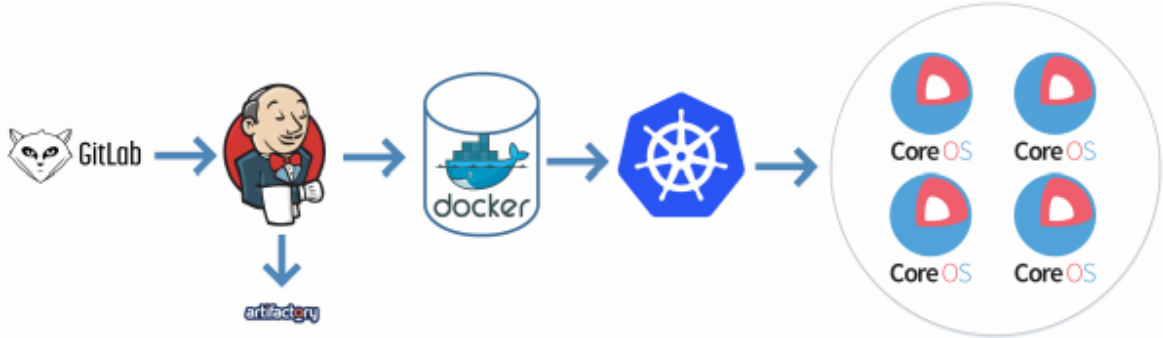


Obrázek 14: Proces přechodu systému Backtory na mikroslužby [25]

3.3.3 **Kontejnerizace a shlukování**

Proces kontejnerizace zahrnuje zapouzdření či zabalení kódu softwaru a všech jeho závislostí tak, aby byl schopen běžet jednotně a konzistentně na jakékoliv infrastruktuře. S tím, jak rychle se tato technologie vyvíjí a dospívá můžeme pozorovat měřitelné výhody, které přináší vývojářům a týmům stejně tak, jako celé infrastruktuře. Kontejnerizace umožňuje vývojářům vytvářet a nasazovat aplikace mnohem bezpečněji a rychleji. Při klasickém přístupu je kód vyvíjen v určitých podmínkách a v určitém prostředí a při pokusu o přesun do prostředí nového často dochází k chybám a selhávání. Nejčastější případy nastávají při pokusu o přesun ze stolního počítače do prostředí virtuálního stroje *(VM – Virtual machine)* nebo z operačního systému Linux do prostředí operačního systému Windows. Kontejnerizace redukuje a eliminuje tyto problémy pomocí zabalování kódu s konfiguračními soubory, knihovnami a závislostmi nezbytnými pro běh aplikace. Každý takový balíček neboli **kontejner** je oddělen od hostitelského operačního systému, a tudíž se z něj stává přenositelná jednotka schopná běžet napříč platformami. [26]

Pokud bychom dále srovnávali virtualizaci a tzv. kontejnerizaci, hlavní výhoda kontejnerů jsou nízké režijní náklady. Aby byla tato vlastnost kontejnerů využita co nejvíce, jsou kontejnery implementovány společně s nenáročnými operačními systémy, které mají minimum částí, které jsou ale potřebné pro běh kontejnerů. Takovým operačním systémem je například RedHat OpenShift *(pozn. autora: ve zdroji* [2] *je uveden CoreOS, tomu ale v roce 2020 skončila podpora a stal se tak plně součástí RedHat OpenShift* [27]*).* Tento operační systém je dále jednoduše integrovatelný s nástrojem Google **Kubernetes**, který slouží k orchestraci pro jednoduché nasazování kontejnerů do shluků, tzv. **clusterů.** Pomocí nástroje Kubernetes mohou být načteny ze soukromého úložiště a nasazeny do příslušných clusterů. Pro představu – určitá služba může být nasazena pokaždé se třemi běžícími instancemi. Dále je možné nastavit shluk několika instancí RedHat OpenShift s Kubernetes agenty, kteří zajistí jejich orchestraci. Dále se na tento shluk operačních systémů nasadí požadované služby namísto nasazení na jeden server. Následující *obrázek 7* ukazuje finální posloupnost instrukcí pro prvky sdružené do tzv. **Pipeline** (čes.: potrubí/roura), které se využívá v procesech postupné integrace/postupného doručování – CI/CD (podrobněji popsáno v kapitole 3.4 této práce). Modelová posloupnost je následující [2]:



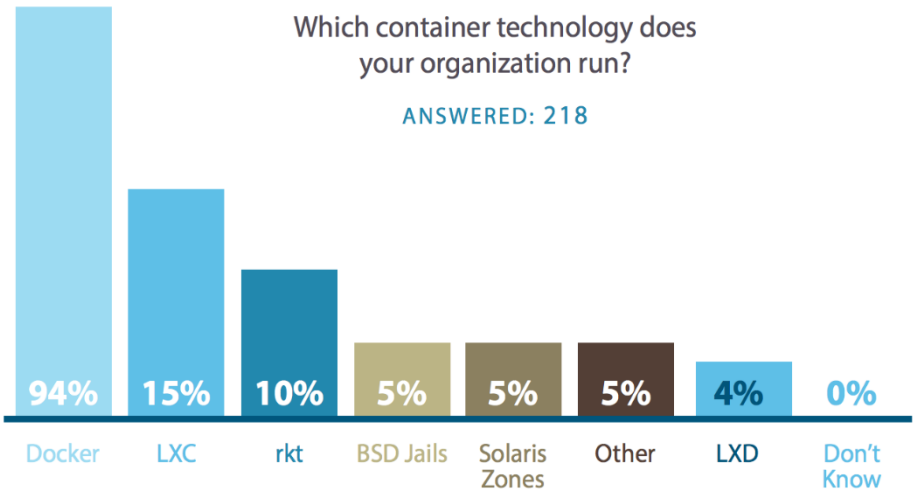
Obrázek 15: Modelová doručovací pipeline [2]

GitLab (Git úložiště zdrojového kódu, DevOps platforma [28]) → **Jenkins** (automatizační nástroj pro nasazování, kompilaci, testování atd. [29]) → **Docker** (podrobněji dále v této práci) → **Kubernetes** (podrobněji dále v této práci) → **Cluster** (shluk operačních systémů)

Docker

Docker je open-source software, který poskytuje jednotné prostředí pro izolaci aplikací do kontejnerů napříč platformami. Projekt Docker byl vytvořen v roce 2013 a je vytvořen v programovacím jazyce Golang. Docker jako takový je složen z několika technologií současně. Patří mezi ně specifikace kontejnerů, runtime *(neboli běhové prostředí součástí kterého jsou tzv. Dockerfiles – konfigurační soubory umožňující opakované vytvoření kontejnerů)* nebo DockerHub *(slouží jako úložiště kontejnerů)*. [22] [30]

Docker je zároveň nejpoužívanější kontejnerizační technologií *(obrázek 8)*. Dalšími technologie jsou například *LXC, rkt, BSD Jails, Solaris Zones*, a další. [31]



Obrázek 16: Nejpoužívanější kontejnerizační technologie [31]

Dále rozlišujeme tzv. Docker image neboli obraz Dockeru. Docker image je označení pro obraz dané aplikace obsahující veškeré spustitelné soubory, závislosti, knihovny a konfigurační soubory. Takové obrazy se skládají z několika vrstev, kde každá z vrstev obsahuje příslušná meta data a informace o změně oproti vrstvě předchozí. Vrstvy mají tedy určitou hierarchii, kde nejspodnější výchozí vrstvou je tzv. base image (neboli základní obraz), ze kterého vychází vrstvy další. Každá další vrstva navazuje na svého předka přidáním různých dalších implementací, například serveru Apache, Git verzovacího systému apod. Base vrstva žádného předka nemá a může jí být například jednoduchá Linuxová distribuce jako je Debian, RedHat, CentOS a další. Tato vlastnost umožňuje doručovat docker obrazy jako soubory modifikací nad určitým base obrazem. [32] [22]

V základu existují 2 přístupy vytvoření takového docker obrazu – manuální a automatický. V manuálním případě se spustí existující image a v něm jsou pak provedené určité změny. Tento proces je ale neflexibilní a je nutné jeho kroky provádět znovu a znovu při vytváření dalších obrazů. K automatizovanému přístupu se využívají Dockerfiles (neboli konfigurační soubory dockeru). Podle definice v dockerfile dojde k vytvoření kontejneru se všemi potřebnými závislostmi a soubory pokaždé úplně stejně. [32]

Docker je spuštěný jako tzv. **daemon** *(neboli démon – v informatice označení programu, který je spuštěn dlouhodobě a není v přímém kontaktu s uživatelem a často není ani závislý na tom, zda je uživatel přihlášen. Jeho úkolem je vyčkávat na nějakou událost a tu posléze odbavit* [33] *).* Takový program při přijetí instrukce spouští kontejnery, kontroluje izolace mezi kontejnery a ověřuje, že kontejnery využívají pouze ty dané a přidělené zdroje. Démon také sleduje stavy kontejnerů a vykonává například jejich restart, pokud je to potřeba. Dalším úkolem démona je i správa docker obrazů a kontejnerů, respektive jejich stažení případně nahrání do příslušného vzdáleného úložiště zvaného **Docker registry** *(Docker registr je bezstavová aplikace, která je škálovatelná a běžící na straně serveru. Umožňuje distribuci docker obrazů.* [34]*).* Příkladem Docker registru může být *DockerHub* již dříve zmíněný. Pomocí jmenných prostorů dockeru, zvaných **Docker Namespaces** pak Docker řídí izolaci jednotlivých kontejnerů a procesů mezi sebou. [30] [22]

3.3.4 **Orchestrátory**

Díky designu architektury mikroslužeb a technologii kontejnerů rozdělíme aplikaci na jednotlivé menší části, které zabalíme do jednotlivých kontejnerů společně s jejich závislostmi a knihovnami. Jak už bylo zmíněno, kontejnery nám umožní spustit aplikaci či její část v různých prostředích a systémech vždy se stejným výsledkem bez nutnosti jakékoliv konfigurace. Samozřejmě, že v případě jedné či dvou aplikací spravované v například 10 kontejnerech, může se tento přístup znát poněkud složitý a dá se celý proces zvládnout manuálně. V případě, kdy je třeba spravovat řádově stovky služeb v tisícovkách kontejnerů, stává se správa kontejnerů manuálně nemožným úkolem. Přesně z těchto důvodů vznikly tzv. Orchestrátory, které mají na starosti úkoly jako [22] [35]:

- Stažení a spouštění požadovaného docker obrazu na vhodném určeném serveru, přičemž snahou orchestrátoru je rovnoměrné rozložení zátěže napříč dostupnými servery a přiřadit kontejnerům příslušné zdroje.
- Správné ukončování aplikace tak, aby nedošlo ke ztrátě ani poškození dat. Transakce jsou při tom potvrzeny a využité zdroje se uvolní.
- Sledování stavu běžících kontejnerů a jejich správa. Sleduje, zdali kontejnery fungují, jak mají a pokud například dojde uvnitř nějakého kontejneru k chybě, orchestrátor by jej měl ukončit a na jeho místo vytvořit nový dle stejných a parametrů.
- Škálování aplikace na požádání. Orchestrátor na požádání vytvoří počet požadovaných instancí aplikací. Toho se využije v momentě, kdy je při špičce aplikace zatížena, dojde k navýšení počtu kontejnerů dané komponenty.
- Load balancing neboli vyrovnávání zátěže, kterým rozloží zátěž mezi všechny spuštěné instance aplikace. Stejně tak, pokud dojde k poškození jedné z instancí, orchestrátor na ní přestane posílat požadavky a rozdělí je mezi zbylé, než dojde k nápravě chybového stavu či restartu kontejneru.
- Poskytování přístupu z vnějšího světa. Uživatelům je vystaven port veřejné adresy, na který uživatelé přistupují a mohou tak danou službu uvnitř kontejneru využívat.

- Monitoring aplikací. Orchestrátory poskytují statistiky například o využití jednotlivých kontejnerů či umožňují procházet logy služeb, což je užitečná vlastnost při řešení a hledání příčin chybových stavů aplikace či služby.

Orchestrátory jsou mocným pomocníkem zejména v případě větších a dynamičtějších aplikací, kde by údržba a správa byla velice náročná z hlediska lidských i finančních zdrojů. Další užitečnou vlastností je možnost automatizace procesů nasazování aplikací nebo nasazování novějších/starších verzí aplikací. Pokud nastane případ, že se chyba dané aplikace projeví až v produkčním stádiu u koncového uživatele, může se jednoduše nasadit starší verze neobsahující chybu (jen v určitých případech) a po opravě chyby znovu nasadit verzi novější. Není tak zapotřebí velké množství inženýrů na řešení a správu kontejnerů, ale pouze menší tým pracovníků, kteří obsluhují a konfigurují orchestrátor. [22]

Zástupci takových softwarů jsou například produkty Docker Swarm, produkt společnosti Docker, který má skvělou integraci s nástrojem Docker, dále pak Mesos Marathon, který poskytuje vrstvu abstrakce nad fyzickými servery v datovém a centru, anebo Kubernetes, kterému je věnována další část této práce. [36] [37]

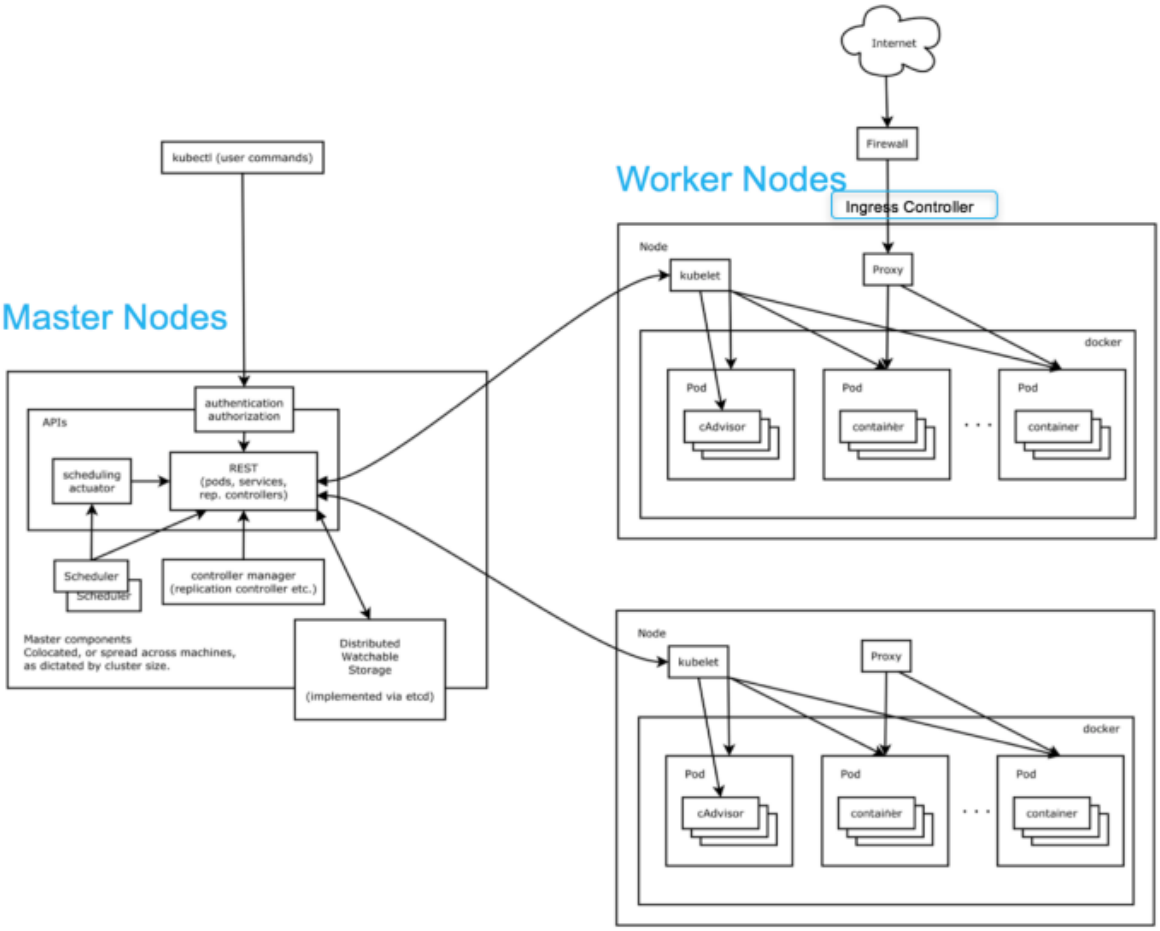
Kubernetes

„Kubernetes (občas zkráceně K8s s 8 udávajícím počet písmen mezi “K“ a “s“) je open source systém pro nasazování, škálování a správu kontejnerizovaných aplikací kdekoliv.“ [38]

Jak již bylo v této práci zmíněno, Kubernetes byl původně vyvíjen společností Google pod hlavičkou Google Cloud, ale v roce 2014 jej společnost vypustila do světa jako open source aplikaci. Jedná se o kontejnerově orientovaný software určený ke správě, který se stal de facto standardním nástrojem pro nasazování a operace spojené s kontejnery. Kubernetes automatizuje operační úlohy spojené se správou kontejnerů pomocí zabudovaných příkazů určených pro nasazení, zavádění změn aplikací, škálování aplikací směrem nahoru a dolu tak, aby vyhovovaly potřebám, monitorování aplikací a zjednodušení správy aplikací. [37] [38]

Pojmy Kubernetes a Docker jsou občas veřejností zaměňované, ačkoliv se jedná o úplně jiné nástroje s rozdílným účelem. Rozdíl je ve zkratce ten, že Docker se stará o to, aby všechno spojené s aplikací a jejím během uzavřel do kontejnerů a aby byl obsah dostupný ve správný moment. Dá se tedy vyříknout tvrzení, že Docker se stará o kontejnery zevnitř, kdežto Kubernetes zvenčí, jelikož pak s danými kontejnery manipuluje. Kubernetes může být použit samostatně bez Dockeru s jiným nástrojem pro vnitřní správu kontejnerů. Otázka tedy není „jaký je rozdíl mezi Kubernetem a Dockerem?“, ale „Jak používat Kubernetes s Dockerem?“. Každý nástroj má svou specifickou roli v úloze kontejnerizace a správa aplikací. [38]

Architektura Kubernetu je složena ze dvou základních částí – *Master Node (hlavní uzel)* a *Worker Nodes (pracovní uzly)*. Tyto uzly jsou dále členěny dalšími komponentami a jednotlivé uzly pak mohou být nasazeny jak na fyzických, tak i virtuálních serverech. Schéma architektury ukazuje následující obrázek [22]:



Obrázek 17: Architektura orchestrátoru K8s [22]

Kontrolní plán je vytvořen hlavním serverem (Master server) a ten je pak zodpovědný za správu a obsluhu událostí a **podů** (*neboli „lusků“ – Pod je ta nejmenší nasaditelná jednotka, kterou lze v Kubernetu vytvořit a spravovat. Je to skupina jednoho či více kontejnerů se sdíleným úložištěm, síťovými zdroji a specifikací, jak řídit kontejnery.* [39]). Hlavní sever tvoří také další komponenty jako API *(application programming interface)* server, scheduler *(plánovač)* a controller *(kontrolér)*. API server definuje a otevírá REST API, pomocí kterého může probíhat komunikace s ostatními pody a s Kubernetem. Komponenta controller sleduje stav clusteru skrze API server a zajišťuje požadovaný stav clusteru. Controller je dále rozdělen na *pod controller, service controller* nebo *replication controller*, který kontroluje počet podů a zapíná/vypíná pody podle konfigurace na požadovaný počet. Komponenta scheduler je zodpovědná za spouštění podů na pracovních uzlech.

Další komponentou je Node server. Ten má za úkol obstarávat běh jednotlivých podů a je řízený master komponentou. Získává z ní potřebné informace a zpět poskytuje informace o stavech a zátěži. Node server je dále složen z komponent *kubelet* a *kube proxy*. Kube proxy komponenta vykonává jednoduché směrování na bázi TCP/UDP a je přítomná na každém podu. Kubelet pak zařizuje komunikaci s komponentou Master a zajišťuje řízení a správu podů. Z definice podu v předchozím odstavci vyplývá, že kontejnery v podu mají stejnou IP adresu, sdílejí síťové porty a komunikují přes lokální adresu *localhost*. Různé kontejnery v podu pak musí používat jiné porty a přistupují ke sdílenému úložišti.

Aby se jednotlivé pody oddělily, využívá Kubernetes Namespaces, podobně jako Docker. Jednotlivé zdroje uvnitř svých Namespaces jsou izolované a komunikovat mezi s sebou mohou pouze přes veřejné rozhraní. Kubernetes nabízí opravdu mnoho možností, jak spravovat dané aplikace a vzhledem ke své modulární architektuře mohou být jeho jednotlivé komponenty nahrazeny anebo rozšířeny jakýmkoliv vlastním řešením. Například pokud budeme požadovat jiný algoritmus, než nabízí *scheduler*, je možné nahradit jej libovolnou službou, která splní naše požadavky. [22] [37]

3.4 CI/CD

CI/CD je často užívanou zkratkou pro automatizované procesy v rámci agilního vývoje systémů v prostředí cloudu. Jedná se o kombinaci 2 metod **CI – Continuous Integration (průběžná integrace)** a **CD – Continuous Delivery/Deployment (průběžné doručování/nasazování)**. Tyto 2 metody a přístupy ve spojení umožňují vývojovým týmům častěji a spolehlivěji zveřejňovat nové verze vyvíjeného softwaru právě díky automatizaci dílčích částí. Umožňují také lépe integrovat práci s ostatními týmy a mít přehled o výsledku a změnách. Hlavním výsledkem takové spolupráce je rychlost a kvalita celého vývoje softwaru. [40] [41]

3.4.1 CI

Jelikož se na vývoji softwaru může podílet několik jednotlivců nebo týmů či rozsáhlý vývojový tým, který může paralelně vyvíjet a pracovat na různých komponentách systémů, je nutné tyto komponenty pak spojovat do funkčního celku. Principy pro integraci kódu, předcházení chybám a plynulosti vývoje, které nabízí průběžná integrace se tak jeví jako vhodná volba. [41]

Martin Fowler definuje průběžnou integraci takto [42]:

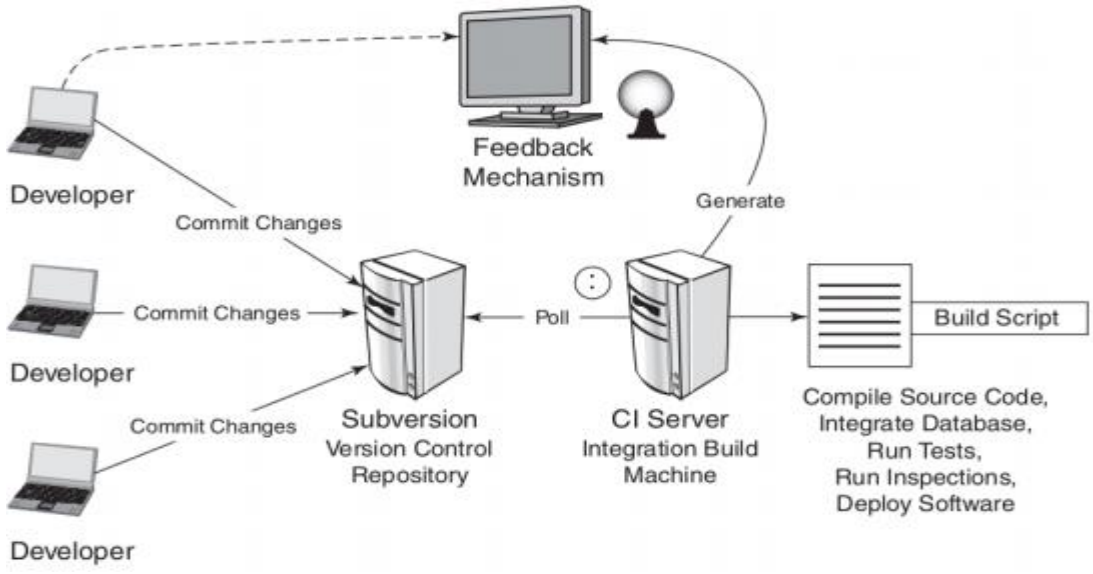
„Průběžná integrace je softwarová vývojová praktika, při které členové týmu integrují jejich práci často, obvykle každá osoba integruje alespoň jednou denně až několikrát za den. Každá integrace je ověřena automatickým sestavením (včetně testování) za účelem detekce integračních chyb tak rychle, jak je to jen možné. Mnoho týmů přichází na to, že tento přístup vede k signifikantní redukci integračních problémů a umožňuje týmu vyvíjet soudržný software mnohem rychleji.“

Paul M. Duvall v knize *Continuous Integration Improving Software Quality and Reducing Risk* navazuje a definuje určité body, které průběžnou integraci charakterizují [43]:

- Mělo by se předejít narušení vzájemné integrace. Docílí se toho tím, že vývojáři sestaví software či část softwaru (upraveného kódu) nejdříve na své pracovní stanici ještě před odesláním do verzovacího systému.

- Je vhodnější rozdělení sestavení kódu softwaru na menší části a každá úprava se musí podrobit testování
- Existuje jeden dedikovaný stroj, který provádí veškeré výpočetní operace spojené s průběžnou integrací, kterému se říká **CI server**
- Zdrojový kód musí být otestován pozitivně a oprava nalezených chyb musí být prioritní
- Průběžné a celkové výsledky sestavovacího procesu a testů se automaticky zpřístupní skrze webový server
- Složitost celého projektu je díky CI redukována pomocí odstranění velké části manuálních a repetetivních činností spojených s úpravami či vydáváním softwaru

Následující obrázek obsahuje spolupracující komponenty podílející se na CI procesu [43]:



Obrázek 18: Schéma praktiky CI [43]

Developer neboli vývojář je iniciátorem procesu. Nemusí se jednat nutně o programátora, ale může tím být myšlen jakýkoliv člen týmu. Potvrzení změn neboli operace commit posílá kód do **Version Control Repository** neboli verzovací systém, jehož smyslem je poskytovat unifikované úložiště zdrojového kódu. **CI server** sleduje změny kódu v úložišti a provádí sestavování projektu. Průběh a výsledek převádí do mechanismu **zpětné vazby**, ten informuje vývojáře o stavu a výsledku sestavování celku. Poslední krok je tzv. **Build Script**, který má za cíl automatizovaně kompilovat, testovat a případně nasazovat změny ve zdrojovém kódu. [40] [43]

3.4.2 CD

Průběžné doručování:

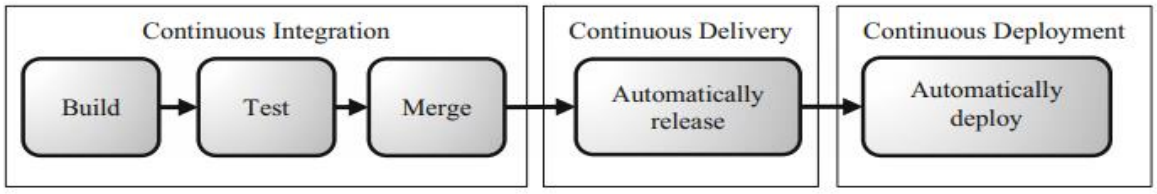
Průběžné doručování, první význam zkratky CD (Continuous Delivery), představuje určité rozšíření CI, kdy přináší záruku, že pokud budou úspěšně dokončeny všechny předchozí kroky, že bude software udržován v nasaditelném stavu. Může být tedy nasazen kdykoliv a tím se snižuje i time-to-market. [40] [44]

Průběžné nasazování:

Průběžné nasazování je druhým možným chápáním zkratky CD (Continuous Deployment) a navazuje na průběžné doručování. Spočívá v automatizovaném způsobu nasazování nových verzí na různá prostředí. Toto celé má smysl pouze bez nutnosti zásahu vývojáře. [40] [44]

CI/CD Pipeline:

Kombinace CI a CD představuje průběžnou automatizaci nejen pro validaci bezchybnosti integrovaného kódu, ale také podporu pro automatizaci ve smyslu doručení a nasazení. Tento celý spojený proces je známý pod pojmem **„*CI/CD pipeline*“**, kde na sebe přímo navazuje CD po úspěšně proběhlém CI. Následující obrázek mapuje základní strukturu CI/CD pipeline a vzájemný vztah procesů [40]:



Obrázek 19: CI/CD pipeline [40]

3.4.3 Nástroje

1. **Verzovací nástroje**

Verzovací nástroje užívají vývojáři ke sledování průběžných změn souborů s kódem softwaru v čase a sledování konfliktních stavů nebo chyb. Lze se vracet k verzím předchozím, a to bez újmy na změnách či bez čekání. Většinou jsou tyto nástroje značeny VCS – version control systém nebo pod DVCS – distributed version control systém. Pro VCS je typické centralizované úložiště spravované týmem, kdežto pro DVCS úložiště

spravuje systém sám a vývojářům poskytuje možnost zrcadlit či větvit. Nejužívanější nástroje jsou například **GIT** (DVCS), **Mercurial** (DVCS), **Apache Subversion** (VCS) nebo **CSV** (VCS). [40] [41]

2. **Nástroje pro statickou analýzu kódu**

Tyto nástroje analyzují kód bez nutnosti jeho spuštění a hledají v něm chyby či nedostatky. Staly se nezbytnou součástí CI/CD a bývají integrovány do procesu sestavování nebo do testovacího procesu. Typickými zástupci mohou být **Code Sonar** (C, C++, C#, Java)**, Veracode** (bezpečnostní pohled, součást SaaS modelu) nebo **Coverity Scan** (podporuje 14 jazyků, např. JavaScript, .NET, Java, …) [40]

3. **Automatizované sestavování (build)**

Automatizované sestavování je proces pro automatizaci kroků při vytvoření nového sestaveného balíku kódu. Sestavováním je myšleno konvertování zdrojového kódu kompilací a sestaveným balíkem je myšlena verze vhodná pro uvolnění a nasazení. Takové nástroje pak mají konfigurovatelný rozvrh nebo pravidla, podle kterých se buildy spouští. Zástupci nástrojů jsou **Apache Maven, Gradle, Apache Ant** atd. [40] [45]

4. **Nástroje pro automatické testy**

CI/CD vyžaduje po každé provedené změně spustit automatizované testy proti nové verzi obsahující nové změny. Aby byl tento proces efektivní, je zapotřebí správná orchestrace testů. Takový proces zahrnuje nastavení prostředí a vybrání správných testovacích scénářů. CI část procesu je orientovaná na integrační testy, kdežto CD část, potažmo celá CI/CD pipeline vyžaduje podporu pro funkční a zátěžové testy, k čemuž nám pomohou příslušné nástroje. Populární a nejvíce užívané jsou **JUnit** (jednotkové testy), **Selenium** (UI testy) nebo **Selenide** (rozšíření Selenia pro UI testy), **Cucumber** (BDD (Behavior Driven Development) framework užívaný nejčastěji se Seleniem pro akceptační testování), **Apache JMeter** (zátěžové a funkční testy) a další. [40] [5]

5. **CI/CD Servery**

CI/CD servery jsou nástroje, které integrují, sestavují a testují software automaticky. V dnešní době již existuje široká škála těchto nástrojů a také různé kombinace. Některé obsahují pouze CI komponentu, jiné CI i CD a některé poskytují

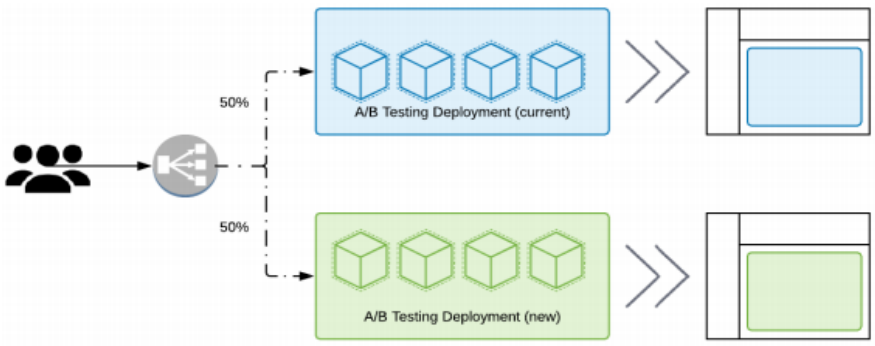
možnosti propojení s nástroji jako je trasování, plánování, správa defektů atp. Některé jsou korporátní, některé open source. Jsou jimi například **Bitbucket** (CI) **a Bamboo** (CD), **Jenkins** (open-source CI/CD), **Azure DevOps** (dříve TFS, celá sada nástrojů), **Travis-CI** (open-source CI), **Spinnaker** (open-source Cloud Native CD), **Ansible** (hybridní cloud CD), **CircleCI** (open-source CI/CD) a další. [40] [46] [47] [48] [49] [50]

3.4.4 Pohled z hlediska automatizovaného testování na CI/CD

Automatizované testování je zcela kritické pro CI/CD procesy, protože kdyby nedošlo k jejich spuštění, porušila by se kontinuita, a tudíž průběžná integrace by nebyla průběžnou. Proto aby svým zákazníkům mohli zaručit určitou kvalitu, musí vzniknout komplexní sada automatizovaných testů integrovaná do procesu vývoje. Tím získáme přehled o jeho stavu už během vývoje i menších změn a můžeme tak reagovat velice pružně. [40] [41]

Ani automatizované testy integrované do CI/CD nám ale negarantují bezchybný aplikační software, pouze zvyšují jeho důvěryhodnost. Jsou zkrátka nevhodné k prokazování nepřítomnosti chyb, jsou pouze cestou k jejich odhalení v rámci předem definovaného pokrytí. [44]

Zajímavou technikou je pak tzv. **A/B testování nasazení**, kterou testujeme vlastnosti frontendové aplikace. Technika si zakládá na zpětné vazbě přímo od uživatelů, a je tak provozována zejména na produkčním prostředí. Reakce uživatelů ukáží, jestli jsou provedené změny intuitivní a vřele přijímány. Princip dále spočívá v tom, že se provedené změny nasadí pouze 50 % uživatelům a poté se vyhodnotí jejich zpětná vazba. Díky CI/CD a Cloud Native je velice jednoduché tuto techniku implementovat do procesu vývoje. [51]



Obrázek 20: A/B testing deployment [51]

4 Vlastní práce

Praktická část této práce se zabývá aplikací teoretických znalostí nabytých literární rešerší a analýzou procesů implementace automatizovaného testování softwaru z pohledu životního cyklu vývoje softwaru přístupem CI/CD a agilních metodik. Analytická část této práce popisuje výchozí a aktuální situaci ve společnosti ZOOM International a.s. Při realizaci této části bylo vycházeno z existujícího technického řešení implementace, vývoje a testování frontendového systému aplikací kontaktních center. Dále jsou vytvořeny modelované podnikové procesy automatizovaného testování v CI/CD a agilního vývoje za pomoci Business Process Model Notation (BPMN). Z důvodu bezpečnosti a zachování tajemství a know-how společnosti jsou reálné údaje pozměněny tak, aby nebylo možné zjistit informace o skutečné infrastruktuře, ale zároveň tak, aby zůstal zachován princip modelovaného procesu.

Mezi činnosti autora práce v rámci společnosti patří činnosti spojené s pozicí *Quality Assurance Engineer (zkratka – QA)* v týmu, který je součástí DevOps metodiky vývoje softwarových produktů společnosti. V agilním prostředí vývoje dochází i k překryvům jednotlivých týmů, a tak jsou ve společnosti vytvořené jakési abstraktní týmy, které sdružují například právě pozice QA.

4.1 Charakteristika společnosti

Vybraná společnost nese název ZOOM International a.s. a byla založena v roce 1999 v Praze. V roce 2020 však byla z obchodních účelů její část přejmenována na Elevēo. Její hlavním zaměřením je vývoj softwarových produktů a řešení, která jsou určena pro kontaktní centra (známější označení – call centra) po celém světě. Mezi produkty společnosti patří nahrávání telefonních hovorů, nahrávání video hovorů, rozpoznávání řeči pomocí hlasové analýzy, hodnocení agentů kontaktních center, plánování směn agentů a další činnosti a operace využitelné v každodenních situacích kontaktního centra. V současné době má společnost více než 170 zaměstnanců a působí ve více než 90 zemích po celém světě. Centrála sídlí v Praze a další pobočky jsou umístěné v USA, na Slovensku, v Rusku nebo Velké Británii. Vývojová centra jsou ale pouze v Praze a na Slovensku.

4.2 Analýza aktuálního stavu ve vybrané společnosti

Společnost z důvodu udržení schopnosti konkurence začala s vývojem softwarových produktů, které je možné nasadit on-premise, ale i v cloudu. V současné době nabízí svým zákazníkům 3 varianty:

1) Elevēo As A Service
    - V první řadě snaha o přechod z on-premise na cloud
    - Hosting je v režii společnosti
    - Aktualizace produktů je zdarma, platba formou předplatného
2) Subscription
    - Oproti předchozímu nabízí on-premise nebo cloudový hosting
3) Pay Per Use
    - Platba pouze za reálně využité prostředky
    - Cloudové řešení na bázi Amazon Connect

4.2.1 IT infrastruktura

Jak si lze povšimnout, žádná z variant zatím aktuálně nenabízí cloud-native přístup k provozu infrastruktury softwarových produktů. Cloud-native je aktuální téma vývoje nové verze softwarového produktu společnosti. Tím pádem začaly být kladeny větší nároky na dynamičtější a volnější procesy vývoje a testování. Z těchto důvodů, ale i z důvodu potřeby cloudového řešení musela společnost zavést užívání dvojí infrastruktury.

Prvním typem je on-premise infrastruktura, kterou společnost dlouhodobě buduje nejen pro provozní účely, ale primárně i pro vývoj všech produktů a běh všech podpůrných složek systému potřebných pro vývoj a testování produktů. Tato infrastruktura je zejména z důvodu jednoduchosti umístěna v datovém centru v Praze.

Druhým typem nebo spíše zdrojem výpočetní infrastruktury, kterou společnosti využívá je cloudová služba Amazon AWS. Z důvodu vyšších provozních nákladů společnost rozhodla o využívání tohoto zdroje pouze pro projekty a produkty zákazníků, tudíž pro preprodukční a produkční prostředí. Jen pro představu, tyto náklady by dosahovaly zhruba trojnásobku on-premise infrastruktury. Tento typ tedy není využíván pro potřeby

vývoje ani testování v procesu vývoje i z důvodu, že je obtížně či nemožné některé podpůrné systémy migrovat do služby AWS, jelikož toto provozovatel nepodporuje.

Společnost ZOOM aktuálně využívá následující produkty Amazon Web Services:

- Amazon VPC – Za účelem vytvoření privátní infrastruktury pro virtuální síť
- Amazon RDS – Relační databázový model s implementací SQL
- Amazon S3 – Datové úložiště pro klienty ve formě webového rozhraní
- Amazon EC2 – za účely škálování potřebné výpočetní kapacity
- Amazon ALB – Load Balancer
- Amazon ACM – HTTPS certifikát pro ALB
- Amazon ECR – Elastic Container Registry – registr pro správu kontejnerů
- Amazon EKS – Elastic Kubernetes Service – správa flexibility aplikace v Kubernetes

Aktuální on-premise část infrastruktury je primárně využívána týmy R&D (Research and Development – týmy v rámci DevOps metodiky v prostředí vybrané společnosti). Prostředí a platforma pro virtualizaci je vytvořena technologiemi společnosti VMWare, konkrétně VMWare vSphere. Celkový objem čítá přes 850 virtuálních strojů, přičemž přibližně 550 virtuálních strojů slouží potřebám vývoje a testování. Ostatní jsou určené pro účely jiných oddělení jako produkční virtuální servery nebo provozní systémy. Přes 90 % všech strojů využívá operační systém CentOS Linux 7 či 8. Zbylých 10 % zastávají jiné Linuxové distribuce či operační systém Windows.

Správu celé této infrastruktury má na starosti vnitřní IT oddělení společnosti a stará se tak o činnosti spojené se správou hardwaru a softwaru. V rámci řešené virtualizační problematiky spojené s vývojem a testováním však končí odpovědnost IT oddělení vytvořením a přidělením virtuálních strojů danému zaměstnanci. Ten pak přebírá odpovědnost za svoje stroje v rámci instalace operačního systému až po instalaci a implementaci softwarového produktu či jeho části potřebné pro dané účely aktuálního vývoje či testování. Práva na manipulaci a správou stroje však stále náleží IT oddělení a nikoliv zaměstnanci. To umožňuje zajištění bezpečnosti, ale také vztažení oprávnění na určitý denní úkol vývojáře či testera.

V základu se tedy ve vybrané společnosti pro účely vývoje a testování jedná o IaaS infrastrukturu a virtuální privátní cloud. Z hlediska zákaznických požadavků však ve většině případů není realizovatelné provozovat klientské aplikace jako PaaS či SaaS infrastrukturu či nasazovat klientovi aplikaci v modelu privátního cloudu. Někteří klienti si kvůli svojí bezpečností politice nemohou dovolit vystavit svoje data či procesy mimo svou uzavřenou infrastrukturu.

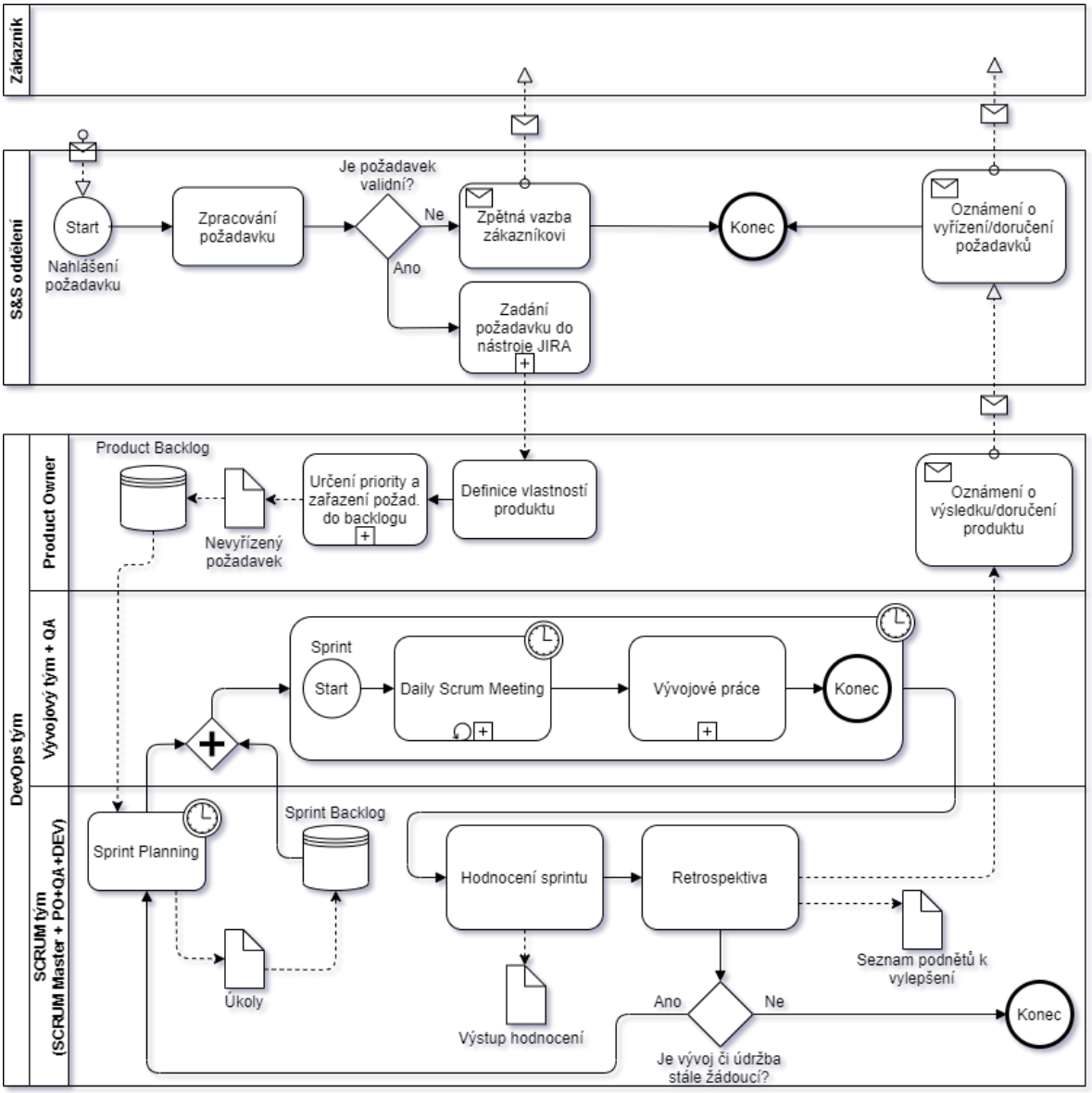
4.2.2 Vývoj a DevOps týmy

Jak zde již bylo zmíněno, vývoj a testování má na starosti oddělení R&D (Research and development). V současné době je toto oddělení členěno do 10 celků, týmů, které pracují na jednotlivých částech systému, do kterého jsou sdružené dílčí softwarové produkty. Tyto týmy čítají povětšinou od 5 do 12 členů. V celkovém tzv. high-level pohledu spravují a vyvíjejí mezi 60 a 70 službami.

Jak si týmy spravují metodiky vývoje záleží čistě na jejich vnitřní dohodě. Důležité je ale zaměření na kvalitu dodávaného produktu a tím pádem je za kvalitu výsledku zodpovědný celý tým. V každém týmu je přítomný 1 až 2 pracovníci QA (Quality Assurance), kteří dohlíží a pomáhají zajišťovat správnými metodikami požadovanou kvalitu. Reprezentantem obchodních požadavků je tzv. Product Owner neboli vlastník produktu. Ten výsledek vyvinutý týmem zaštiťuje a prezentuje směrem k ostatním týmům i dále do firmy. Je také určující ve smyslu náplně práce, která je potřeba za daný časový úsek doručit. Velmi důležitý je také faktor sdílení informací a postupů, a tak velmi často dochází ke krátkodobým výpomocem či transferům vývojářů či QA mezi týmy.

Nejčetnější metodikou používanou napříč týmy je tzv. SCRUM. Scrum se řadí mezi nejznámější agilní metodiky. Tato metodika přináší do týmu další roli zvanou Scrum Master, která v podstatě dopomáhá k dodržování tzv. ceremonií, na kterých se týmy vnitřně předem domluvili. Stejně jako Product Owner, tak i Scrum Master nenáleží pouze jednomu týmu, ale dochází k tomu, že jsou tyto pozice sdílené více týmy. To přináší a umožňuje bližší a přesnější koordinaci práce napříč týmy a mezi týmy dochází mnohdy k inspiraci

a výpomoci mezi sebou. Následující obrázek znázorňuje proces a metodiku Scrum užívanou v prostředí společnosti a životní cyklus požadavku v tomto procesu:



Obrázek 21: Proces a metodika SCRUM [zdroj: autor]

Roli klienta (client) v prostředí společnosti zastupuje Product Owner a ostatní týmy. Vstupy od koncových uživatelů či zákazníků prochází skrze oddělení S&S (Support and Services), které slouží jako komunikační oddělení podpory pro zákazníky. Toto oddělení zpracovává požadavky či přidává vlastní skrze nástroj *JIRA* od firmy *Atlassian*, do jejíž rodiny patří i *Bamboo* pro CI/CD či *BitBucket* pro úložiště zdrojového kódu. Zde vstupuje do procesu metodiky role *Product Owner*, který definuje požadované vlastnosti produktu,

které jsou zapotřebí opravit či implementovat. Dále požadavky seřadí a prioritizuje do strukturovaného seznamu. Takový seznam se v metodice *Scrum* jmenuje *Product Backlog* neboli zásobník nevyřízených požadavků. Následuje první celotýmová ceremonie zvaná *Sprint Planning* neboli plánování sprintu. *Sprintem* se rozumí časově omezená jednotka (nejčastěji týmy ve společnosti volí 1 až 2 týdny, ale sprint může trvat i déle – záleží na domluvě v daném týmu) dedikovaná pro vývoj a odbavení vybraných požadavků neboli *Tasků* (Tasks = Úkoly/Úlohy). Tyto úlohy jsou tedy během ceremonie plánování sprintu tzv. přidané do sprintu (*Sprint Backlog)*, což znamená, že se tým zavazuje je ve vymezeném čase pro sprint realizovat. Do takto vymezeného objemu úloh se již žádné další v průběhu trvání sprintu nemají přidávat, ale není to vyloučeno a také se tak v týmech běžně děje. Po dobu trvání sprintu je iterována další ceremonie, kterou je tzv. *Daily Scrum Meeting* neboli denní scrum setkání. Ta slouží členům týmu ke vzájemné synchronizaci a symbolicky každý den ráno zahajuje práci v ten daný den. Po celou dobu je týmu nápomocná přítomnost *Scrum Mastera,* který týmu obvykle doporučuje vhodné či další kroky. Po uplynutí stanovené doby se obvykle týmy domluvili praktikovat další ceremonie zvané *Retrospektiva* a *Hodnocení sprintu*. Při hodnocení dochází k evaluaci sprintu a zdali bylo dosaženo stanovených cílů. *Retrospektiva* je poté zaměřena na možnost ponaučení a vylepšení stávajících procesů, přidání nových či odstranění určitých překážek do dalšího sprintu. Poté se celý cyklus opakuje počínaje plánováním sprintu. Sprinty jsou opakovány v podstatě do „nekonečna“, jelikož mají pouze dílčí termíny doručení (release procesy). Tím pádem ukončení cyklu sprintů nastane až v momentě, kdy společnost přestane produkt (či jeho část) vyvíjet a podporovat nebo je tým rozpuštěn (vývoj či údržba není nadále žádoucí).

V části popisující IT infrastrukturu společnosti bylo analyzováno, že během vývoje a testování jsou pracovníkům k dispozici virtuální stroje. Každý pracovník má 2 soukromé virtuální stroje, přičemž nad nimi má plnou kontrolu. Z hlediska prostředí se tak jedná o vývojové DEV a o testovací QA prostředí, kde pro odlišení jsou použita doménová jména a DNS adresy strojů jsou pak ve formátu:

- Vývojové DEV prostředí:

*vm<čísloStroje>.**dev.**<doménaZemě>.<doménaDruhéhoŘádu>.<doménaPrvníhoŘádu>*

- Testovací QA prostředí:

*vm<čísloStroje>.**qa.**<doménaZemě>.<doménaDruhéhoŘádu>.<doménaPrvníhoŘádu>*

Dalšími využívanými prostředími jsou:

- Tréninkové EDU prostředí
- Integrační INT prostředí
- Stabilizační PRE před produkční prostředí
- Produkční prostředí (dle požadavků zákazníka)

Pro sestavování balíčků kódu a integraci jednotlivých komponent vývojáři používají praktiky průběžné integrace CI v prostředí nástroje Bamboo od společnosti Atlassian. Pro úložiště zdrojového kódu je používám nástroj BitBucket také od společnosti Atlassian, díky čemu jsou tyto 2 komponenty jednoduše použitelné a integrovatelný dohromady. Díky integraci verzovacího systému Git je pak velice snadné aplikovat praktiky DevOps a pracovat tak v týmech na principech větvení zdrojového kódu. Následnou problematiku nasazení na požadované prostředí řeší vývojáři, QA pracovníci, ale vlastně všichni, kdo potřebují nasadit určitou verzi produktu na dané prostředí buď stažením a následnou manuální instalací skrze nástroj pro správu virtuálních strojů VMWare, anebo automatizovaně skrze funkcionality CD v nástroji Bamboo. Skrze Bamboo jsou naplánované a spustitelné i plány pro automatizované testy, jako jsou například tzv. E2E neboli end-to-end testy, ale také velmi užitečné noční běhy testů. Vývojáři tedy pak mohou ráno přijít do práce a znají výsledky integračních či jiných testů.

4.2.3 Požadavky na testování a kvalitu

Kvalita softwaru hraje ve vybrané společnosti velmi významnou roli. Požadavky na zajištění kvality a testování jsou strukturovány vzhledem k metodice DevOps, ale mají i určitou hierarchii. Všechny hlavní procesy a akceptační úroveň má na starosti manažer testování. Ten koordinuje pracovníky QA napříč jednotlivými týmy a zajišťuje jejich nezbytnou integraci. Jelikož každý tým vyvíjí a udržuje jenom určitou část výsledného produktu, je zapotřebí častá mezi týmová komunikace a sdílení postupů a přístupů k testování. Často dochází k potřebě integrace s částí produktu, kterou vyvíjí jiný tým a tím pádem je potřeba tyto části důkladně integračně otestovat. Oddělení zjištění kvality pak definuje a garantuje společnosti a týmům určité služby:

- Definování kvalitativních akceptačních kritérií
- Dohled nad testováním během období sprintů
- Akceptace doručovaných balíčků z hlediska kvality
- Testování
- Nasazování
- Management konce životního cyklu produktu

Zmiňovaná hierarchie funguje tím způsobem, že oddělení Quality Assurance definuje určitá kritéria, která je nezbytné splnit, aby mohl být výsledný softwarový produkt přijat jako dostatečně kvalitní. Tato kritéria pak slouží jednotlivým týmům jako doporučení pro nastavení jejich dílčích standardů pro kvalitu jejich části produktu. Týmy pak většinou na základě těchto kritérií definují svoje vnitřní DOD (Definition Of Done), čímž se rozumí sada kritérií, které musí každý jednotlivý zadaný úkol splňovat, aby jej bylo možno považovat za hotový. Jak pro vývojáře, tak i pro QA, kteří programují automatizované testy jsou zároveň závazné standardy kódování. Ty definují požadovanou a dohodnutou podobu zdrojového kódu, aby byl výsledný produkt jednotný a ucelený.

Definition Of Done jednoho týmu vypadá například takto:

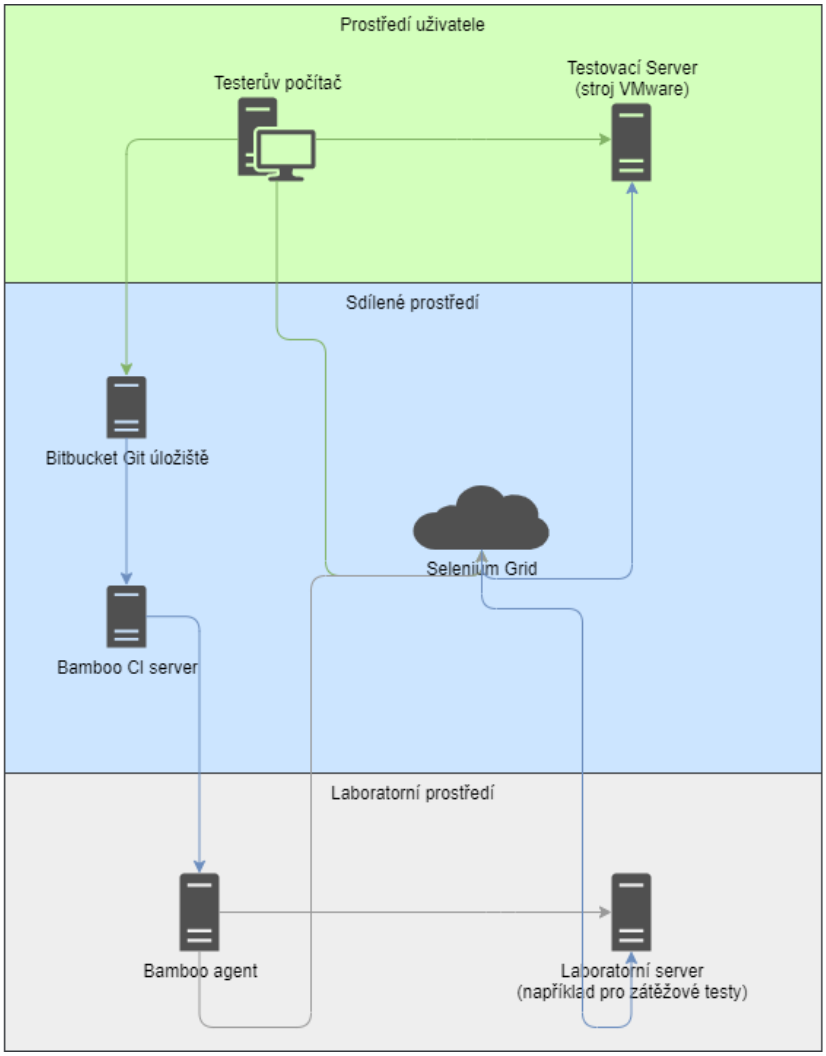
Typ / úkol	Co má být splněno
Komponenty kódu	Výsledný kód splňuje standardy pro kódování
Jednotkové testy	Vytvoření/Úprava jednotkových testů, testy procházejí na 100 % a nenachází žádnou chybu v softwaru
Pokrytí jednotkovými testy	Kód aplikace bude pokrytý minimálně ze 70 %
Revize kódu	Nalezené problémy jsou reportovány nebo opraveny
Akceptační (E2E) testy	Vytvoření/Úprava end-to-end testů, procházejí na 100 %
Integrační testy	Vytvoření/Úprava integračních testů, procházejí na 100 %
Zátěžové testy	Vytvoření/Úprava zátěžových testů, procházejí na 100 %
Úprava uživatelské příručky	Úprava revidována začleněna
Skripty pro sestavení balíčku	Sestavení balíčku proběhlo bez chybových hlášek

Tabulka 8: Definition Of Done

Pro udržení komponent naprogramovaných dle standardů kódování vývojáři používají nástroje pro statickou analýzu kódu, což buduje prvek prevence proti nejzákladnějším chybám. Dále je zdrojový kód pokryt minimálně 70 % jednotkových testů, které nesmí objevit žádnou chybu. Revize kódu neboli *Code Review* je nedílnou součástí CI/CD procesů a je tak posledním preventivním záchytným bodem, kdy se může prověřit kvalita kódu sama o sobě. Revizi provádí vždy 1-2 další vývojáři dle složitosti úkolu. Pokud je kód schválen, je sloučen s kódem v hlavní vývojové větvi za pomoci nástroje GIT skrze BitBucket. Takto sloučený kód pak QA nasadí na testovací prostředí a podle případu provede testovací proces.

Pokud se jedná o novou funkcionalitu, tak ve většině případů není automatizovaný test naprogramován a testováni se provádí manuálně. Některé týmy proto do Definition Of Done vkládají nutnost vytvoření automatizovaného testu před sloučením kódu do hlavní větve. Dále jsou podle povahy úkolu spuštěny integrační nebo zátěžové testy na příslušných prostředích. Zátěžové testy probíhají v tzv. zátěžové laboratoři, kde jsou speciálně pro tyto potřeby vyčleněny stroje dle parametrů z praxe a jsou podrobeny příslušné zátěži, zdali produkt obstojí různým frekvencím provozu. Jelikož se jedná o front-end aplikaci, je nutné prověřit pomocí parametrů daných automatizovaným testům funkčnost ve všech společností podporovaných prohlížečích. V některých případech je ale nutné manuální dotestování, aby se obsáhly i případy, na které pouhý stroj nestačí.

Schéma infrastruktury testovacího prostředí pak vypadá následovně:

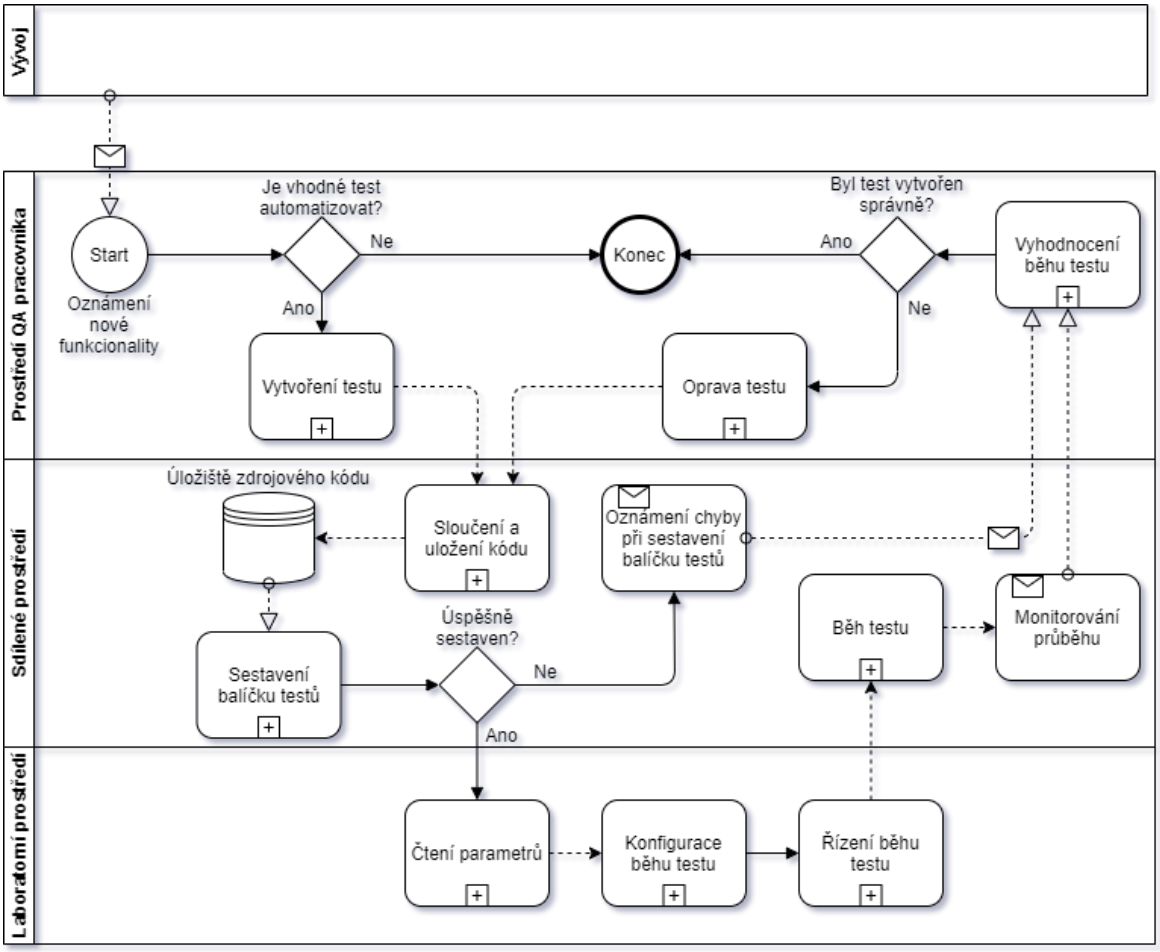


Obrázek 22: Infrastruktura testovacího prostředí [zdroj: autor]

Řídícím uzlem je *testerův počítač*. *Testovací server* obsahuje právě nasazenou verzi softwaru vhodnou k testování. Ze svého PC pak řídí jakým způsobem, jaké testy a kde se mají spouštět. V zásadě se tak děje buď tzv. lokálně a nebo skrze sdílené prostředí. Lokálně má pak také na výběr, zdali chce testy reálně spustit na hardwaru svého počítače anebo skze nástroj *Selenium Grid*. *Selenium Grid* spustí testy na dedikovaném laboratorním serveru, ale tester stále vidí výsledky a přehled na svém počítači bez nutnosti přihlašování na server nebo jsou testy spuštěny nad připraveným testovacím server s testovacím prostředím. Druhým způsobem je spouštění skrze nástroj *Bamboo*. Zde jsou nakonfigurované scénáře běhů a je nutné tedy pouze upravit parametry jako je url serveru testovacího prostředí apod. Zde dojde k inicializaci *Bamboo agenta*, který zajistí spuštění

skrze *Selenium Grid* nad testovacím či laboratorním prostředím. Výsledky testování jsou pak zobrazeny v *Bamboo* za využití reportovacího nástroje *Allure Test Report*.

Proces vytvoření automatizovaného testu ve výše zmíněné infrastruktuře testovacího prostředí je popsán na následujícím modelu:

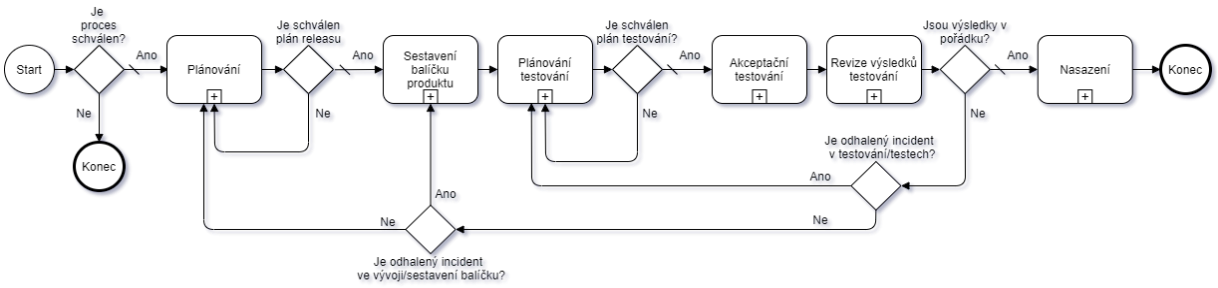


Obrázek 23: Proces vytvoření automatického testu [zdroj: autor]

4.2.4 **Release proces**

Release proces neboli proces nasazování či uvolňování verze softwaru do produkčního prostředí je v prostředí vybrané společnosti řízen release manažerem. Ten koordinuje činnosti s tím spojené na základě výsledků dílčích týmů. Verzí softwaru se rozumí uvedení nových funkcionalit (tzv. new features release), ale i oprava chyb, které nebyly odhaleny během testování či vylepšení na žádost zákazníků (tzv. patch release –

záplata). Release proces lze popsat jako sekvenční sled stavových činností, které jsou závislé na všech dotčených týmech. Jednotlivé činnosti jsou popsány na následujícím obrázku:



Obrázek 24: Release proces [zdroj: autor]

Proces startuje schválením procesu manažery. Následuje proces plánování vedený release manažerem, který zjišťuje dostupné kapacity a zdroje a definuje požadavky, termíny a objem práce. Následuje schválení plánu (případné zamítnutí a doplnění), která ústí v zahájení vývojových prací. Požadovaným výsledkem vývoje je sestavení balíčku produktu, který je nazván tzv. *release kandidátem*, čímž je myšlen softwarový produkt dostatečně kompletní a splňující *Definition of done*, který je připravený na podrobení se akceptačnímu testování. Termínu, kdy jsou zastaveny vývojové práce se říká *code freeze* neboli zmražení kódu k určitému datu bez možnosti dalších úprav. K tomu obvykle dochází 14 dni před plánovaným nasazením. Akceptačnímu testování předchází ještě proces plánování testování a následné schválení plánu testování, kde dochází k definování objemu funkcí nezbytných k otestování (typicky smoke testy, regresní testy a testy nových funkcionalit). Následuje klasifikace automatizovaných a manuálních testů a případné vytvoření doposud nehotových automatizačních testů a sestavení jejich plánu v nástroji Bamboo. Pro manuální testy je využíván nástroj TestRail, který slouží pro sestavení testovací sady a pro udržování testovacích scénářů. Poskytuje také velmi přehledné statistiky o výsledcích testování a je snadno integrovatelný s infrastrukturou pro běh automatizovaných testů. Na tomto procesu se podílí všichni QA pracovníci a vzniká tak v podstatě abstraktní dočasný tým QA oddělení nad množinou všech DevOps týmů. Každý jednotlivý QA přináší poznatky ze svého týmu a stává se tak odborníkem na danou oblast, za jejíž kvalitu také zodpovídá. V následující fázi se revidují výsledky testování a je rozhodnuto buď o zamítnutí implementace, kvůli nedostatečné kvalitě produktu a v takovém případě je stav vrácen do fáze plánování nebo dojde ke schválení a implementaci.

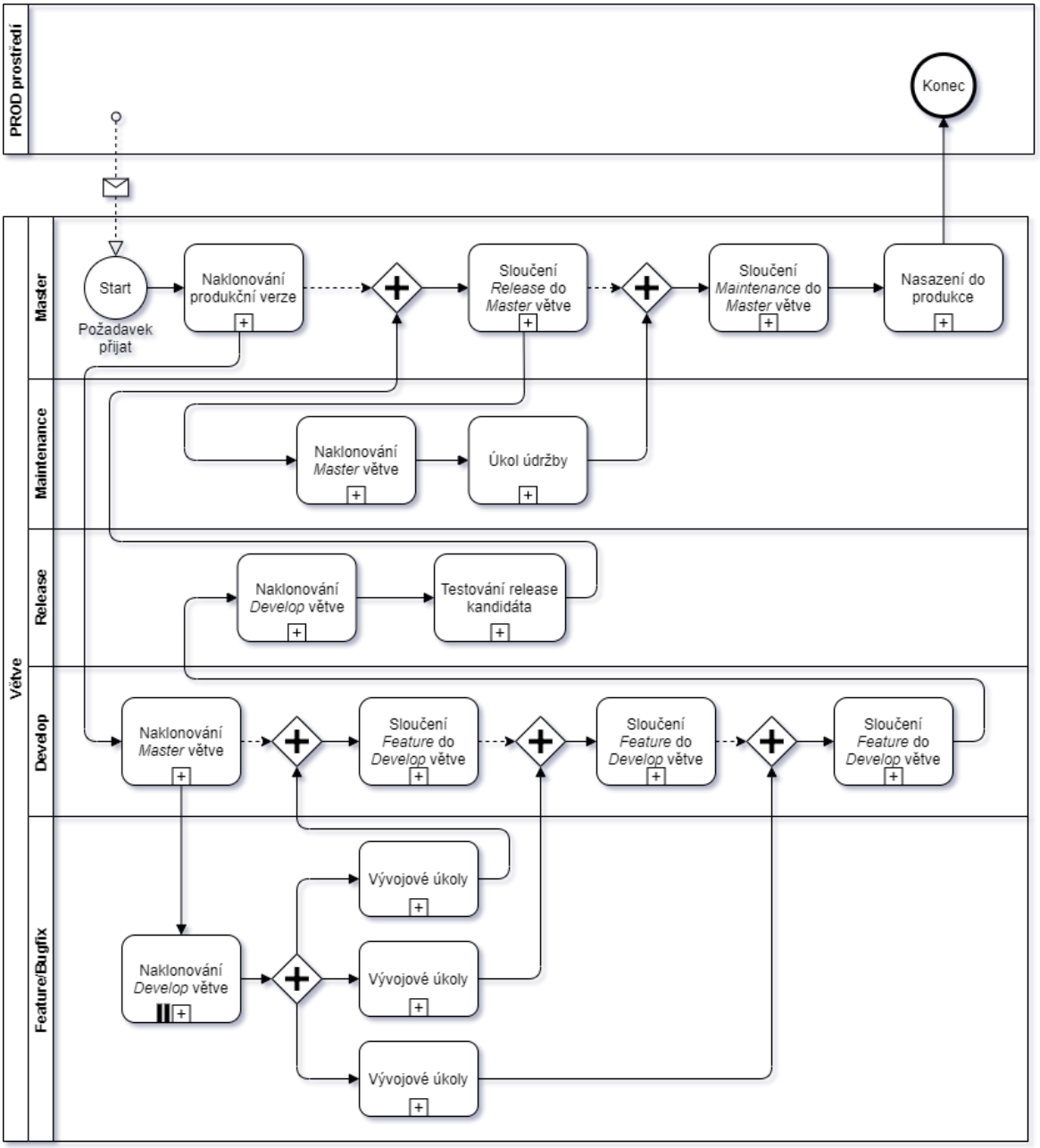
Proces nasazení znamená představení produktu zákazníkovi, který někdy také provádí vlastní nezávislé akceptační testování. Poté je nasazení schváleno, dojde k implementaci dle infrastruktury zákazníka a proces je u konce.

4.2.5 **Proces práce a větvení vývoje**

Výše zmíněné činnosti obsahují určitou komplexitu vnitřních procesů a tím pádem i časovou náročnost. Nejnáročnější činností z pravidla bývá vývoj a sestavování balíčku softwarového produktu. Proces vývojových prací je založený na principu zvaném *branching* neboli větvení. Jak zde již bylo zmíněno, ve společnosti je využívám software pro správu verzí *Git,* který poskytuje komplexní pracovní zázemí pro správu verzí zdrojového kódu a umožňuje jednoduchou integraci prací více vývojářů do jedné hlavní větve. Odtud také vychází název tohoto principu, jelikož středobodem Gitu jsou větve, mezi kterými je definován vzájemný vztah a vzájemná hierarchie a nad kterými jsou prováděny odpovídající operace. Výsledkem procesu sestavování balíčku produktu je tedy operace sloučení prací všech vývojářů (angl. *Merge*).

Větev *Master* je hlavní větev odpovídající verzím produkčního prostředí (například až na data, která zákazník poskytnout nemůže). *Maintenance* větev slouží pro údržbu, kterou není vhodné provádět na produkčním prostředí a zároveň se nejedná o chybu ani novou funkcionalitu. Může také sloužit pro drobnější vývoj v pozdní fázi, kdy je zjištěn nějaký nedostatek na produktu, ale není to kritický problém. *Release* větev slouží pro testování *release kandidáta* v akceptační fázi release procesu. Větev *Develop* je hlavní vývojová větev, kde ale přímo neprobíhá vývoj samotný, ale slučující se do ní všechny účelové vývojové větve. *Feature* větve (v případě opravování chyby se větev jmenuje *BugFix*) slouží pro vývojové úkoly, které vývojáři každodenně plní. Společně se sloučením do hlavní *Develop* větve pak zanikají, kdežto větve *Develop*, *Release* a *Master* jsou zachovány.

Modelový proces vyřízení požadavku a jeho implementace pomocí vývojových větví a prostředí ukazuje následující obrázek:



Obrázek 25: Proces práce a větvení vývoje [zdroj: autor]

4.2.6 Přechod na Cloud Native architekturu

Přechod do architektury, která bude pro cloud nativní je ve vybrané společnosti aktuální téma posledních pár let. Pro potřeby společnosti však není toto téma triviálního charakteru. Velká část klientů zejména kvůli bezpečnosti a strachu nemůže nebo nechce vystavovat své zájmy jakémukoliv necentralizovanému architektonickému návrhu ve smysli umístění IT infrastruktury. Pro takové zákazníky je jediným možným řešením on-premise architektura.

Pro ostatní zákazníky, kteří zvolí cloudové řešení, se ale nejedná o cloud native řešení, nýbrž o jednu ze zmíněných variant na začátku této kapitoly. Kdybychom se vrátili k obrázku 24, nachází se společnost ve fázi „Sestavování balíčku produktu“ velkého release procesu. Týmy napříč společností tedy pracují na přípravě mikroslužeb ze svých stávajících služeb. V současnosti společnost udržuje pro své zákazníky 2 produkční verze produktu, které se liší pouze drobnými změnami vyvolanými požadavky typu údržby od různých klientů. V budoucnu tedy dojde k přidání verze nové, verze s cloud native architekturou, která v několika dalších letech plně nahradí verze současné. Dojde tak k rozbití monolitické architektury, která se ukázala být s rostoucím počtem zákazníků hůře udržitelná.

Při plánování však stále zůstávala nezodpovězena otázka „Co udělat se zákazníky, kteří požadují on-premise řešení?“. Při snaze o zodpovězení vznikalo několik koncepčních řešení, a tak se technické vedení firmy shodlo na implementaci přístupu, který si pro účely této práce dovolím nazvat „*Hybridní cloud native architektura“*. Ve zkratce se jedná o využití návrhu cloud native architektury s mikro službami a technologiemi typickými pro cloud native, ale celé toto řešení bude uzavřeno do jednoho celku a nasazeno na zákazníkovu hardwarovou architekturu. Dojde tak ke sjednocení technologické úrovně ve společnosti, ale nebude splněna myšlenka návrhu cloud native architektury.

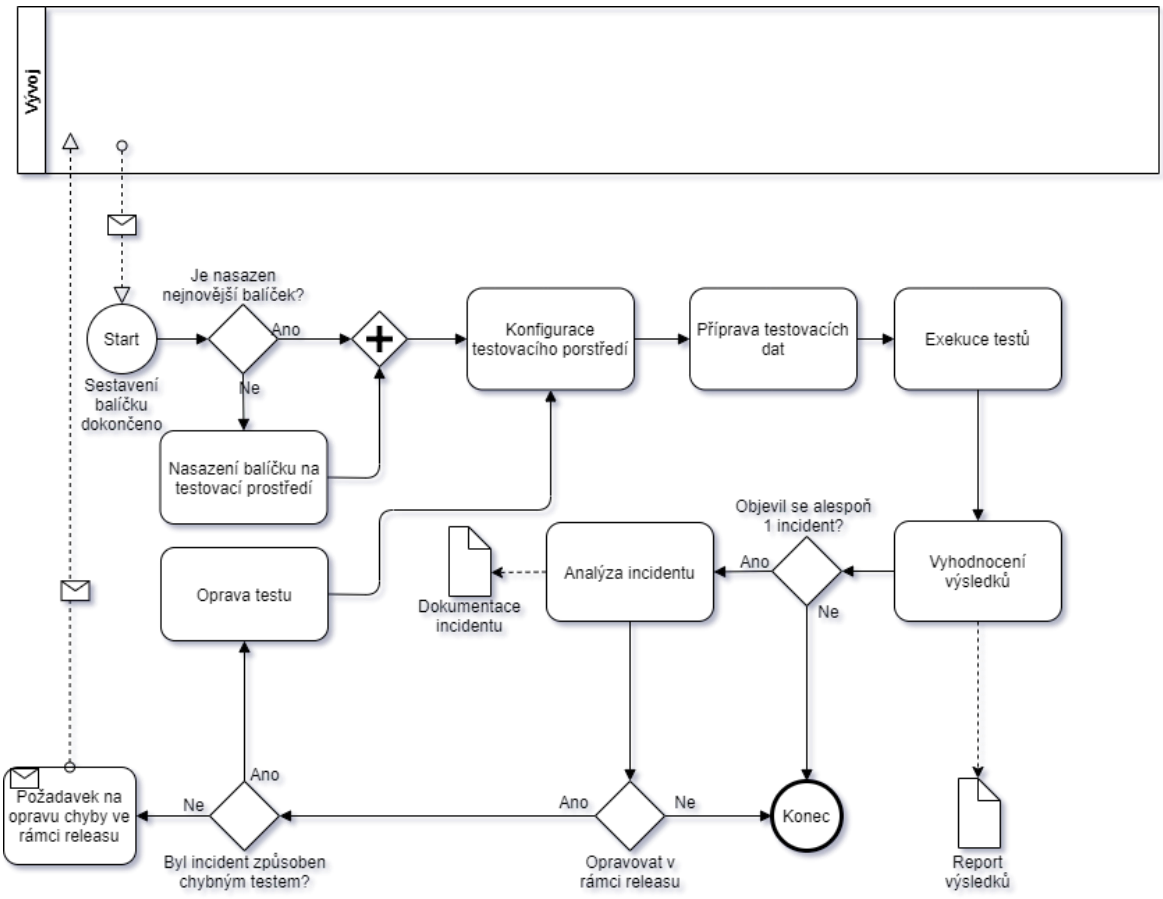
Použitými technologiemi pro cloud native řešení a DevOps pracovní metodiku jsou *Kubernetes* pro orchestraci kontejnerů, *Docker* pro kontejnerizaci a shlukování, *RedHat OpenShift* jako operační systém, několik databázových řešení (Apache Solr, PostreSQL, Oracle), několik produktů od společnosti *Atlassian – Bamboo* pro průběžnou integraci a doručování, *BitBucket* jako úložiště zdrojového kódu, *Jira* pro sledování chyb, SCRUM,

plánování, *Confluence* pro dokumentaci, dále pak *Ansible* pro hybridní cloud průběžné doručování a nasazování, *GIT* pro správu verzí zdrojového kódu a další podpůrné nástroje.

Nástroje pro automatizované testování jsou závislé na každém týmu, ale nejčetněji užívané jsou *JUnit* pro jednotkové testy, implementace *Selenium* nebo *Selenide* v jazyce Java, *Cypress* pro end-to-end testování frontedových částí aplikací (implementace v jazyce JavaScript) a *Apache Maven* či *Gradle* pro sestavení balíčku testů.

4.3 Automatizované testování ve společnosti

4.3.1 Automatizované testování v rámci release procesu

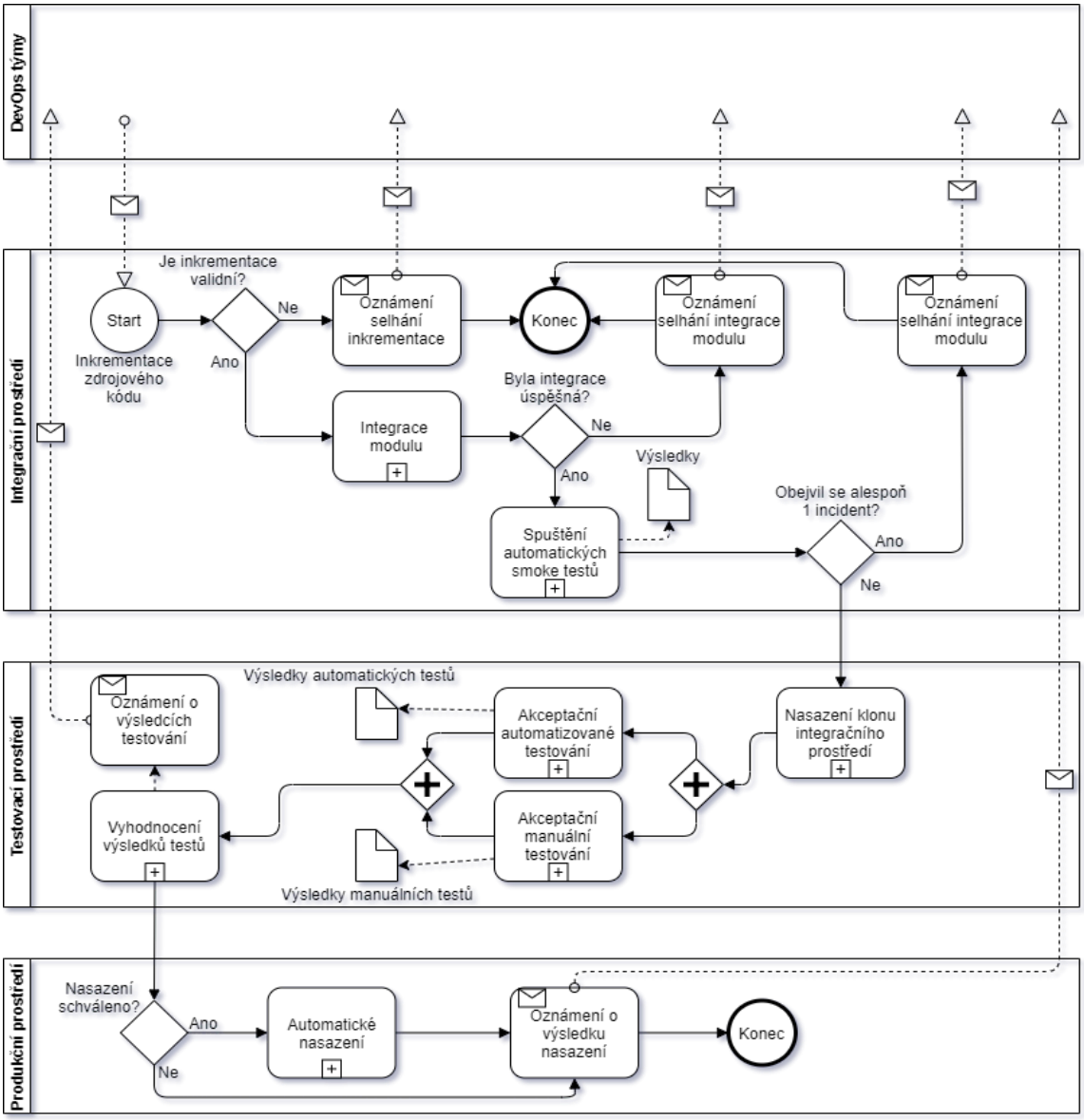


Obrázek 26: Automatizované testování v release procesu [zdroj: autor]

Tento proces automatizovaného testování je aktuálně implementován v release procesu nových verzí (v kapitole *4.2.4 Release proces* se jedná o *Akceptační testování*). Jedná se tedy o přístup a řešení on-premise, nikoliv cloud native.

4.3.2 **Automatizované testování v rámci CI/CD**

Proces automatického testování v rámci CI/CD je ve společnosti momentálně v začátcích. Je to zejména způsobeno aktuálně aktivním procesem přechodu do cloud native návrhu, a tak zatím není vytvořeno stabilní integrační prostředí pro celý produkt. Jeho některé části (moduly) však již integrovány jsou a oddělení R&D je aktuálně používá pro demonstraci postupu pro obchodní část společnosti. Proto je tento následující model z části modelový a stále neimplementovaný jako celek. Například sub proces *Nasazení klonu integračního prostředí* je zatím iniciován manuálně, popřípadě pomocí rozvrhu spouštění nástroji Bamboo. Vize vývojářů je taková, že integrační prostředí bude automaticky obsluhovat nástroj *Ansible,* který byl v této práci zmíněn dříve (kapitola 3.4.3).



Obrázek 27: Automatizované testování v rámci CI/CD [zdroj: autor]

5 Výsledky a diskuse

5.1 Výsledky

5.1.1 Přehled řešené problematiky – Výsledky literární rešerše

Při analýze teoretických východisek bylo zjištěno, že testování je záležitost netriviálního charakteru. Testování nepřináší přímý zisk a znamená výrazné zatížení pro oddělení vývoje z hlediska nákladů. Zpravidla se tak děje u projektů, které s testováním nepočítají od samého začátku a pak jej složitě implementují do svých vývojových procesů. Je tak mnohem výhodnější jak pro společnost, tak pro procesy spojené s vývojem zahrnout testování už v samotných začátcích plánování.

Automatizované testování je dalším faktorem, který je vhodné realizovat už od samotného začátku procesu vývoje softwarového produktu. Nastavením nejvhodnějších metodik vývoje a testování může společnost předejít nepříjemným situacím vyvolaných nedostatečnou kvalitou dodávaného produktu. Ani prostředí aplikací nativních pro cloudové prostředí není výjimkou a v CI/CD se nejvíce osvědčují metodiky agilní. Často rezonujícím a v současnosti široce využívaným způsobem práce zvaným DevOps společnosti velice často docílí odstranění bariér mezi oddělením vývoje a provozu a testování. DevOps klade důraz na kvalitu dodávek produktu a nabízí tak rozdělení těžkopádných velkých dodávek do menších, průběžných a iterativních dodání produktu. Tento proces je navíc vhodný kandidát pro automatizaci i z hlediska testování. Zajištění kvality samotné pak probíhá v procesech velmi úzce spjatých se samotným vývojem, čímž se zvyšuje celková kvalita všech dodávek.

Možnosti, které nabízí cloud computing přináší do světa informačních technologií nepřeberné množství využití a celkově zrychlují přenos informací mezi poskytovatelem a koncovým uživatelem. Bylo tedy pouze otázkou času, kdy vzniknou technologie, které budou tomuto prostředí vlastní, tedy nativní. Jsou však situace, kdy je jejich použití nevhodné a často je to způsobeno i obavami zákazníků, kteří nechtějí vystavovat svá data světu venku mimo své servery a zdi. Neznamená to ale nutně, že se cloud rovná nezabezpečenému přístupu, jde pouze o rizika, která jsou ale spojená s každou technologií. I z tohoto důvodu je tedy udržet kvalitu dodávaného produktu na určité úrovni nebo

se ji rovnou snažit zvyšovat v průběhu dodávek. Při používání agilních metodik je pak méně problémové vyřadit určité části produktu, které ještě nejsou zcela hotové a dodat je zákazníkovi s další dodávkou. V současné době také existuje několik velkých dodavatelů takových infrastruktur, kteří mají různé tarify a lze tak optimalizovat náklady přímo na míru požadovanému řešení. Není tak nutné z hlediska dodavatelské společnosti takovou infrastrukturu přímo pořizovat a provozovat.

Zmíněné cloud native aplikace jsou pak zaměřené na celý životní cyklus produktu. Jejich smyslem je pak zoptimalizovat a co nejvíce zjednodušit přechod od neforemných a monolitických aplikací do formy menších mikroslužeb spravovaných ve formě cloudu. Tento zmíněný přechod byl v této práci vysvětlen na příkladu a případové studii z praktického použití, kde přímo vyplývá, že s lehce rostoucí komplexitou je potřeba změnit přístup ke správě a provozu systémů. Tato potřeba zřejmě způsobila vznik aplikací pro zjednodušenou a zefektivněnou manipulaci a orchestraci takových komponent, které lze libovolně shlukovat a škálovat.

Manuální udržitelnost je u takových systému čím dál tím náročnější, a tak se v určitém bodě začne nabízet implementace CI/CD metodiky na procesy vývoje, provozu a testování jako nejvhodnější. Ve spojení s agilními metodikami jde tak o silnou kombinaci, která umožňuje častěji a spolehlivěji zveřejňovat nové verze vyvíjeného softwarového produktu. Uvnitř společnosti pak pomáhá lépe a efektivněji integrovat práce jednotlivých týmů. Tím pádem je možné maximalizovat rychlost dodání a kvalitu celého softwarového produktu.

5.1.2 Analýza současného stavu v podnikové praxi

Při analýze současného stavu ve vybraném podniku bylo vycházeno z jeho existujícího řešení, které je silně orientováno ve směru k zákazníkovi a jeho potřebám. Jelikož je softwarovým produktem soubor aplikací využitelných zejména v kontaktních centrech, je zde kladen velký důraz na práci s daty a na kvalitu dodávaného řešení, jelikož se zákazníci společnosti potřebují spolehnout na každý proces, který opět poskytují dále ve svých systémech pro své koncové zákazníky.

Vybraný podnik v současné době poskytuje několik hotových řešení, ale žádné zatím nesplňuje charakteristiky cloud native architektury. Tento fakt společnost aktuálně řeší implementací technologií typických pro cloud native prostředí s využitím agilních metodik a CI/CD. Kvalitativně orientované procesy jsou pro společnost důležité, a tak je v každém DevOps týmu přítomný alespoň jeden pracovník oddělení zajišťující kvalitu neboli QA. Všechny tyto týmy využívají tedy agilních metodik ke každodenní organizaci práce a většina těchto týmů používá metodiku zvanou SCRUM.

Testování a kvalita je tedy důležitým faktorem a prvkem v procesu vývoje, který stanovuje požadavky, kde až po jejichž splnění je produkt vhodný k předání zákazníkovi. Optimalizaci zajišťuje vzájemné propojení abstraktním QA týmem, ve kterém všichni jednotliví pracovníci oddělení zajištění kvality sdílí své postupy a navzájem si předkládají konstruktivní kritiky. Definováním všeobecných vnitřních standardů na kvalitu pak společnost sjednocuje rámec všech týmů, které tak mohou lépe balancovat požadavky na kvalitu pro svůj tým, a tudíž i jednotlivé komponenty.

5.1.3 Modelování podnikových procesů

Modelované procesy byly vybírány na základě různých pohledů na roli testování a zajišťování kvality v prostředí vybrané společnosti. Prvním modelovaným procesem bylo užití metodiky SCRUM v závislosti na životní cyklus požadavku vzneseného zákazníkem. Na tomto modelu je vidět koordinace týmových pracovních činností řízená týmem samotným. Cykličnost a poučení z vlastních chyb pak v procesu figuruje jako hlavní pilíř zajišťující optimalizaci a postupné zefektivňování práce a doručování výsledků.

Druhým modelovaným procesem byl zvolen proces vytvoření automatizovaného testu, který může být při správném návrhu podprocesem. Tento proces byl nastíněn i z pohledu fungování v rámci infrastruktury využívané k testování. Determinován je také požadavky typu *Definition Of Done*, které se odrážejí od standardů daných společností, respektive jejím oddělením zajišťování kvality a ty jsou poté upraveny týmy samotnými. Pokud byl test vytvořený v souladu s těmito požadavky, proces vývoje tím končí. V opačném případě je test opraven novou iterací skrze celý tento proces.

Release proces je zde modelován, jelikož povrchně zobrazuje celý proces uvedení verze softwarového produktu do provozu. Obsahuje komponenty od plánování přes vývoj samotný, přes testování až k nasazení verze do produkčního prostředí. Jeho časová náročnost je proměnlivá v závislosti na typu verze softwaru. Verzí softwaru se obecně rozumí uvedení nových funkcionalit, ale může se jednat také o tzv. záplatu chyb nalezených v softwaru nebo o změnové požadavky směrem od zákazníků.

Dalším procesem vybraným k modelování byl proces práce a větvení vývoje. V podnikových týmech dochází k velmi úzké spolupráci vývojářů a QA pracovníků, a tak samotní QA prochází tímto procesem například právě při vývoji automatizovaných testů, které jsou spjaté s vyvíjenou aplikací. Jejich zdrojový kód je pak udržován v závislosti na změnách dané komponenty aplikace, zejména pak při změnách frontendové části.

Automatizované testování v release procesu pak podrobněji přibližuje všechny prováděné činnosti při fázi akceptačního testování, které rozhoduje o tom, zda bude daná testovaná verze softwaru dostatečně kvalitní a vhodná pro nasazení u zákazníka. V současné době se však jedná o testovaní v rámci řešení on-premise, tedy ne cloud native.

Hlavním modelovaným procesem je pak automatizované testování v rámci CI/CD. Společnost, jak zde již bylo zmíněno, teprve implementuje přechod do této metodiky, a tak je tento proces jakýmsi vhledem do budoucna. Jeho některé dílčí části jsou již funkční a k dispozici ostatním týmům, nicméně není aktuálně plně funkční. Při jeho správném návrhu však může ušetřit velkou část nákladů, které by přineslo neustálé optimalizování a předělávání jednotlivých komponent.

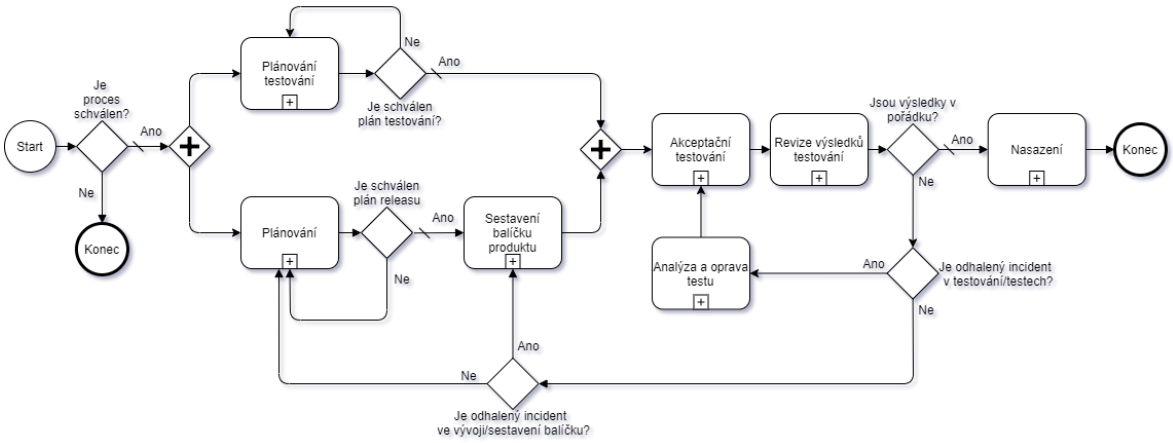
Proces přechodu na Cloud Native architekturu modelován nebyl, protože v prostředí společnosti odpovídá přechodu popsaném v kapitole *3.2.3 – Proces přechodu na architekturu mikroslužeb* této práce. Pro modelování procesů testování není pak nijak významný ve smyslu toho, že jeho součástí jsou procesy výše zmíněné.

5.2 Diskuze

Vybraná společnost si dle mého názoru vede velmi obstojně, pokud porovnávám s fakty zjištěnými během literární rešerše. Přemýšlí progresivně o rozšíření své nabídky produktů o cloud native přístup, který je z moderního pohledu mnohdy i nezbytný a žádoucí. Nezapomíná přitom na důležitost testování a zajišťování kvality a takřka ve všech procesech spjatých s vývojem vystupuje přítomnost alespoň jednoho QA pracovníka.

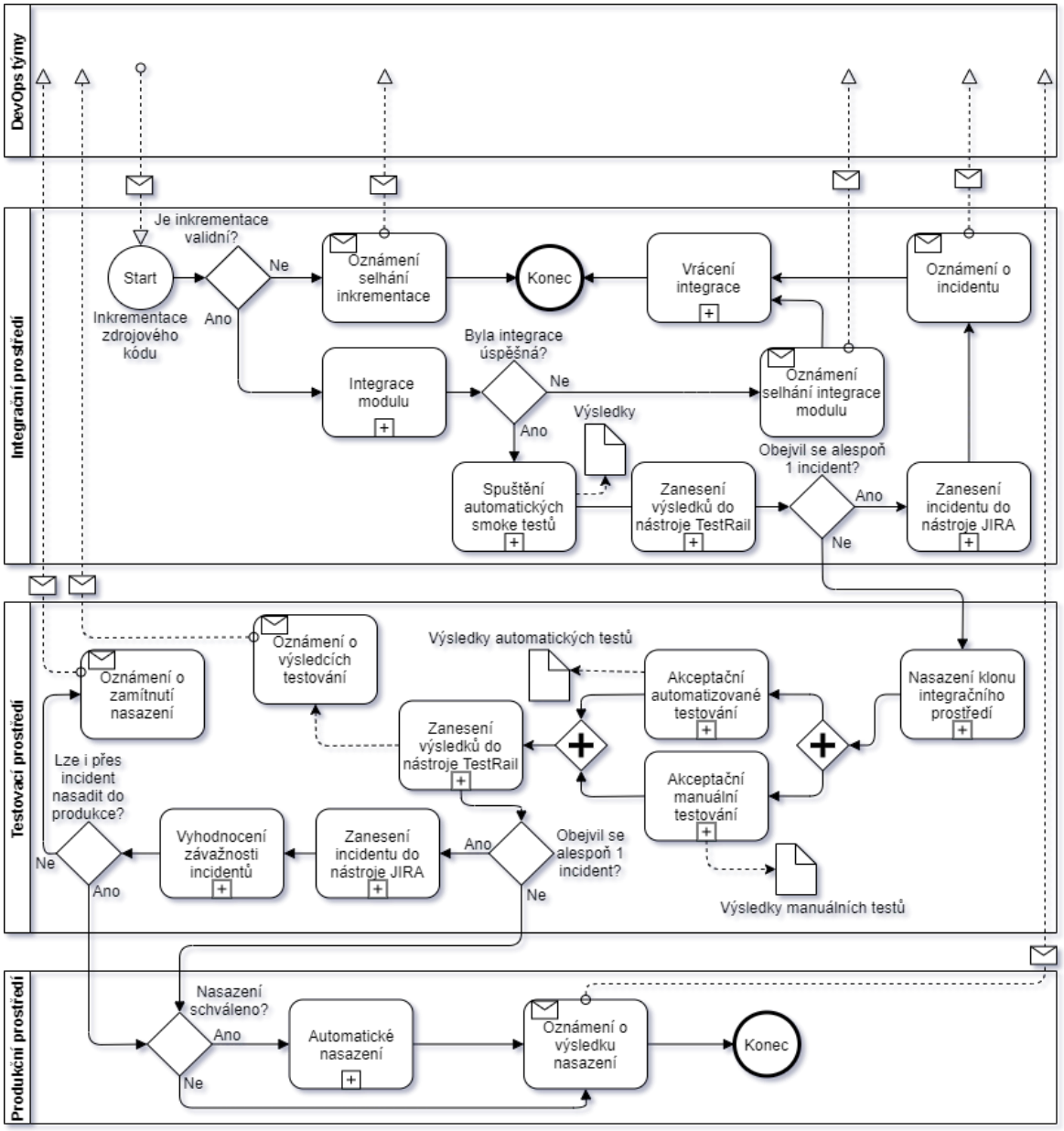
5.2.1 Návrh na zefektivnění procesu – release proces

Při prvním pohledu na modelovaný release proces mne napadlo, že sekvenční vykonávání činností lze optimalizovat pomocí zavedení paralelního vykonávání činností. Plánování testování by tedy započalo současně s plánováním všech aktivit a tudíž se ušetří čas, který by stál mezi dokončeným vývojem a začátkem testování. Zároveň by podproces Akceptační testování mohl být zahájen okamžitě po ukončení a sestavení balíčku produktu. To by dále přispělo k zavedení a napojení metodik CI a CD, kde by se nynější manuální činnosti a procesy daly pohodlně zautomatizovat. Druhým zefektivněním je navázání přímo na akceptační testování s do-implementací procesu analýzy a opravy testu při odhalení chybného testu či testování. Odstraní se tak vazba na nutnost znovu plánovat testování samotné.



Obrázek 28: Návrh na zefektivnění release procesu [zdroj: autor]

5.2.2 Návrh na zefektivnění procesu – automatizované testování



Obrázek 29: Návrh na zefektivnění procesu automat. testování v CI/CD [zdroj: autor]

Jako první návrh na zefektivnění je přidání sub procesu „Vrácení integrace“ inkrementovaného modulu na jeho předchozí verzi (případně jeho odstranění, jedná-li se o první integrovanou verzi). To umožní celkové zrychlení integračního procesu bez nutnosti nového nasazení. Dále ke zefektivnění povede integrace zanášení výsledků testování do reportovacích nástrojů jako je např. TestRail. V případě nálezu incidentu pak integrace jeho sledování pomocí nástroje JIRA. Sub proces vyhodnocení závažnosti incidentů pak umožní i v případě nálezu incidentu přejít k nasazení, pokud se manažeři rozhodnout, že se nejedná o incidenty blokující či kritické.

6 Závěr

V teoretické části této diplomové práce bylo vysvětleno a prozkoumáno několik klíčových aspektů, které jsou nezbytné znát pro pochopení problematiky automatizovaného testování cloud native aplikací. Dále byly tyto aspekty rozvinuty o představení metodik aplikovaných v běžné praxi vývojovými týmy zabývajícími se agilním vývojem spjatým s různými typy testování. Přehled problematiky testování a kvality byl pak demonstrován na případové studii. Pomocí další případové studie byl popsán postup přechodu od architektury monolitické do architektury cloud native a mikroslužeb. Mikroslužby se ukázaly jako klíčové při CI/CD, kde dochází k inkrementacím a testují se samostatně anebo v kontextu celé aplikace v rámci integračního či testovacího prostředí. Přehledem vytvořeným na základě odborných zdrojů byl pak splněn první z dílčích cílů této práce.

V praktické části pak byly splněny další 2 dílčí cíle. Prvním z nich byl splněn provedením analýzy současného stavu a požadavků pro návrh a optimalizaci testování aplikace, která je v prostředí vybrané společnosti vyvíjena na míru zákazníkům. Analyzovány byly jak IT infrastruktura, struktura jednotlivých oddělení a týmů, tak i procesů implementovaných napříč odděleními a týmy. Další dílčí cíl v praktické části se podařilo splnit pomocí metodiky BPMN a BPD diagramů, kterými byly modelovány procesy stěžejní pro testování a kvalitu softwarového produktu vybrané společnosti.

Posledním dílčím cílem bylo navržení zefektivnění procesu testování v životním cyklu vývoje softwaru. Pro tento účel byly vybrány celkem 2 procesy, kde jsem se pokusil v prvním případě navrhnout zefektivnění testování z hlediska jeho pozice v řetězci činností a v případě druhém pak metodiku testování a reportování výsledků samotných v rámci CI/CD procesech.

Splněním dílčích cílů se mi podařilo splnit také cíl hlavní, kde došlo k analyzování procesů automatizovaného testování softwaru z pohledu životního cyklu přístupem CI/CD. Při implementaci výše zmíněných doporučení může dojít ke zefektivnění testování i vývoje samotného. Tím dojde k podrobnějšímu sledování v jednotlivých fázích a tím dojde i k včasnému odchycení případných chyb, které lze velice pružně opravit a doručit tak zákazníkovi kvalitnější produkt.

7 Seznam použitých zdrojů

[1] BUCHALCEVOVÁ, Alena a Jan KUČERA. Hodnocení metodik vývoje informačních systémů z pohledu testování. Systémová integrace. 2008, 15*(2), 13. ISSN ISSN 1210-9479*.

[2] BALALAIE, Armin, Abbas HEYDARNOORI a Pooyan JAMSHIDI. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. European Conference on *ServiceOriented and Cloud Computing [online]*. *Springer, 2015*, , 1-15 [cit. 2021-01-04]. Dostupné z: doi:arXiv:1507.08217

[3] KRATZKE, Nane a Peter-Christian QUINT. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. The *Journal of Systems and Software [online]*. *2017*, (126), 1-16 [cit. 2021-01-19].

[4] Draw.io [online]. Dostupné také z: https://app.diagrams.net/

[5] BUREŠ, Miroslav, Miroslav RENDA, Michal DOLEŽEL, Peter SVOBODA, Zdeněk GRÖSSL, Martin KOMÁREK, Ondřej MACEK a Radoslav MLYNÁŘ. Efektivní testování softwaru: Klíčové otázky *pro efektivitu testovacího procesu*. 1. vydání. Praha: © Grada Publishing, a.s., 2016. ISBN 978-80-271-9389-9.

[6] SLÁDEK, Martin. Automatizace testovacích scénářů. Praha, 2018. Bakalářská práce. Česká zemědělská univerzita v Praze. Vedoucí práce Ing. Josef Pavlíček, Ph.D.

[7] BOROVCOVÁ, Anna. Testování webových aplikací. Praha, 2008. Diplomová práce. Univerzita Karlova v Praze.

[8] MAZOCH, Břetislav. Funkční testování webových aplikací. Brno, 2012. Bakalářská práce. Masarykova Univerzita.

[9] PATTON, Ron. Testování softwaru. 1. Praha: Computer Press, 2002. ISBN ISBN 80-722-6636-5.

[10] IEEE 829-1998 - IEEE Standard for Software Test Documentation. IEEE SA, 1998.

[11] CRISPIN, Lisa a Janet GREGORY. Agile testing: a practical guide for testers and agile teams. 1. Boston, USA: Pearson *Education,* Inc., 2009. ISBN 978-0-321-53446-0.

[12] BALALAIE, Armin, Abbas HEYDARNOORI a Pooyan JAMSHIDI. Microservices Architecture Enables DevOps: an Experience Report on Migration to a Cloud-Native Architecture. IEEE Software [online]. 2016, , 1-12 [cit. 2021-01-19]. Dostupné z: doi:10.1109/MS.2016.64

[13] STORBAKKEN, Mandy. DevOps: Where Are We and How Did We Get Here? [online]. , 1 [cit. 2021-01-24]. Dostupné z: https://blogs.vmware.com/management/2020/05/devops-where-are-we-and-how-did-we-get-here.html

[14] BOROVCOVÁ, Anna. Kvalita a testování v České republice. Systémová integrace. Praha, 2011, 18(1), 1-16. ISSN 1804-2716.

[15] SOSINSKY, Barrie. Cloud computing bible. 1. vydání. Indianapolis, Indiana: Wiley Publishing, Inc., 2011. ISBN 978-0-470-90356-8.

[16] SMITKA, Jakub. Výhody a riziká pri nasadení cloud computingu. Banská Bystrica, 2012. Diplomová práce. Bankovní institut vysoká škola Praha. Vedoucí práce Doc. Ing. Marcel Harakaľ, PhD.

[17] KADLEČKOVÁ, Michaela, DiS. Analýza využití cloud computingu z pohledu poskytovatele. Hradec Králové, 2015. Bakalářská práce. Univerzita Hradec Králové.

[18] SSI. The Top 10 Most Used AWS Services. SSI [online]. 2020, , 1 [cit. 2021-01-16]. Dostupné z: https://www.ssi-net.com/the-top-10-most-used-aws-services/

[19] AWS Amazon. AWS Amazon [online]. [cit. 2021-01-17]. Dostupné z: https://aws.amazon.com/

[20] Windows Azure. Windows Azure [online]. [cit. 2021-01-17]. Dostupné z: https://azure.microsoft.com/

[21] Google Cloud. Google Cloud [online]. online, 2021 [cit. 2021-01-17]. Dostupné z: https://cloud.google.com/

[22] CACH, Jan. Implementace hybridního multi-cloud konceptu pro běh distribuovaných aplikací v kontejnerech a virtuálních serverech do enterprise prostředí. Hradec Králové, 2019. Diplomová práce. Univerzita Hradec Králové.

[23] BRUNNER, Sandro, Martin BLÖCHLINGER, Giovanni TOFFETTI, Josef SPILLNER a Thomas BOHNERT. Experimental Evaluation of the Cloud-Native Application Design. IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC*)* [online]. Zurich University of Applied Sciences, School of Engineering, 2015, , 1-6 [cit. 2021-01-18]. Dostupné z: doi:10.1109/UCC.2015.87

[24] WU, Andy. Taking the Cloud-Native Approach with Microservices [online]. Google Inc., 2019, , 1-13 [cit. 2021-01-18]. Dostupné z: https://cloud.google.com/files/Cloud-native-approach-with-microservices.pdf

[25] AL KISWANI, Jalal Hasan Ahmed. Smart-Cloud: A Framework for Cloud Native Applications Development. Reno, 2019. Disertační práce. University of Nevada, Reno. Vedoucí práce Dr. Sergiu M. Dascalu.

[26] Containerization [online]. internet: IBM Cloud Education, 2019 [cit. 2021-01-20]. Dostupné z: https://www.ibm.com/cloud/learn/containerization

[27] What was CoreOS and CoreOS Container Linux? [online]. internet: RedHat OpenShift TECH TOPIC, 2020 [cit. 2021-01-20]. Dostupné z: https://www.openshift.com/learn/topics/coreos

[28] GitLab [online]. internet: GitLab, 2021 [cit. 2021-01-20]. Dostupné z: https://about.gitlab.com/

[29] Jenkins [online]. internet: Jenkins, 2021 [cit. 2021-01-20]. Dostupné z: https://www.jenkins.io/

[30] Docker [online]. internet: Docker, 2021 [cit. 2021-01-20]. Dostupné z: https://www.docker.com/

[31] Docker container use adoption devops clusterhq survey. In: Content Nanobox [online]. 2017 [cit. 2021-01-20]. Dostupné z: https://content.nanobox.io/content/images/2017/06/docker-container-use-adoption-devops-clusterhq-survey.png

[32] COMBE, Theo, Antony MARTIN a Prof. DI PIETRO. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing* [online]. 2016, **2016**, 1-10 [cit. 2021-01-20]. Dostupné z: doi:10.1109/MCC.2016.100

[33] RAYMOND, Eric S. The New Hacker's Dictionary version 4.2.2. tebbo, 2012. ISBN 1743473168.

[34] Docker Registry [online]. internet: Docker docs, 2021 [cit. 2021-01-21]. Dostupné z: https://docs.docker.com/registry/

[35] ELDRIDGE, Isaac. What Is Container Orchestration?. In: New Relic [online]. New Relic, 2018 [cit. 2021-01-21].

[36] Swarm mode overview [online]. internet: Docker docs, 2013-2021 [cit. 2021-01-21]. Dostupné z: https://docs.docker.com/engine/swarm/

[37] Kubernetes [online]. internet: Kubernetes, 2021 [cit. 2021-01-21]. Dostupné z: https://kubernetes.io/

[38] What is Kubernetes? [online]. internet: Google, 2021 [cit. 2021-01-21]. Dostupné z: https://cloud.google.com/learn/what-is-kubernetes

[39] Pods [online]. internet: Kubernetes, 2021 [cit. 2021-01-21]. Dostupné z: https://kubernetes.io/docs/concepts/workloads/pods/

[40] CANO, Patricia Ortegon. A Taxonomy on Continuous Integration and Deployment Tools and Frameworks. New Perspectives in Software Engineering: Proceedings of the *9th International Conference on Software Process Improvement (CIMPS 2020)*. 2020, , 232-336. ISSN 2194-5365. Dostupné z: doi:https://doi.org/10.1007/978-3-030-63329-5_22

[41] LECIÁN, Bc. Tomáš. Vývojářské metody – kontinuální integrace a kontinuální doručování. Zlín, 2019. Diplomová práce. Univerzita Tomáše Bati ve Zlíně.

[42] FOWLER, Martin. Continuous Integration [online]. 2006, , 1 [cit. 2021-01-25]. Dostupné z: https://martinfowler.com/articles/continuousIntegration.html

[43] DUVALL, Paul, Steve MATYAS a Andrew GLOVER. Continuous integration: improving software quality and reducing risk. 1. Addison-Wesley Professional, 2007. ISBN ISBN 978-0-321-33638-5.

[44] MACHÁČ, Martin. Průběžná integrace a průběžná dodávka/nasazení projektu KYPO. Brno, 2018. Diplomová práce. Masarykova univerzita.

[45] Gradle [online]. 2021 [cit. 2021-01-25]. Dostupné z: https://gradle.org/

[46] Spinnaker [online]. 2021 [cit. 2021-01-25]. Dostupné z: https://spinnaker.io/

[47] Ansible [online]. 2021 [cit. 2021-01-25]. Dostupné z: https://www.ansible.com/

[48] CircleCI [online]. 2021 [cit. 2021-01-25]. Dostupné z: https://circleci.com/

[49] Bitbucket [online]. 2021 [cit. 2021-01-25]. Dostupné z: https://www.atlassian.com/software/bitbucket

[50] Bamboo [online]. 2021 [cit. 2021-01-25]. Dostupné z: https://www.atlassian.com/software/bamboo

[51] FUGARO, Luigi a Mauro VOCALE. Hands-On Cloud-Native Microservices with Jakarta EE. 1. Birmingham: Packt Publishing Ltd., 2019. ISBN 978-1-78883-786-6.