Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Automatické testování cloud native aplikací

Bc. Natallia Luhina

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Natallia Luhina

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 problematiku automatického testování aplikací s architekturou mikroslužeb. Hlavním cílem je snížit rizika a zvýšit účinnost procesu vývoje aplikace pomocí použití metodiky Continuous Testing (CT) v procesu testování.
Dílčí cíle diplomové práce jsou:
• Vytvoření literární rešerše týkající se problematiky automatického testování aplikací s architekturou mikroslužeb.
• Analýza hlavních metodik, jako jsou Devops, Agile-testování a CI/CD, které se v současné době používají pro automatizované testování v podnikatelském prostředí.
• Modelovaní procesu automatizovaného testování s použitím Continuous Testing pomocí BPMN.
• Hodnocení navrhovaných doporučení.
• Syntetizace výsledků práce, formulace přínosů a závěry práce.

Metodika

Metodika této práce je založena na studiu kvalitních odborných zdrojů informací a analýze praktických zkušeností. Praktická část práce se zaměří na zlepšení účinnosti procesu automatizovaného testování prostřednictvím praxe Continuous Testing. Pro modelování procesu bude použitá notace BPMN. Na základě generalizace a syntézy znalostí, které byly získány v procesu vytváření teoretické části, jakož i výsledků praktické části, budou vyvozeny obecné závěry.

Doporučený rozsah práce

60-80

Klíčová slova

Automatické testování, Continuous Testing, cloud native, architektura mikroslužeb, BPMN

Doporučené zdroje informací

ARUNDEL, John a Justin DOMINGUS. Cloud Native DevOps with Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud. United States of America: O'Reilly Media, 2019. ISBN 978-1-492-04076-7.

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.

RICHARDSON, Chris. Microservices Patterns with examples in Java. United States: Manning Publications, 2019. ISBN 978-5-4461-0996-8.

ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. Řízení kvality softwaru: průvodce testováním. Brno: Computer Press, 2013. ISBN 978-802-5138-168.

SVOZILOVÁ, A. *Zlepšování podnikových procesů*. Praha: Grada, 2011. ISBN 978-80-247-3938-0.

Předběžný termín obhajoby

2021/22 LS – PEF

Vedoucí práce

doc. Ing. Jan Tyrychtr, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 1. 11. 2021

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 23. 11. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 06. 09. 2022

Čestné prohlášení

Prohlašuji, že svou diplomovou práci „Automatické testování cloud native aplikací“ jsem vypracovala 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 autorka uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušila autorská práva třetích osob.

V Praze dne 27.03.2023 _____

Poděkování

Ráda bych touto cestou poděkovala svému vedoucímu práce panu doktoru Janu Tyrychtrovi za odborné vedení mé práce a jeho cenné rady.

Automatické testování cloud native aplikací

Abstrakt

Tématem této práce je Automatické testování cloud native aplikací. Hlavním cílem je snížit rizika a zlepšit proces vývoje aplikací, a to prostřednictvím metodiky průběžného testování. Pro dosažení tohoto cíle jsou v teoretické části analyzovány odborné literární zdroje spolu se zkušenostmi a zprávami předních světových IT společností, které v praxi využívají mikroservisní architekturu, cloudové technologie a nejefektivnější metodiky řízení IT projektů.

Praktická část práce je založena na projektu, ve kterém se vyvíjí cloud native mobilní aplikace. Tento projekt je ve stavu vývoje. V současné fázi již byla do procesu vývoje implementována metodika průběžné integrace a průběžného doručování, proces testování však dosud nebyl automatizován. Proto jsou v rámci praktické části vypracovány modely podnikových procesů, které popisují možnosti zlepšení procesu testování pomocí automatizace a využití metodiky průběžného testování. Procesy jsou modelovány podle IT architektury a IT stacku použitého v rámci tohoto projektu. Modely vytvořené v praktické části jsou předány zpět týmu jako návrh na zlepšení stávajících procesů testování. Modely BPMN jsou vytvořeny pro procesy zahrnující automatizované testování nebo procesy související s plánováním/realizací testování.

Klíčová slova: cloud native, architektura mikroslužeb, automatické testování, Continuous Testing, CI/CD, BPMN, pipeline

Automated testing of cloud native applications

Abstract

The topic of this diploma thesis is Automatic testing of cloud native applications. The main objective is to reduce the risks and improve the application development process using a continuous testing methodology. To achieve this goal, the theoretical part analyzes the literature sources and experiences and reports of the world's leading IT companies that use microservice architecture, cloud technologies and the most effective IT project management methodologies in practice.

The practical part of this diploma thesis is based on a project in which a cloud native mobile application is being developed. This project is in a state of development. At this phase, the continuous integration and continuous delivery methodology has already been implemented in the development process. However, the testing process has not been completely automated yet. Therefore, in the practical part, business process models are developed that describe the possibilities of improving the testing process through automation and using the continuous testing methodology. The processes are modelled according to the IT architecture and IT stack used in this project. The models created in the practical part are given back to the team as a suggestion to improve the existing testing processes. BPMN models are created for processes involving automated testing or processes related to test planning/execution.

Keywords: cloud native, microservices architecture, automated testing, Continuous Testing, CI/CD, BPMN, pipeline

Obsah

1 Úvod ... 10

2 Cíl práce a metodika ... 11

2.1 Cíl práce ... 11

2.2 Metodika ... 11

2.2.1 BPMN ... 11

3 Teoretická východiska ... 14

3.1 Cloud native ... 14

3.1.1 Microservice architecture ... 16

3.1.2 Výhody modelu mikroslužeb ... 18

3.1.3 Výzvy související s používáním mikroslužeb ... 19

3.1.4 Cloud native technologie ... 23

3.2 Testování ... 24

3.2.1 Proces řízení kvality projektu ... 26

3.2.2 Artefakty testování ... 28

3.2.3 Úrovně testování ... 29

3.2.4 Typy testování ... 31

3.2.5 Porovnání typů prostředí ... 32

3.2.6 Automatizace procesu testování ... 33

3.2.7 Zvláštností testování aplikací mikroslužeb ... 35

3.3 Metodiky testování, vývoje a provozu ... 37

3.3.1 Tradiční metodiky vývoje softwaru ... 37

3.3.2 Agile ... 39

3.3.3 DevOps ... 43

3.4 CI/CD a CT ... 46

3.4.1 Průběžná integrace (Continuous Integration) ... 46

3.4.2 Průběžné doručování/nasazení (Continuous Delivery/Deployment) ... 47

3.4.3 Průběžné testování (Continuous Testing) ... 49

4 Vlastní práce ... 52

4.1 Charakteristika společnosti Adastra, s. r. o. ... 52

4.2 Analýza současného stavu vybraného projektu ... 53

4.2.1 Členové projektového týmu ... 53

4.2.2 Požadavky na testování a kvalitu ... 56

4.2.3 Infrastruktura testovacího prostředí ... 57

4.2.4 Proces nasazení ... 58

4.3 Analýza implementace automatizovaného testování v projektu ... 59

4.3.1 Proces vytváření automatického testu ... 59

4.3.2 Automatizované testování v rámci CT 61

Výsledky a diskuse 62

Závěr 64

5 Seznam použitých zdrojů 65

6 Seznam obrázků, tabulek a zkratek 68

6.1 Seznam obrázků 68

6.2 Seznam tabulek 68

6.3 Seznam použitých zkratek 69

1 Úvod

Cloudové technologie jsou již řadu let hlavním celosvětovým technologickým trendem. Cloud computing umožňuje podnikatelské praxi využívat výhod, jako je škálovatelnost, odolnost, pružnost apod. Termín cloud native je v posledních několika letech široce diskutován globálními lídry v oblasti IT. Existují různé názory na to, jak definovat pojem cloud native a jaké aplikace lze vlastně označit za cloud native. Tento přístup vychází nejen z umístění aplikace v cloudu, ale také z dalších charakteristik, mezi nimiž hraje důležitou roli budování aplikace pomocí architektury mikroslužeb. Efektivitu budování aplikací na bázi mikroslužeb dokládá praxe velkých společností, jmenovitě Netflix, AWS, Spotify, Uber a mnoha dalších. [1]

Každá společnost zabývající se vývojem softwaru neustále hledá nová řešení, metodiky a vzory, které mohou zvýšit efektivitu procesu vývoje softwaru. V roce 2001 vznikl manifest agilní metodiky, který posunul proces vývoje na novou úroveň. Používání agilních metodik je dnes aktuálním tématem. Metodiky se aktivně mění. V současné době se aktivně diskutuje o používání metodik, jako jsou DevOps, CI/CD a průběžné testování. Vzhledem k velkému množství typů existujících metodik je nutné vybrat tu správnou, která pomůže zlepšit proces vývoje, testování a podpory vyvíjeného produktu v praxi. [7]

Kromě výše uvedených aktuálních agend stojí svět IT před úkolem snižovat náklady na vývoj softwaru a neustále zlepšovat kvality vytvářeného produktu. K řešení tohoto problému vede více cest, přičemž jednou z nejpodceňovanějších je včasnější zapojení testování do procesu vývoje a jeho automatizace. Toto řešení nejenže snižuje počet chyb způsobených lidskými omyly, ale také pomáhá identifikovat chyby v počátečních fázích projektu a snižuje náklady na manuální testování. Samotná schopnost organizovat a provádět efektivní testování může navíc významně snížit náklady v celé společnosti, a to jak úsporou času vývojářů, tak vytvořením nasazovacího pipeline, kde mohou vývojáři rychle provádět změny v kódu s minimálním rizikem narušení aplikace v různých prostředích včetně produkčního. Na základě údajů z průzkumů v roce 2022 využívalo automatizované testování ve vývojovém procesu pouze přibližně 44 % společností na globálním trhu. [27]

2 Cíl práce a metodika

2.1 Cíl práce

Diplomová práce je tematicky zaměřena na problematiku automatického testování aplikací s architekturou mikroslužeb. Hlavním cílem je snížit rizika a zvýšit účinnost procesu vývoje aplikace pomocí použití metodiky Continuous Testing (CT) v procesu testování.

Dílčí cíle diplomové práce jsou:

- vytvoření literární rešerše týkající se problematiky automatického testování aplikací s architekturou mikroslužeb,
- analýza hlavních metodik, jako jsou DevOps, testování agile a CI/CD, které se v současné době používají pro automatizované testování v podnikatelském prostředí,
- modelování procesu automatizovaného testování s použitím Continuous Testing pomocí BPMN,
- hodnocení navrhovaných doporučení,
- syntetizace výsledků práce, formulace přínosů a závěry práce.

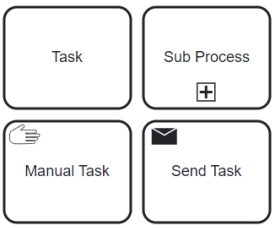
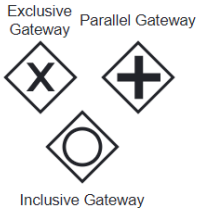
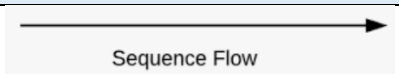
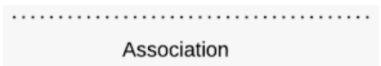
2.2 Metodika

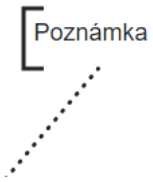
Metodika této práce je založena na studiu kvalitních odborných zdrojů informací a analýze praktických zkušeností. Praktická část práce se zaměřuje na zlepšení účinnosti procesu automatizovaného testování prostřednictvím praxe Continuous Testing. Pro modelování procesu bude použita notace BPMN. Na základě generalizace a syntézy znalostí, které byly získány v procesu vytváření teoretické části, jakož i výsledků praktické části, budou vyvozeny obecné závěry.

2.2.1 BPMN

Business Process Model and Notation (BPMN) je notace používaná k reprezentaci podnikových procesů a byla uznána de facto jako mezinárodní standard. BPMN bylo vyvinuto konsorciem dodavatelů modelování procesů v roce 2003 a již v roce 2006 bylo vydáno jako standard Object Management Group. [47]

Hlavní výhodou BPMN je přístupný popis obchodních procesů. Výsledkem modelování procesu pomocí notace je Business Process Diagram (BPD). Následující tabulka uvádí klíčové komponenty BPMN, které se používají k sestavení BPD:

Tokové objekty		
Událost	Události přímo ovlivňují proces. Pro spuštění události je nutný důvod. Proces začíná a končí událostí a existují také průběžné události.	
Aktivita	Jedná se o obecný grafický prvek znázorňující nějakou činnost.	
Brána	Řídí divergenci a konvergenci toku sekvencí. Vnitřní značky udávají typ chování brány.	
Spojovací objekty		
Sekvenční tok	Definuje pořadí úloh v procesu.	
Tok zpráv	Slouží k zobrazení toku zpráv mezi jednotlivými účastníky procesu.	
Asociace	Slouží k propojení objektu s daty, textem nebo jiným artefaktem.	
Plavecké dráhy		

Bazén	Zohledňuje účastníka procesu. Bazén může, ale nemusí odkazovat na proces. Bazén, který neobsahuje proces, se nazývá „black box“. K bazénu zobrazenému jako black box nelze přiřadit žádné sekvenční toky, nicméně toky zpráv se mohou připojit k hranicím takového bazénu.	
Dráha	Dílčí sekce v rámci bazénu. Dráhy slouží k uspořádání a kategorizaci činností.	
	Artefakty	
Datový objekt	Tyto objekty slouží k zobrazení vytvoření dat nebo potřeby získat určitá data.	
Seskupení	Slouží k dokumentaci, nemá vliv na průběh sekvence.	
Poznámka	Poskytnutí dalších textových informací pro uživatele diagramu BPMN.	

Tabulka 1 - Klíčové komponenty BPMN, které se používají k vytvoření modelu podnikového procesu [47]

K modelování procesů popsaných v praktické části práce byl použit nástroj Camunda Modeler 7.18. Camunda je open-source platforma, která se používá k automatizaci podnikových procesů. K vytváření a úpravě obrázků byl použit program draw.io. [48]

3 Teoretická východiska

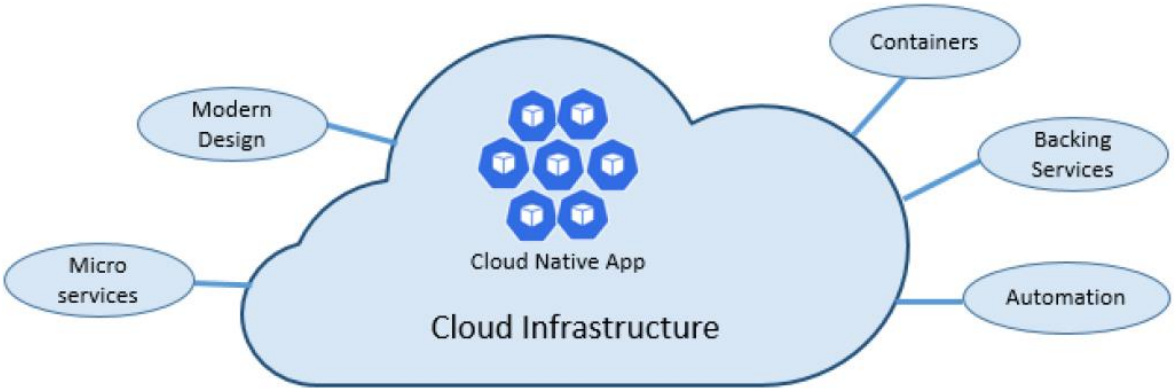
3.1 Cloud native

Termín cloud native se stále častěji používá k tvorbě moderních aplikací a webových služeb. Cloud native aplikace využívají výhod cloudových technologií a architektury mikroslužeb. Tyto aplikace jsou často založeny na softwaru s otevřeným zdrojovým kódem. [1]

Je třeba poznamenat, že aplikace, která prostě běží v cloudu, není cloud native aplikace. Důležité je, jak aplikace funguje, ne kde se zpracovává. Namísto monolitického nasazení cloud native aplikace se využívají distribuované mikroslužby. [1]

Pojem mikroslužby a architektura mikroslužeb bude podrobněji popsán v další kapitole.

Obrázek níže ukazuje hlavní pilíře cloud native [18]:



Obrázek 1- Základy cloud native [18]

Obecné vlastnosti cloud native aplikací [1]:

Vlastnost	Popis
Automatizovaná	Aby bylo možné automatizovat procesy nasazení a správy, musí aplikace odpovídat standardům, rozhraním a formátům. Kubernetes pomáhá vývojářům vyrovnat se s tímto úkolem.

Přenosná a flexibilní	Aplikace, které jsou vyvíjeny v kontejnerech, jsou obvykle odděleny od fyzických zdrojů, takže tyto služby lze přesouvat mezi výpočetními uzly.
Stabilní a škálovatelná	Existuje mnoho důvodů, proč monolitická aplikace přestane fungovat. Příklady zahrnují chyby aplikací, selhání fyzického serveru nebo selhání sítě. Díky tomu, že je cloud native aplikace postavena na volně vázaných službách, je odolná, spravovatelná a sledovatelná.
Dynamická	Použití kontejnerových orchestrátorů pomáhá naplánovat co nejefektivnější využití zdrojů. Kubernetes může například provozovat více kontejnerových instancí, což umožňuje postupnou aktualizaci služeb bez přerušení dostupnosti aplikace.
Sledovaná	Jedním z klíčových požadavků na cloud native aplikace je pozorovatelnost. Vzhledem ke zvláštnosti architektury je obtížné je ovládat a ladit. Následující nástroje pomáhají technikům identifikovat chyby aplikací: monitorování, protokolování, sledování a metriky.
Distribuovaná	Cloud native je přístup k vytváření a spouštění aplikací, který využívá výhod distribuované a decentralizované povahy cloudu.

Tabulka 2 - Obecné vlastnosti cloud native aplikací [1]

V roce 2015 byla vytvořena Cloud Native Computing Foundation (CNCF), která je součástí Linux Foundation. CNCF definuje své vlastní poslání takto [2]:

„Posláním Cloud Native Computing Foundation je zajistit, aby byl Cloud native computing všudypřítomný.”

Definice cloud native podle CNCF zní takto [2]:

„Cloud native technologie umožňují organizacím vytvářet a provozovat škálovatelné aplikace v moderních dynamických prostředích, jako jsou veřejné, soukromé a hybridní cloudy. Příkladem tohoto přístupu jsou kontejnery, sítě služeb, mikroslužby, neměnná infrastruktura a deklarativní rozhraní API. Tyto techniky umožňují volně spojené systémy, které jsou odolné, ovladatelné a pozorovatelné. V kombinaci s robustní automatizací umožňují inženýrům provádět často a předvídatelně vysoce účinné změny s minimální dřinou."

Na vysoké úrovni znamená cloud native přizpůsobení se mnoha novým možnostem – ale velmi odlišné sadě architektonických omezení, které cloud nabízí ve srovnání s tradiční monolitickou architekturou. [18]

Prvky vysoké úrovně, které je potřeba vzít v úvahu při vytváření cloud native aplikace [41]:

- funkční požadavky systému (co by měl dělat),
- nefunkční požadavky (jak by to mělo fungovat),
- omezení (co nelze změnit).

Zatímco funkční aspekty se příliš nemění, cloud nabízí a někdy vyžaduje velmi odlišné způsoby, jak splnit nefunkční požadavky, a klade velmi odlišná architektonická omezení. Pokud se architektům nepodaří přizpůsobit svůj přístup těmto různým omezením, systémy, které navrhují, jsou často křehké, drahé a obtížně se udržují. Na druhou stranu dobře navržený cloudový nativní systém by měl být z velké části samoopravný, nákladově efektivní a snadno aktualizovatelný a udržovaný prostřednictvím kontinuální integrace/průběžného doručování (CI/CD). [41]

3.1.1 Microservice architecture

Vznik cloudu výrazně zvýšil flexibilitu softwaru a škálovatelnost napříč odvětvími. Cloud native aplikace běžící na mikroslužbách jsou nyní pro nově vyvíjené aplikace zásadní. Mikroslužby nám dávají výrazně větší svobodu ovlivňovat a činit různá rozhodnutí, což nám umožňuje rychle reagovat na nevyhnutelné změny ovlivňující všechny členy vývojového týmu. [3]

Definice pojmu mikroslužby podle zdroje [3]:

„Mikroslužby jsou malé, samostatné, spolupracující služby."

Mikroslužby používají více samostatných aplikačních služeb, které interagují přes rozhraní API. Cloud native aplikace jsou vytvářeny pomocí mikroslužeb, aby zůstaly agilní a inovativní. Architektury mikroslužeb umožňují mimo jiné iterativní vývoj, lepší izolaci chyb a zvýšenou škálovatelnost. Zavedení mikroslužeb může být náročné bez náležité přípravy a interního nákupu. [44]

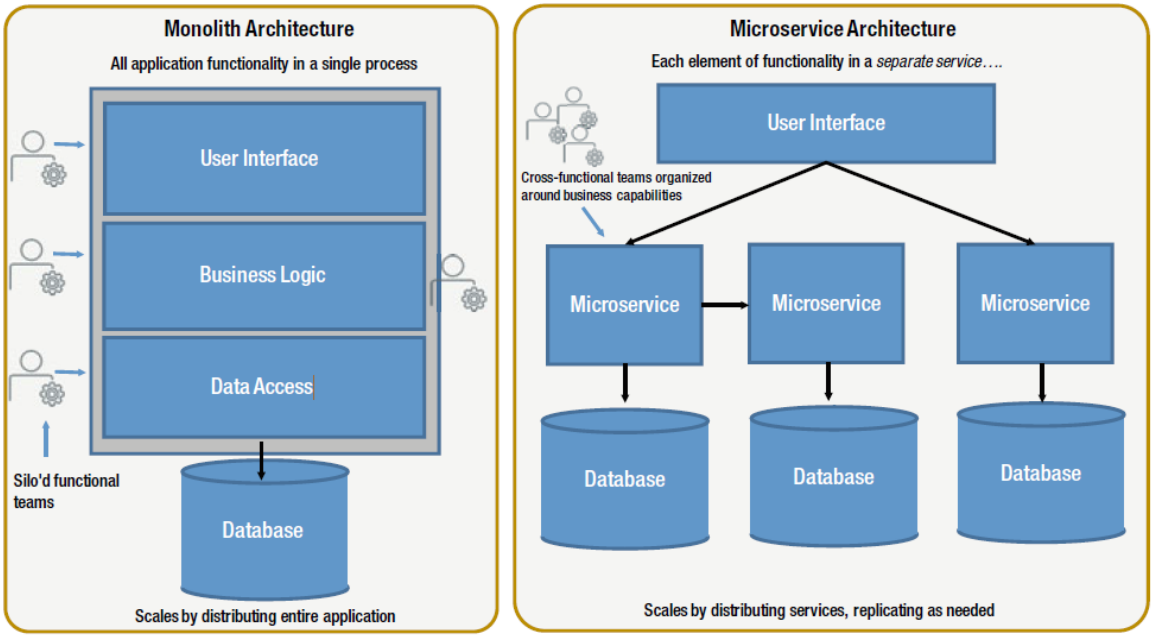
Níže uvedený obrázek ukazuje oborové giganty, kteří aktivně využívají mikroslužby v praxi:

Obrázek 2 - Využití mikroslužeb v praxi [35]

Obrázek číslo 3 ukazuje rozdíly mezi monolitickou a mikroservisní architekturou. Tradiční monolitická architektura se v mnoha případech stala obtížně udržovatelnou a škálovatelnou, protože moderní aplikace se zvětšují a jsou složitější. Myšlenka mikroslužeb jako řešení tohoto problému spočívá v tom, že aplikace lze lépe udržovat, když jsou rozděleny na jednotlivé softwarové komponenty odpovědné za definované, nezávislé úkoly. [3]

Každá mikroslužba může být nasazena a aktualizována nezávisle, aniž by to mělo dopad na jiné části aplikace. Ke konkrétním mikroslužbám mohou být přiřazeny samostatné

vývojové týmy, aby se vyvinula specializace na optimalizaci dané konkrétní služby. Jedinečné zaměření umožňuje specializované odborné znalosti, kratší životní cykly vývoje, rychlejší dobu odezvy na problém a schopnost pracovat v prostředí průběžné integrace a průběžného doručování (CI/CD). [24]



Obrázek 3 - Porovnání monolitické architektury a architektury mikroslužeb [7]

3.1.2 Výhody modelu mikroslužeb

Tato tabulka popisuje výhody spojené s používáním architektury mikroslužeb v praxis [3] [44]

Výhody	Popis
Iterativní vývoj	Díky izolované povaze mikroslužeb mohou vývojáři vytvářet a nasazovat verze nových služeb a aktualizací s minimálním životaschopným produktem (MVP), poté iterovat za účelem vylepšení chyb a dalších problémů a v reakci na zpětnou vazbu koncových uživatelů.
Autonomie vývojáře	Pomocí mikroslužeb může více vývojářských týmů pracovat současně na různých komponentách aplikace, aniž by si vzájemně ovlivňovaly výstup. To znamená, že si také mohou vybrat nástroje

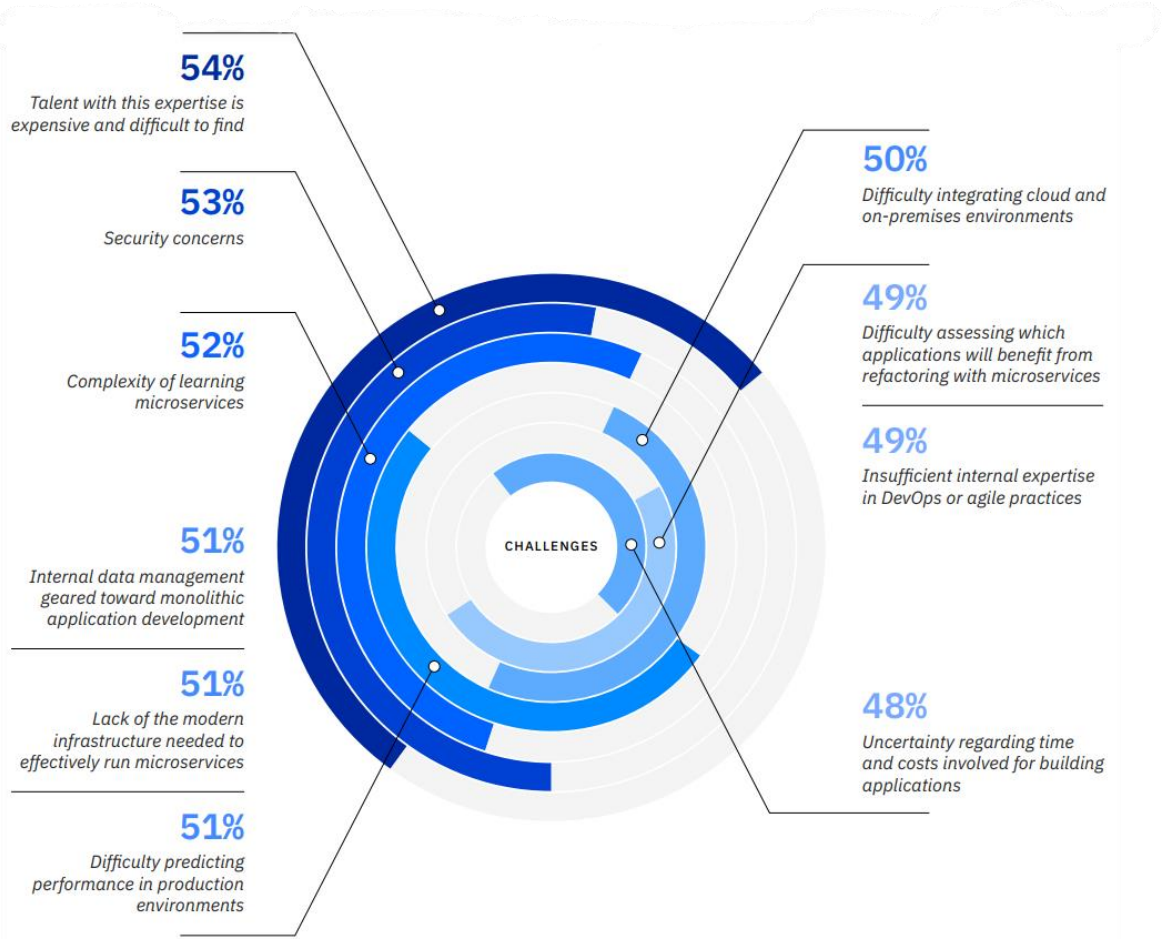
	a přístupy, které nejlépe fungují pro jejich projekt, aniž by se museli obávat, jak jejich práce ovlivní ostatní součásti aplikace.
Izolace chyb	Jednou z nejcennějších výhod mikroslužeb je izolace chyb. To znamená, že chyba nebo porucha v jedné službě nebude mít nepříznivé účinky na zbytek aplikace. Pokud komponenta selže, vývojáři mohou jednoduše použít jinou službu a aplikace může běžet dál.
Méně narušení	Protože jsou mikroslužby průběžně sestavovány, nasazovány a aktualizovány samostatně, ke změnám dochází postupně (na rozdíl od jednoho velkého vydání s mnoha změnami). To vytváří hladší zážitek s menším rušením pro koncového uživatele.
Škálovatelnost	Protože každá mikroslužba běží autonomně, lze je škálovat samostatně, aniž by bylo nutné upravovat celou aplikaci. Když se zvýší poptávka po konkrétní mikroslužbě, lze ji upgradovat a podle potřeby jí mohou být přiděleny zdroje. Škálováním pouze služeb, které to potřebují (a to pouze v případě potřeby), mohou společnosti ušetřit na nákladech na cloudový server.
Snadnost náboru	Model mikroslužeb je dnes atraktivním pracovním prostředím pro vývojáře, zejména mladší, novější vývojáře obeznámené s DevOps a méně spojené s jinými, tradičnějšími způsoby provozu. Zkušení vývojáři se také často těší vyhlídce na menší třenice mezi týmy a možnost pracovat rychle a nezávisle na službách, za které jsou konkrétně zodpovědní.

Tabulka 3 - Výhody modelu mikroslužeb [3] [44]

3.1.3 Výzvy související s používáním mikroslužeb

V roce 2021 provedla společnost IBM průzkum, kterého se zúčastnilo více než 1200 vývojářů a IT lídrů. 87 % respondentů souhlasilo s tím, že výhody mikroslužeb převažují nad riziky spojenými s jejich využíváním. Tento průzkum nám však umožnil identifikovat řadu problémů, na které účastníci průzkumu při zavádění mikroslužeb narazili. [45]

Dále diagram ukazuje identifikované problémy s uvedením procenta, kolik respondentů tento problém naznačilo:



Obrázek 4 - Potíže spojené s používáním architektury mikroslužeb [45]

Problém	Popis
Odborník je drahý a těžko se hledá	Mikroslužby jsou vysoce distribuované systémy. Je nutno ověřit, zda tým má dostatečné znalosti a zkušenosti potřebné k úspěchu. Konkurenční a rychlá povaha vývoje softwaru může vést týmy k přijetí řešení, která mohou nakonec způsobit více problémů, než odstranit. Dokud vývojáři nepochopí, jak překonat problémy a zmírnit rizika spojená s mikroslužbami, může se proces vývoje výrazně zpomalit.
Obavy o bezpečnost	Dekompozice aplikací do mikroslužeb zvyšuje prostor pro možné kybernetické útoky, protože se objevují nové vstupní

	brány a závislosti mezi službami. Z tohoto důvodu vyžaduje zabezpečení mikroslužeb netriviální řešení.
Složitost osvojení mikroslužeb	Aplikace mikroslužeb má více pohyblivých částí než ekvivalentní monolitická aplikace. Každá služba je jednodušší, ale systém je jako celek složitější.
Správa interních dat zaměřená na vývoj monolitických aplikací	V architektuře mikroslužeb je úplné minimum centralizované správy, takže kontrola prostředí je obtížná. Zatímco jedna sada procedur platí pro monolitické prostředí, prostředí mikroslužeb bude pro každé prostředí potřebovat jiné.
Nedostatek moderní infrastruktury potřebné k efektivnímu provozování mikroslužeb	Psaní malých služeb, které se spoléhají na další závislé služby, vyžaduje jiný přístup než psaní tradičních monolitických nebo vrstvených aplikací. Stávající nástroje nejsou vždycky navržené pro práci se závislostmi služeb. Refaktoring přes hranice služeb může být obtížný. Je také náročné testování závislostí služeb, zejména v případě, že je aplikace vyvíjena rychle.
Obtížnost předpovídání výkonu v produkčním prostředí	Předpovídání výkonu aplikací, které mají architekturu mikroslužeb, je obtížný úkol. Všechny komponenty mají své vlastní charakteristiky, závislosti a prostředí. Pro správnou predikci je možné aplikovat přístup, ve kterém je nutné zohlednit intenzitu a rozložení zatížení v topologii aplikace.
Potíže s integrací cloudových a místních prostředí	V závislosti na geografii a právních předpisech může být vyžadováno, aby data zůstala v rámci místních datových center. Pro řešení tohoto problému by mohlo být využito nasazení cloud native služeb v hybridním a multicloudovém prostředí. Hybridní strategie plně podporuje zákaznické operace, které vylučují použití cloudu pro některé pracovní zátěže pro vysoce regulovaná odvětví.

Potíže s posouzením, které aplikace budou mít prospěch z refaktoringu pomocí mikroslužeb	Tým musí provést náležitou péči a pochopit, co funguje pro jejich konkrétní případy použití. Pro menší společnosti může být začátek s monolitickou aplikací jednodušší, rychlejší a levnější, a pokud produkt není příliš vyzrálý, může být stále ve vhodnou dobu migrován na mikroslužby. Obrovské společnosti s miliony uživatelů jsou zřejmými příklady nejlepšího případu použití pro mikroslužby, protože potřebují zajistit dostupnost a škálovatelnost, kterou může poskytnout přidaná modularita.
Nedostatečná interní odbornost v oblasti DevOps nebo agilních praktik	V DevOps je nejvyšší prioritou poskytovat vysoce hodnotné funkce v krátkých časových obdobích prostřednictvím spolupráce mezi týmy. Inženýři musí pravidelně mluvit s interními manažerskými týmy zapojenými do procesu DevOps a musejí si být vědomi cílů, plánu, blokovacích problémů a dalších oblastí projektu. Inženýři DevOps by měli být týmoví hráči a podporovat své kolegy během sprintů nebo softwarových iterací. DevOps by také měli umět mentorovat a radit členům týmu, jak nejlépe doručit kód, jaké nástroje použít při kódování a jak testovat nejnovější funkce.
Nejistota ohledně času a nákladů spojených s vývojem aplikací	I když výhody používání mikroslužeb mohou být značné, je důležité pečlivě zvážit náklady na vývoj mikroslužeb. Náklady a čas na vývoj aplikace mikroslužeb se často dramaticky zvyšují kvůli zavádějícím očekáváním snadného vývoje nebo špatné správy týmu.

Tabulka 4 - Potíže spojené s používáním architektury mikroslužeb [34] [36] [37] [38] [45]

I když jsou výzvy spojené s přijetím přístupu mikroslužeb reálné, mnohé obavy – jako nedostatek zkušených talentů, nejistota ohledně bezpečnostních problémů a zmatek ohledně toho, které aplikace jsou nejlepšími cíli pro přechod na mikroslužby – lze zmírnit přivedením správných talentů. [45]

Další obavy, jako je získání závazku modernizovat infrastrukturu a potřeba vyvinout starší přístupy a procesy navržené pro monolitické aplikace, mohou vyžadovat interní

posuny, které lze usnadnit vybudováním silného obchodního případu. Složitost je zmírněna pomocí technologií kontejnerů, jako je Kubernetes, spolu se sítí služeb, která poskytuje konzistentní strukturu, jež usnadňuje správu komunikace a monitorování zabezpečení mezi mnoha službami, které zahrnují aplikace vyvinuté pomocí mikroslužeb, a napříč nimi. [45]

3.1.4 Cloud native technologie

Cloud native technologie, označované také jako cloud native zásobník, jsou technologie používané k vytváření cloud native aplikací. Tyto technologie umožňují organizacím vytvářet a provozovat škálovatelné aplikace v moderních a dynamických prostředích, jako jsou veřejné, soukromé a hybridní cloudy, a zároveň naplno využívat výhody cloud computingu. Jsou od základu navrženy tak, aby využívaly schopností cloud computing. Kontejnery, servisní sítě, mikroslužby a neměnná infrastruktura jsou příkladem tohoto přístupu. [2]

Následující tabulka ukazuje kategorie technologií, které jsou nutné k vytvoření cloud native aplikace:

<table>
<tr><td>App Definition & Development</td><td colspan="2">Database</td><td colspan="2">Streaming & Messaging</td><td colspan="2">Application Definition & Image Build</td><td colspan="2">Continuous Integration & Delivery</td></tr>
<tr><td>Orchestration & Management</td><td>Scheduling & Orchestration</td><td colspan="2">Coordination & Service Discovery</td><td>Remote Procedure Call</td><td>Service Proxy</td><td>API Gateway</td><td>Service Mesh</td><td></td></tr>
<tr><td>Runtime</td><td colspan="3">Cloud Native Storage</td><td colspan="3">Container Runtime</td><td colspan="2">Cloud Native Network</td></tr>
</table>

Provisioning	Automation & Configuration	Container Registry	Security & Compliance	Key Management	
Platform	Certified Kubernetes - Distribution	Certified Kubernetes Hosted	Certified Kubernetes - Installer	PaaS/Container Service	
Observability & Analysis	Monitoring	Logging	Tracing	Chaos Engineering	Continuous Optimization
Serverless	Tools	Framework	Hosted Platform	Installable Platform	Security
Special	Kubernetes Certified Service Provider	Kubernetes Training Partner	Certified CNFs		

Tabulka 5 - Cloud native technologie [32]

Cloud native stack řeší na vysoké úrovni jednu hlavní skupinu nesnází: nevýhody tradičních provozních modelů IT. Mezi problémy patří mimo jiné potíže s vytvářením škálovatelných a samoopravných aplikací, odolných chybám a také neefektivní využití zdrojů. [2]

Jako skupina umožňují cloud native technologie volně propojené systémy, které jsou odolné, spravovatelné a pozorovatelné. V kombinaci s robustní automatizací umožňují inženýrům provádět často a předvídatelně vysoce účinné změny s minimální dřinou. [2]

3.2 Testování

Testování je mnohostranný koncept, jehož účel se v čase mění. Testování bylo původně vytvořeno jako způsob, jak prokázat, že program funguje správně. Již v polovině

60. let minulého století byl účel testování změněn na hledání chyb v programu. Později testování bylo použito k měření kvality. [4]

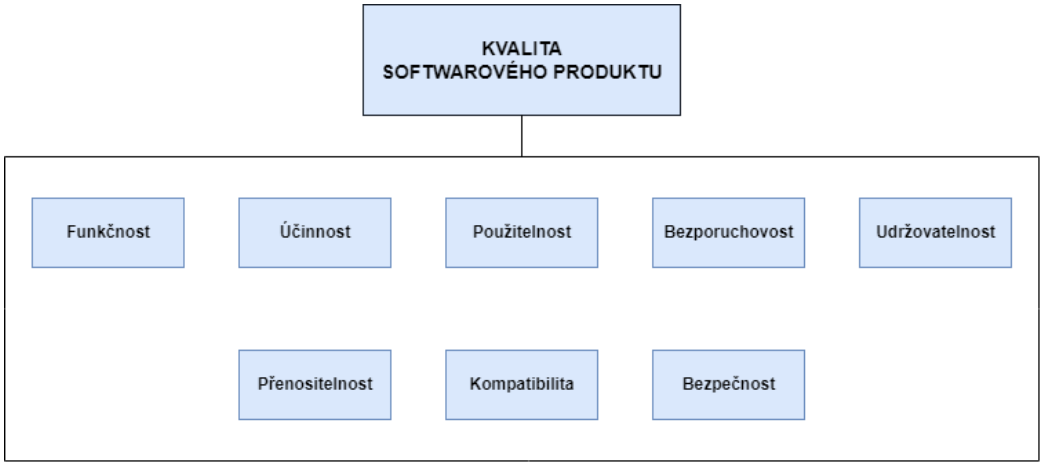
Hlavním cílem firem z hlediska ekonomiky je dosahovat zisku. IT podniky nejsou výjimkou. K dosažení tohoto cíle musejí firmy vytvořit konkurenční výhody. Kvalita softwaru, který vytvářejí, je klíčovou výhodou, která pomáhá IT firmám zvýšit prodej a tím maximalizovat jejich zisky. Jak je již zřejmé, kvalita hraje důležitou roli v procesu vývoje softwaru. Právě z tohoto důvodu byly v průběhu posledních 30 let vytvořeny testovací standardy a nástroje pro efektivní řízení procesu testování. Kvalita používaného softwaru je jednou z hlavních konkurenčních výhod a tím i atributem úspěšné existence mnoha ekonomických subjektů. Z toho důvodu bylo během posledních tří desetiletí vytvořeno mnoho různých standardů zabývajících se dosažením kvality, nástrojů pro efektivnější řízení testování, začaly být pořádány mezinárodní specializované konference pro pracovníky v oblasti testování a v neposlední řadě také vyšla řada knih. [4]

Jak můžeme vidět, podoba testování softwaru se pochopitelně v čase mění a stejně tak i jeho chápání, ovšem bez ohledu na to zůstává stále nepostradatelnou součástí životního cyklu vývoje softwaru. Testování nám umožňuje měřit a zlepšovat kvalitu vyvíjených systémů, které jsou téměř nedílnou součástí našeho života. [4]

Definice pojmu kvalita systému podle ISO/IEC 25010 [5]:

„Kvalita systému je míra, do jaké vytvořený produkt uspokojuje stanovené a implikované potřeby různých zúčastněných stran, a tím poskytuje produktu hodnotu. Potřeby těchto zúčastněných stran (funkčnost, výkon, bezpečnost, udržovatelnost atd.) jsou přesně tím, co je zastoupeno v modelu kvality, který rozděluje kvalitu produktu na vlastnosti a dílčí charakteristiky.”

Následující diagram ukazuje model kvality produktu podle normy ISO/IEC 25010:



Obrázek 5 - Model kvality produktu podle normy ISO/IEC 25010 [5]

Model kvality určuje, které kvalitativní charakteristiky budou brány v úvahu při hodnocení vyvinutého produktu. V tomto standardu je každá charakteristika rozdělena na podrobnější podcharakteristiky, což pomáhá zvýraznit potřebné ukazatele pro testování. [5]

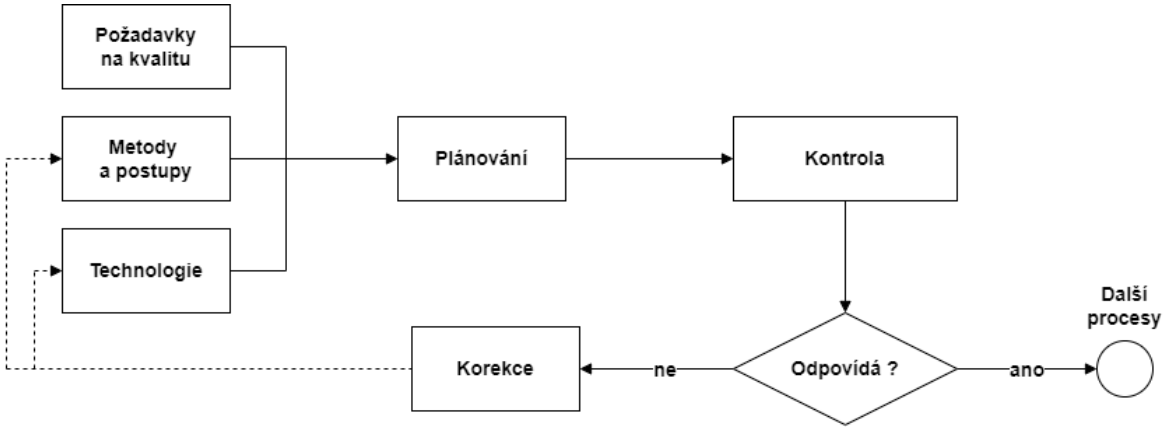
3.2.1 Proces řízení kvality projektu

Řízení kvality projektu je nezbytný proces, který přímo ovlivňuje efektivitu řízení celého projektu.

Tento proces zahrnuje následující požadované položky [6]:

- detailní plánování procesů a metod k dosažení požadovaných kvalitativních znaků, které jsou dohodnuty se zákazníkem,
- testování a audit kvality,
- evidence a prioritizace defektů,
- aktivní komunikace v týmu s cílem zlepšit kvalitu produktu včas, v těch místech, kde kvalita neodpovídá požadované úrovni.

Schématické vyobrazení procesu řízení kvality projektu vypadá takto:

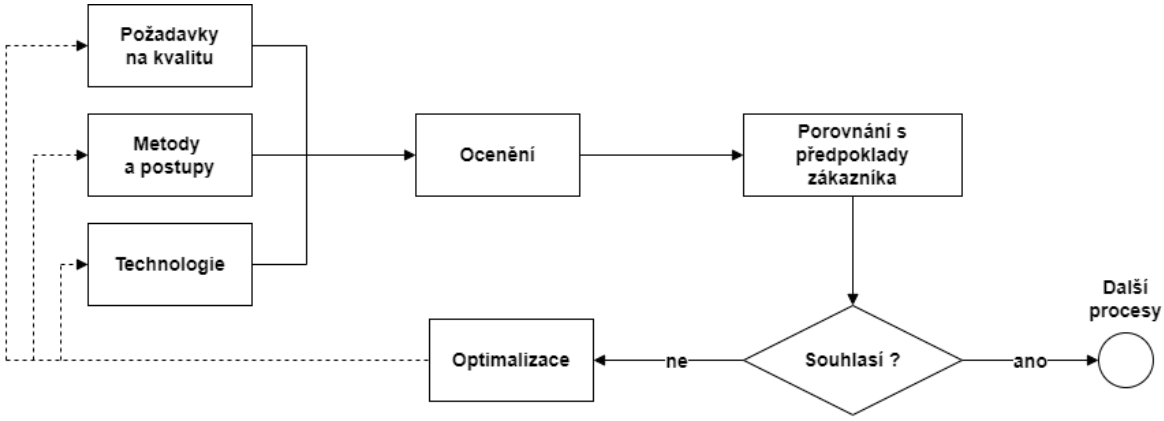


Obrázek 6 - Diagram procesu řízení kvality [6]

Hlavním cílem procesu řízení kvality projektu dle zdroje [6] je:

„Cílem procesu řízení kvality je zajištění kvalitativních požadavků na výstupy projektu, efektivita a účinnost v průběhu zpracování projektu."

Následující schéma ukazuje koncept kvality projektu a dopad na jeho ceny [6]:



Obrázek 7 - Koncept kvality projektu dopady na jeho cenu [6]

Tento koncept vyžaduje zvýšenou pozornost projektového manažera a zahrnuje následující body [6]:

- je nutné, aby požadavky zákazníka, které určují kvalitativní charakteristiky, odpovídaly předmětu projektu,
- společné plánování procesů (technologických a administrativních) s vývojovým týmem, k vytvoření efektivní komunikace v týmu, která je nezbytná pro dosažení cílů kvality,

- neustálá analýza a zlepšování interních procesů, aby se předešlo různým možným odchylkám od požadavků na kvalitu.

3.2.2 Artefakty testování

V procesu testování lze použít různé artefakty testování. Vymezení pojmu artefakty testování podle zdroje [14]:

„Artefakty testování jsou soubor dokumentů a doplňkových materiálů, které se používají v životním cyklu testování. Artefakty může využívat nejen tým QA, ale i další zainteresované strany, přesněji řečeno vývojový tým, klient nebo stakeholdeři."

Nejběžnější artefakty testování jsou [13]:

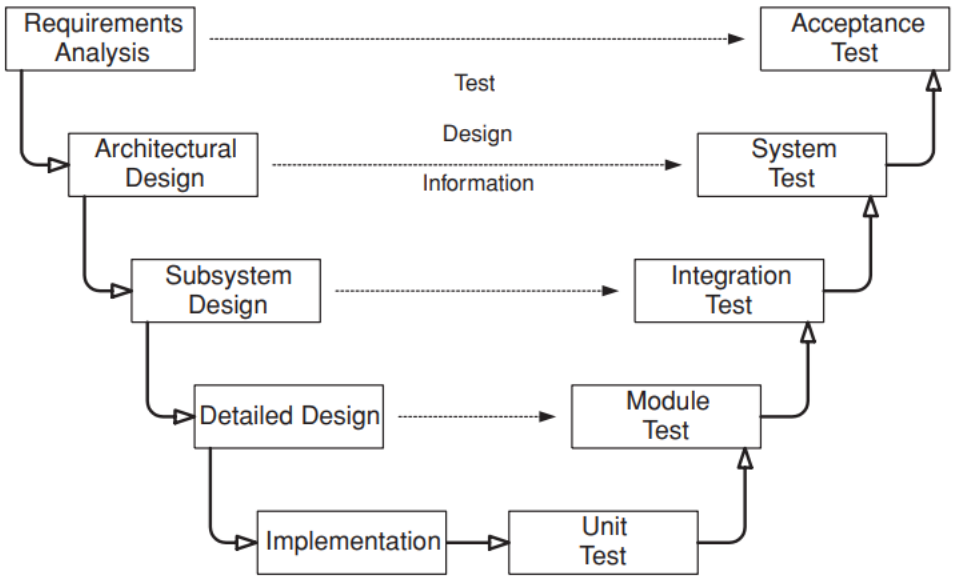
Test Plan	Popisuje, jakými prostředky a v jakých termínech bude dosaženo stanovených cílů testování.
Test Design Specification	Definuje podmínky testu, pokrytí testu, přístup, který se má použít při testování, a identifikuje sady testů.
Test Case Specification	Definuje sadu testovacích případů, konkrétně co je cílem, jaké vstupy budou použity a předpoklady, kroky a očekávaný výsledek.
Test Procedure Specification	Určuje pořadí, ve kterém má být konkrétní test spuštěn.
Test Log	Záznam podrobností o provedení testu, který je veden v chronologickém pořadí.
Test Incident Report	Dokument, který zachycuje jakoukoli událost, ke které dojde během testování a která neodpovídá očekávanému výsledku.
Test Summary Report	Dokument, který shrnuje výsledky testování.

Tabulka 6 - Přehled testovacích artefaktů podle standardu IEEE-829 [13]

Uvažované artefakty jsou jistě důležité pro testování, ale je třeba si uvědomit, že použití testovacích artefaktů je specifické pro různé projekty. Proto lze na různých projektech použít rozdílné testovací artefakty. [14]

3.2.3 Úrovně testování

Aby bylo možné provádět lepší testování, příprava testu by měla vycházet z různých dat, jmenovitě: požadavků a specifikací, artefaktů návrhu a zdrojového kódu. Různé úrovně testování odpovídají určité fázi vývoje softwaru. Následující diagram ukazuje úrovně testování z pohledu V-modelu. [15]



Obrázek 8 - Aktivity v oblasti vývoje softwaru a úrovně testování (V-model) [15]

Úrovně testů hrají důležitou roli v procesu testování jako celku. Testy je nutné připravovat na základě každé fáze vývoje, protože to může pomoci identifikovat defekty v rozhodnutích o návrhu, a to ještě před okamžikem, kdy tyto chyby budou implementovány do kódu. Včasné odhalení defektů je nejlepším prostředkem ke snížení nákladů na projekt. [15]

Následující tabulka poskytuje podrobný popis úrovní testování:

Úrovně testování	Popis
Jednotkové testování	Posouzení softwaru s ohledem na implementaci. Tento typ je určen pro testování jednotlivých softwarových komponent

	vytvořených ve fázi implementace. Tato úroveň je „nejnižší" a obvykle ji provádí programátor. Testování jednotek je často automatizované, protože tyto skripty lze snadno zabalit spolu s odpovídajícím kódem, například pomocí JUnit.
Testování modulů	Posouzení softwaru s ohledem na detailní návrh. Testování modulů je navrženo tak, aby vyhodnocovalo jednotlivé moduly izolovaně, včetně toho, jak moduly vzájemně spolupracují, a jejich související datové struktury. Stejně jako předchozí úroveň, testování jednotek často provádí vývojář. Programový modul (procedura) je jeden nebo více souvisejících programových příkazů s názvem, který ostatní části softwaru používají k jeho volání.
Integrační testování	Posouzení softwaru s ohledem na návrh subsystému. Integrační testování je navrženo tak, aby vyhodnotilo, zda moduly správně spolupracují. Integrační testování by mělo předpokládat, že moduly fungují správně. Tento typ testování obvykle provádějí vývojáři.
Systémové testování	Posouzení softwaru s ohledem na architektonický návrh. Systémové testování kontroluje shodu vyvinutého produktu a specifikace. V této fázi se kontroluje, zda systém funguje jako celek. Tato úroveň testování obvykle hledá problémy s designem a specifikacemi. Testování systému obvykle provádí tým testerů. Je důležité pochopit, že detekce chyb nízké úrovně v této fázi s sebou nese vysoké náklady na jejich opravu.
Akceptační testování	Posouzení softwaru s ohledem na požadavky. Akceptační testování pomáhá zjistit, zda vyvinutý produkt splňuje očekávání klienta. Akceptační testování se obvykle provádí na straně klienta, což zahrnuje testery a beta uživatele.

Tabulka 7 - Úrovně testování [15]

3.2.4 Typy testování

Tento odstavec pojednává o hlavních typech z hlediska předmětu testování, a to funkčním a nefunkčním testování.

Funkční testování ověřuje, zda součást nebo systém jako celek splňuje funkční požadavky. Nefunkční testování je zaměřeno na testování charakteristik, které nesouvisejí s funkčností, a to: spolehlivost, účinnost, použitelnost, udržovatelnost a přenositelnost, pohodlí, dostupnost a další. [13]

V následující tabulce jsou uvedeny poddruhy funkčního a nefunkčního testování:

Functional testing	
Functional correctness testing	Testování, zda funkce vyvinutého softwaru odpovídají specifikovaným nebo implikovaným požadavkům. Tento podtyp se provádí na všech úrovních testování.
Functional appropriateness testing	Analýza vhodnosti funkcí aplikace k provedení nějakého obchodního úkolu. Obvykle se provádí na systémové úrovni testování, ale může být také provedeno v pozdějších fázích integrační úrovně testování.
Function completeness testing	Kontrola, do jaké míry rozvinutá sada funkcí pokrývá všechny zadané úkoly a cíle uživatele. Tento podtyp testování se provádí na všech úrovních testování.
Non-functional testing	
GUI testing	Analýza shody grafického uživatelského rozhraní programu se specifikacemi, rozvrženími, prototypy, standardy.
Usability testing	Testování použitelnosti vyvíjené aplikace, která je založena na standardech, osvědčených postupech a zahrnuje beta-uživatele jako testery.

Installation testing	Testování vyvinutého softwaru, zaměřené na kontrolu procesů instalace, odstranění, obnovy, aktualizace, licencování.
Security testing	Testování pro hodnocení bezpečnosti softwarového produktu.
Accessibility testing	Testování za účelem zjištění úrovně přístupnosti aplikace pro uživatele se zdravotním postižením.
Performance testing	Proces testování k určení výkonu softwarového produktu.
Load testing	Tento podtyp testování se provádí za účelem vyhodnocení chování systému při zvyšující se zátěži (počet souběžných uživatelů nebo např. počet transakcí). Provádí se za účelem stanovení maximální přípustné úrovně zatížení.
Stress testing	Typ testování výkonu, který hodnotí systém nebo komponentu na hranici zátěže nebo ve stavu omezených zdrojů, jako je paměť nebo přístup k serveru.
Internationalization testing	Analýza aplikace a jejího rozhraní pro schopnost překladu.
Localization testing	Analýza vyvíjeného softwaru a jeho doprovodné dokumentace pro správnost v konkrétní lokalitě
Cross browser testing	Typ testování kompatibility zaměřený na analýzu toho, jak aplikace funguje v různých prohlížečích nebo různých verzích stejného prohlížeče.
Cross platform testing	Kontrola výkonu aplikace na různých operačních systémech/platformách.

Tabulka 8 - Poddruhy funkčního a nefunkčního testování [13] [14]

3.2.5 Porovnání typů prostředí

V procesu plánování testů je třeba vzít v úvahu infrastrukturu testovacího prostředí. Příprava nutných prostředí může zahrnovat instalaci potřebných nástrojů, přípravu testovací databáze a její naplnění testovacími daty. Pokud testovací prostředí není připraveno včas

nebo je nekvalitní, může se tento problém odhalit až v průběhu testování, což nevyhnutelně povede nejen ke zvýšení nákladů, ale i času potřebného pro vývoj softwaru. [23]

Definice testovacího prostředí dle [16]:

„Testovací prostředí je prostředí obsahující hardware, vybavení, simulátory, softwarové nástroje a další podpůrné prvky potřebné k provedení testu."

Níže uvedená tabulka ukazuje srovnání nejběžněji používaných prostředí pro vývoj a testování softwaru [20]:

Název prostředí	Zkratka	Funkce	Shodnost s produkčním prostředím	Výkonnost	Správa prostředí	Integrova-nost
Pískoviště	–	Prototypování, ověřování vývojo-vých 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, systémo-vé 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 akcep-tační testy, testy nasazení	Ano (SW i HW platformy)	Ano	Oddělení provozu	Ano
Podpora pro-dukce (Produc-tion support)	PRS	Simulování produkčních chyb, tes-tování 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 zálo-ha (Production and backup)	PRO	Produkční prostředí	–	Ano	Oddělení provozu	Ano
	BAC	Záložní prostředí	–	Ano	Oddělení provozu	Ano

Tabulka 9 - Typy prostředí a jejich atributy [20]

3.2.6 Automatizace procesu testování

Role testování při vývoji softwaru je často podceňována a proces je nákladný a časově náročný. Podle informací převzatých z knihy „Introduction to Software Testing", která byla vydána University of Cambridge [15]:

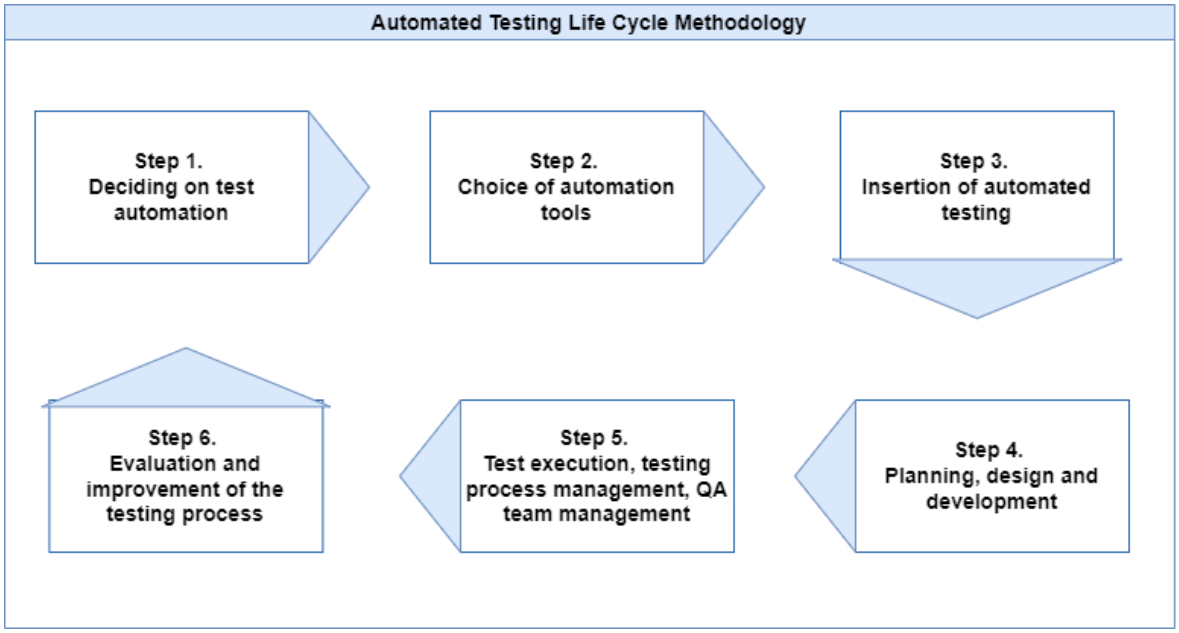
„Testování představuje až 50 % celkových nákladů na vývoj softwaru a u aplikací kritických pro bezpečnost to může být až 70 %."

Hlavním nástrojem, který pomáhá výrazně snížit náklady na testování, zvýšit rychlost jeho provádění, snížit možnost chyb a zjednodušit neustálé regresní testování stejného typu, je automatizace tohoto procesu. [15]

Vymezení pojmu automatické testování podle zdroje [17]:

„Automatické testování je řízení práce a provádění testovacích činností, včetně vývoje a provádění testovacích skriptů, za účelem splnění požadavků na testování, pomocí automatizovaných testovacích nástrojů."

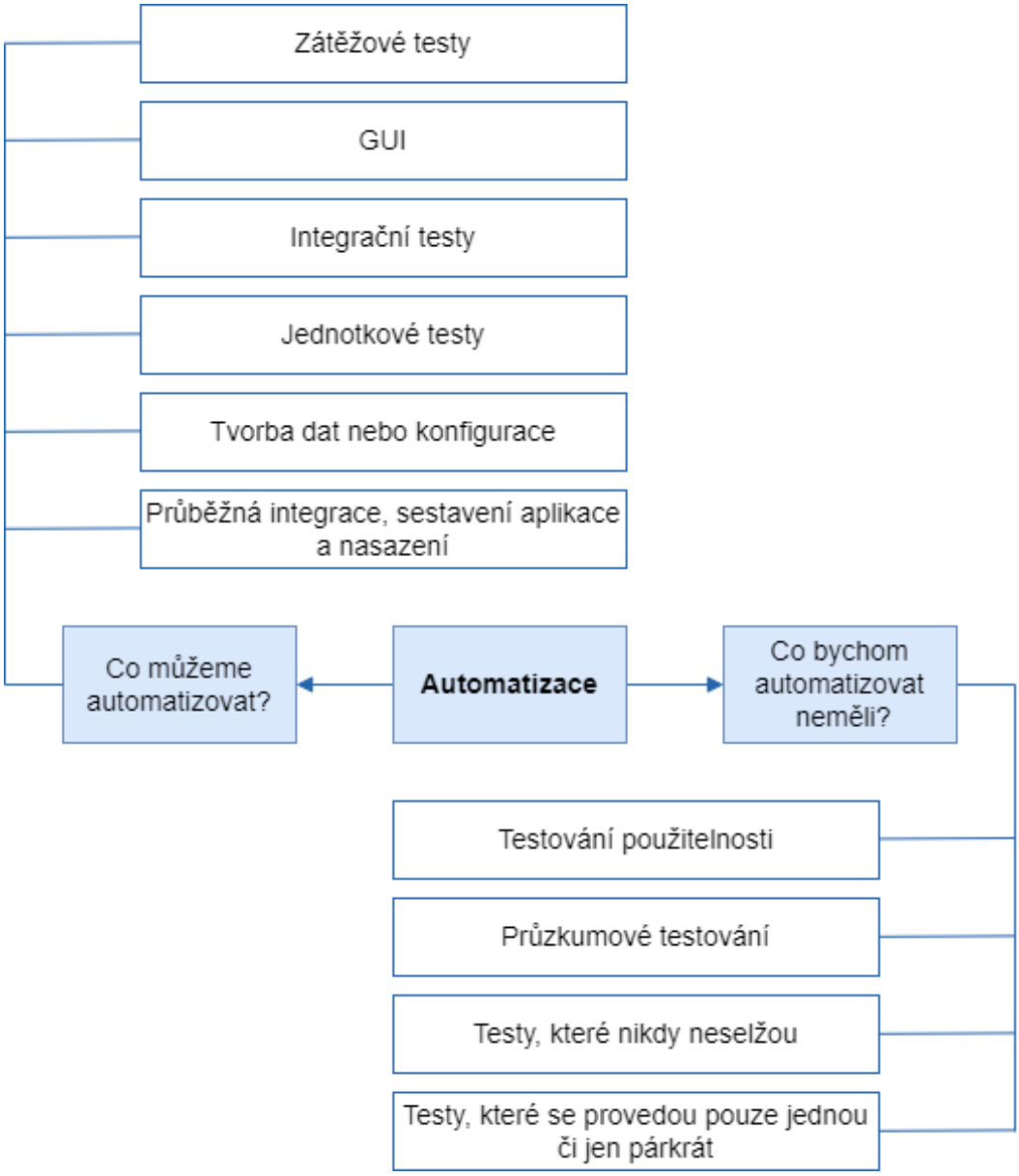
Následující schéma ukazuje metodiku automatického testování [17]:



Obrázek 9 - Metodika životního cyklu automatizovaného testování [17]

Diagram ukazuje, že vše začíná rozhodnutím o použití automatizace testování, po kterém je nutné vybrat potřebné automatizované testovací nástroje. Třetím krokem je zavedení automatizovaného testování do procesu vývoje produktu, poté začíná jeden z nejdůležitějších kroků „plánování – návrh – vývoj" automatizovaných testovacích skriptů. Pátým krokem je přímé provádění testovacích skriptů, stejně jako paralelní řízení týmu QA a testovacího procesu. Tento cyklus končí posouzením procesu testování a vypracováním návrhů na zlepšení tohoto procesu. [17]

Následující diagram ukazuje, co se doporučuje automatizovat z hlediska metodiky Agile [23]:



Obrázek 10 - Agile-strategie automatizace testování [23]

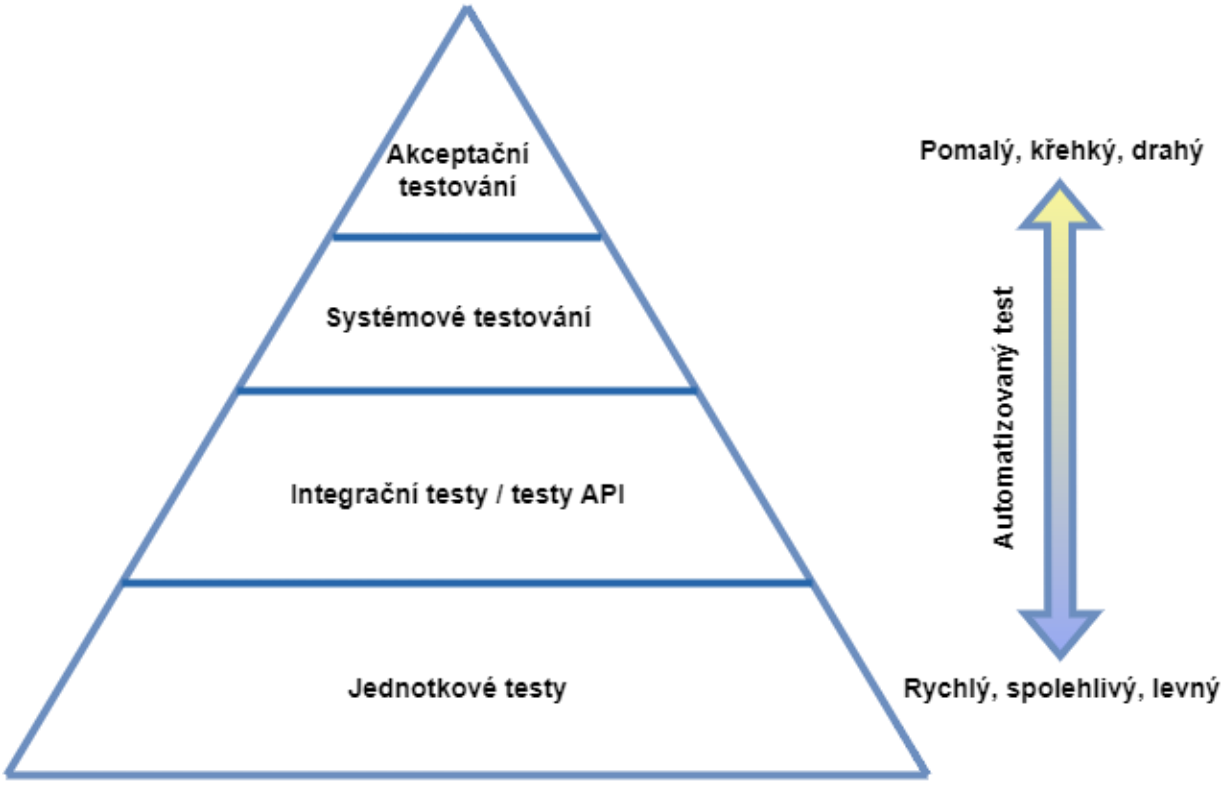
V podmínkách neustálých změn a přidávání nových požadavků, automatické testování slouží jako důležitý ověřovací mechanismus, k zajištění přesnosti a stability softwarového produktu s každou novou verzí. [23]

3.2.7 Zvláštností testování aplikací mikroslužeb

Jednou z hlavních výhod architektury mikroslužeb je zlepšená testovatelnost, ale úspěšné testování vyžaduje aktivní zavedení automatizovaného testování do procesu vývoje. Při testování mikroslužeb je důležité zkontrolovat, zda služby vzájemně správně spolupracují, ale aby se zkrátila doba testování, je nutné [24]:

1. upřednostnit úrovně testů pro automatizaci,
2. minimalizovat počet složitých a nespolehlivých testů, které zahrnují velké množství služeb.

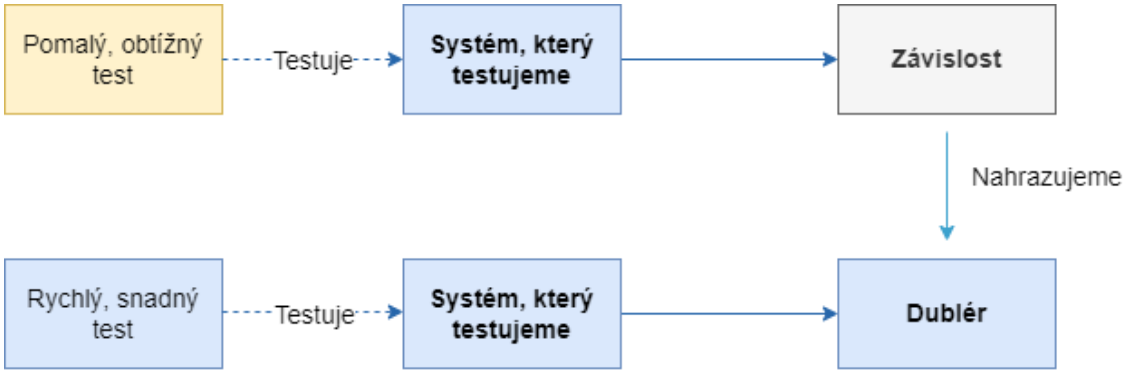
Pro stanovení priorit bude použita testovací pyramida [24]:



Obrázek 11 - Testovací pyramida [24]

Testovací pyramida pomáhá určit poměr mezi úrovněmi testování a počtem automatizovaných testů. Čím vyšší je úroveň testování, tím méně automatických testů je třeba psát, protože testy jsou pomalejší, méně spolehlivé a dražší. V souladu s tím je nutné automatizovat testy jednotek a testy na úrovni integrace. [24]

V aplikaci s architekturou mikroslužeb je klíčová integrace služeb. Pro kontrolu správné interakce je třeba použít API (Application Programming Interface). Toto rozhraní zahájí interakci, kterou lze použít k ověření správnosti vráceného výsledku. Je však třeba se vyhnout psaní těžkých a pomalých systémových testů, protože to spustí mnoho dalších přechodných závislostí těchto služeb. Řešením tohoto problému může být nahrazení závislostí "dublérem". Dublér je objekt, který simuluje chování závislosti. [24]



Obrázek 12 - Použití dubléru k testování systému v izolaci [24]

Obrázek 12 ukazuje způsob použití dubléru. Existují dva typy "dublérů": stubs a mocks. Stubs vrací hodnoty do testovaného systému. Mocks jsou obvykle používány automatickými testy k ověření, že systém správně volá své závislosti. Často se jako stubs používají mocks. [24]

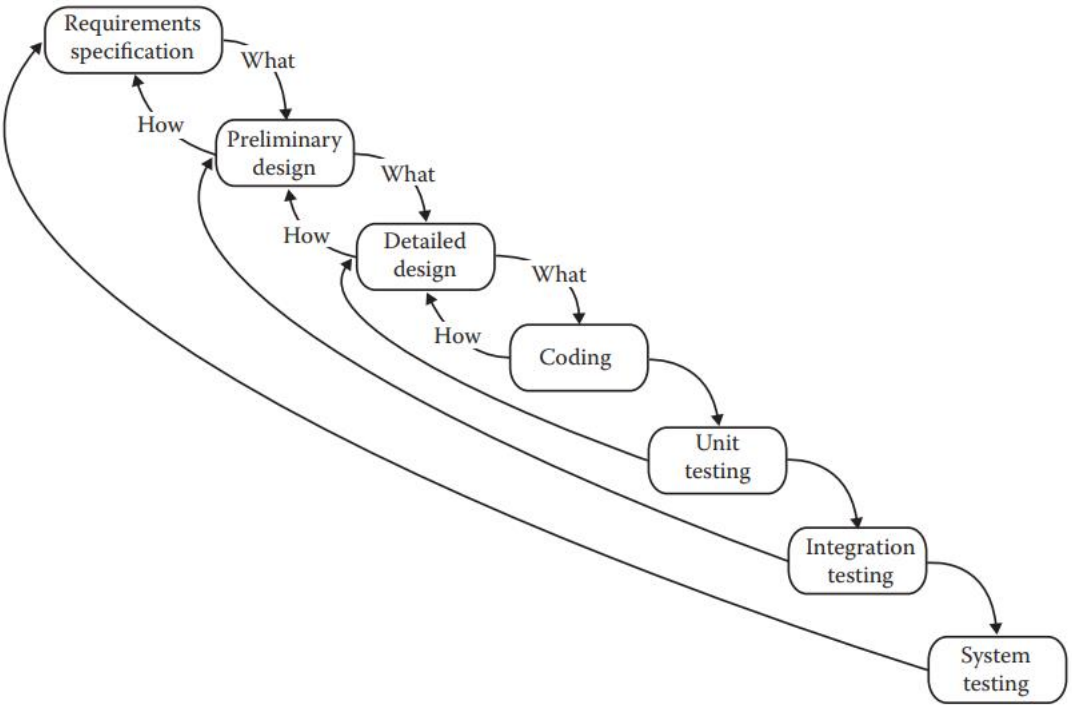
3.3 Metodiky testování, vývoje a provozu

3.3.1 Tradiční metodiky vývoje softwaru

Jednou z tradičních technik vývoje softwaru je Waterfall. Tato technika se objevila jako výsledek rozvoje vědeckého managementu v 70. letech minulého století a byla nejpopulárnější v průběhu následujících 30 let. [10]

Metodologie Waterfall je založena na „vývoji shora dolů" a designu pomocí funkčního rozkladu. Ve Waterfallu je každá předchozí fáze základem pro další. Dokud není určitá fáze dokončena, nelze zahájit novou fázi. [11]

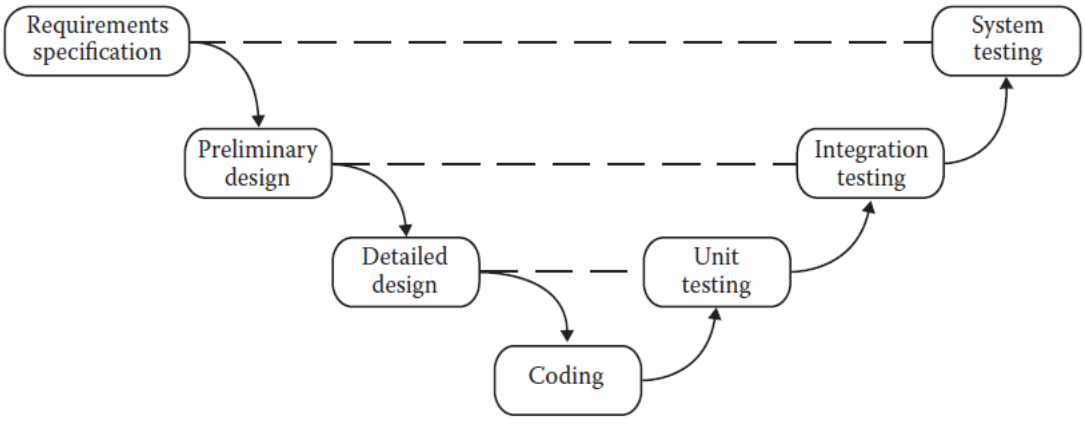
Následující diagram znázorňuje životní cyklus metodiky Waterfall z hlediska testování [11]:



Obrázek 13 - Životní cyklus metodiky Waterfall [11]

Ve Waterfallu hrají důležitou roli informace získané v určité fázi vývoje, na základě kterých se vyvíjí testování pro tuto úroveň.

Na základě Waterfallu byla vytvořena další metodika vývoje softwaru – V-model:



Obrázek 14 - Životní cyklus metodiky V-model [11]

Stejně jako v metodice Waterfall, tak i ve V-modelu každá předchozí fáze určuje, jak a kdy bude provedena další fáze. Rozdíl je v tom, že V-model také definuje korelaci mezi vývojovými kroky a určitými typy testování. Například na základě tohoto diagramu lze vidět, že testovací případy pro testování na úrovni systému by měly být vyvinuty v souladu

se specifikací požadavků a testování jednotek by mělo být založeno na detailním návrhu produktu. Tato metodika také určuje pořadí, ve kterém by měly být typy testování prováděny: nejprve by mělo být provedeno testování jednotek, poté integrační testování a poté by mělo být dokončeno testování systému. Toto pořadí lze definovat jako přístup „zdola nahoru". Tento přístup byl vytvořen v souladu s jasnou logikou: Nejprve je třeba otestovat jednotlivé komponenty, poté zkontrolovat výkon v subsystému a poté bude testován celý systém. Testování systému zpravidla hraničí s akceptačním testováním. [11]

Následující body lze zdůraznit jako výhody používání Waterfallu a V-modelu [10]:

- díky hierarchické struktuře řízení projektů lze tyto techniky v praxi snadno používat,
- pro každou fázi by měla být definována jasná kritéria pro dokončení (výstup).

Hlavní nevýhody a omezení uvažovaných metodik jsou [10]:

- dlouhá zpětná vazba mezi fází specifikace požadavků a testováním systému, která obvykle nezahrnuje zákazníka,
- model klade důraz na analýzu a téměř vylučuje syntézu, ke které dochází pouze v době integračního testování,
- chyby provedené ve specifikaci pronikají do zbývajících fází životního cyklu projektu a jsou zpravidla detekovány až ve fázi testování, která se provádí ve fázích dokončení projektu.

Vzhledem k těmto omezením při použití Waterfallu nebo V-modelu bylo vynaloženo obrovské množství času pouze na vývoj specifikace požadavků, protože specifikace by měla být úplná a jasná, ale ani tato opatření nemohla pomoci k úplnému odstranění chyby. [10]

3.3.2 Agile

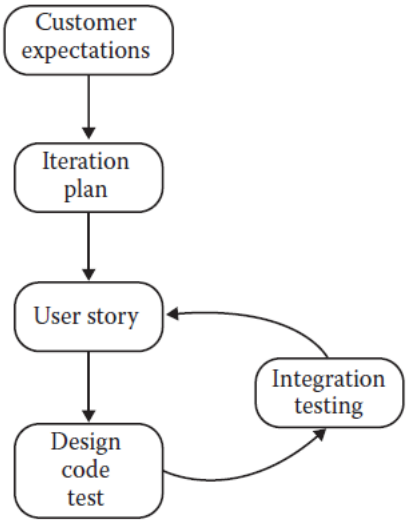
Historie agilní metodiky začala v únoru 2001, kdy skupina 17 vývojářů vytvořila Agile Manifesto, které bylo publikováno na http://agilemanifesto.org/ a přeloženo do 42 jazyků. [11]

Hlavní charakteristiky životního cyklu agile jsou [11]:

- vývoj řídí klient,

- vývoj „zdola nahoru”,
- flexibilita ve vztahu k požadavkům specifikace,
- častá dodávka funkčních komponentů.

Následující diagram znázorňuje životní cyklus agilní metodiky:

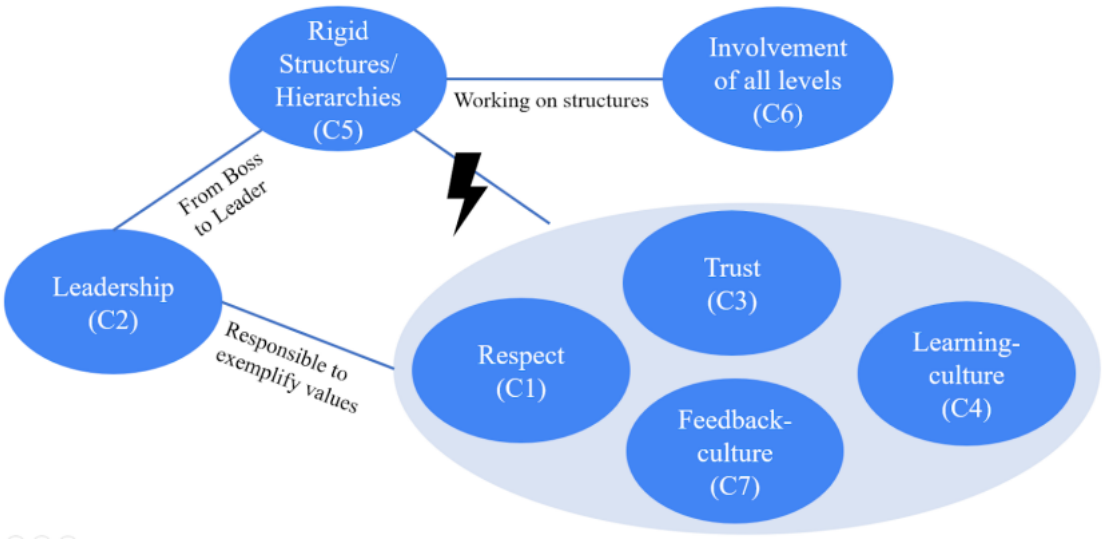


Obrázek 15 - Obecný životní cyklus metodiky agile [11]

V agilních metodikách hrají hlavní roli očekávání zákazníků. Zákazníci sdělují svá očekávání prostřednictvím „user-story”. Tyto user-story jsou posuzovány z hlediska požadavků na vyvíjený produkt. Na základě očekávání zákazníků je také vypracován iterační plán. Iterace je jedním z hlavních prvků agilního vývoje, který pomáhá vyhnout se zastřešujícímu plánu na samém začátku projektu, kdy je obvykle minimální pochopení toho, co je třeba udělat. Pro každou iteraci jsou definovány úkoly, které může tým během nadcházející iterace splnit. Na vývoji iteračního plánu se podílí celý tým a tento proces končí popisem a dohodou cílů iterace. Plánování probíhá nepřetržitě, prostřednictvím procesu neustálého sledování a přizpůsobování. To umožňuje projektu se měnit a vyvíjet s tím, jak roste porozumění a objevují se další detaily a požadavky. [11]

Iterace je obvykle krátký cyklus, který zahrnuje tři fáze: návrh – kód – testování. Agilní projekt končí po dokončení všech user-story klienta. [11]

Je důležité pochopit, že navzdory mnoha výhodám existují bariéry, které znesnadňují aplikaci agilní metodiky v praxi. Následující diagram ukazuje model bariér, se kterými se lze setkat při používání agilních metodologií:



Obrázek 16 - Model klíčových výzev souvisejících s metodikou agile [9]

Podle článku s názvem „Key Challenges with Agile Culture - A Survey among Practitioners" nejčastějšími překážkami jsou kulturní konflikty a obecný odpor ke změnám. Obrázek 16 ukazuje, že problémy byly identifikovány s respektem, důvěrou, kulturou učení a také kulturou poskytování zpětné vazby. Tyto bariéry zpravidla vznikají v důsledku nedostatečné podpory týmu při snaze transformovat stávající proces na „flexibilnější". Je důležité pochopit, že tyto bariéry mohou být použity jako základ pro rozvoj silnější firemní kultury v rámci firmy. Odstavec C6 odráží rozdíl v bariérách v závislosti na uvažované úrovni (individuální, kolektivní, organizační), proto je při implementaci Agile nutné vzít v úvahu rysy projevu problémů a rychlost jejich rozvoje pro určité úrovně. Změna podnikové kultury není triviální úkol, proto je důležité pochopit, že úspěšná implementace tohoto procesu bude vyžadovat značné množství času. [9]

Mezi metodiky agile patří následující přístupy [11]:

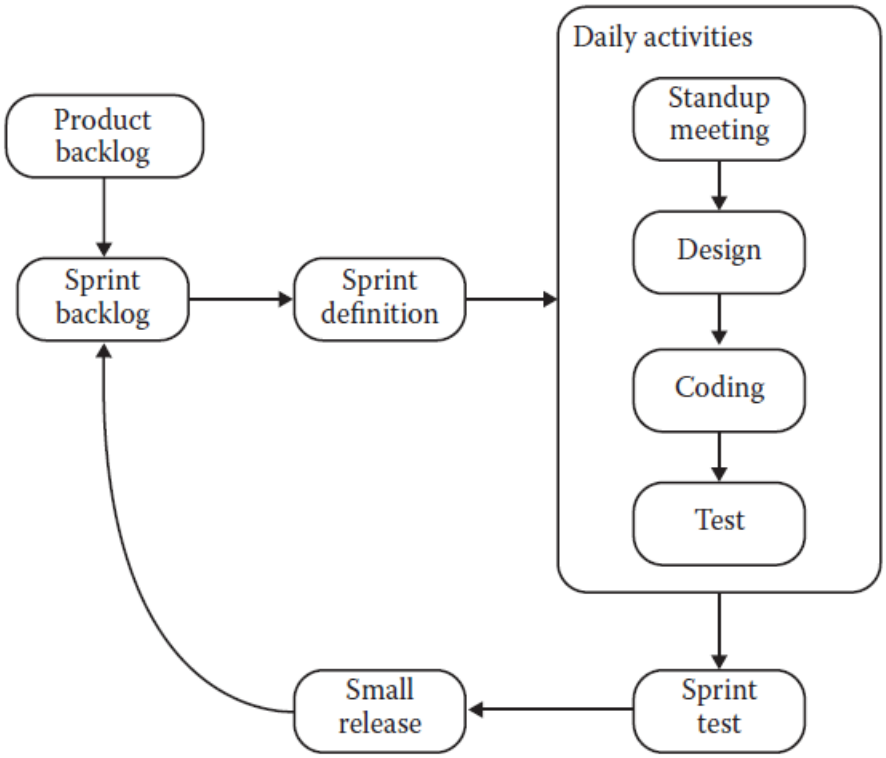
- Dynamic Systems Development Method (DSDM)
- Testy Řízený Vývoj (TDD)
- Adaptivní Vývoj Software (ASD)
- Vlastnostmi Řízený Vývoj (FDD)
- Extrémní Programování (XP)
- Lean Development
- SCRUM

- metodiky Crystal.

Jednou z nejpoužívanějších agilních metodik je SCRUM. Podle zdroje [11]:

„Název pochází z ragbyového manévru. Jedná se o týmovou hru, při které všichni hráči společně běhají s míčem po hřišti. Rugby scrimmage vyžaduje organizovanou týmovou práci – odtud název programovacího procesu.“

Následující diagram znázorňuje životní cyklus metodiky SCRUM. [11]



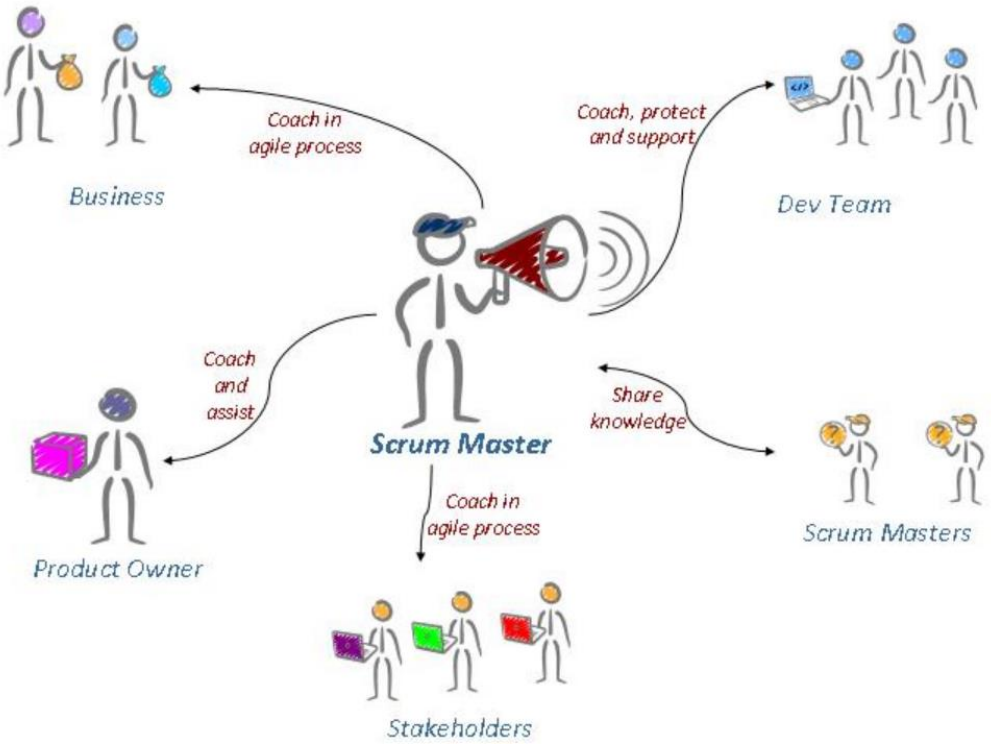
Obrázek 17 - Životní cyklus metodiky SCRUM [11]

Výběr sprint backlogu z product backlogu provádí zákazník. Tuto fázi je možné korelovat s fází vývoje požadavků v tradičních metodách vývoje. Na rozdíl od jiných agilních metodik SCRUM transformuje tradiční iterace na tzv. sprinty, trvající od 2 do 4 týdnů. Backlog sprintu je seznam úkolů, které tým plánuje dokončit v jednom sprintu. Definice sprintu je podobná fázi před návrhem. Právě v této fázi SCRUM tým určuje pořadí a obsah jednotlivých sprintů. V rámci sprintu se každé ráno provádějí schůzky, tzv. stand-up meeting. Stand-up schůzky jsou potřebné k pochopení toho, co bylo uděláno předchozí den a co je třeba udělat dnes. Následuje standardní cyklus: design-kód-testování. Po

dokončení sprintu probíhá integrační testování a uvolnění. Sprinty přispívají k rychlejšímu a lepšímu vývoji softwaru. [10] [11]

Co se týče testování, v metodice SCRUM se vyskytuje na dvou úrovních – komponentní (na konci každého dne) a integrační (na konci sprintu). [11]

SCRUM klade velký důraz na členy kolektivu a týmovou práci. SCRUM vytvořil nového člena týmu v osobě SCRUM mastera. SCRUM master působí jako kouč pro zbytek týmu. Následující obrázek znázorňuje roli SCRUM mastera v kolektivu [39]:



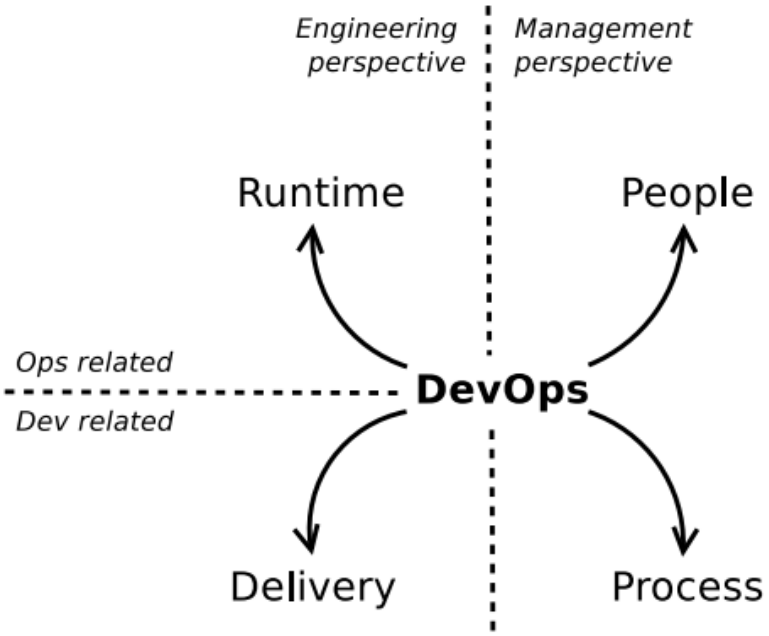
Obrázek 18 - Role SCRUM mastera v projektu [39]

3.3.3 DevOps

DevOps se objevil jako důsledek rozvoje metodiky agile. Je důležité si uvědomit, že metodika agile nevěnuje velkou pozornost metodám nasazení. Agile zahrnuje časté nasazení změn v prostředí, které se blíží podmínkám produkčního prostředí, v důsledku čehož je proces uvolňování produktu do produkčního prostředí namáhavý, se spoustou manuálních úkonů, což nepochybně vede k chybám. Přístup DevOps tento problém řeší tím, že nabízí různé postupy pro efektivní, iterativní dodávání softwaru do požadovaného prostředí. [8]

Hlavní rozdíl mezi metodikou DevOps a tradičním pohledem na proces vývoje spočívá v systémovém myšlení. Cílem DevOps je prolomit bariéry mezi všemi účastníky procesu vývoje softwaru zavedením vylepšení do všech fází životního cyklu vývoje. Vedoucí manažeři hrají důležitou roli v efektivním využívání DevOps tím, že používají otevřenou komunikaci a vytvářejí mechanismy zpětné vazby v rámci týmů i mezi nimi. [7]

Níže uvedený diagram znázorňuje koncept metodiky DevOps [8]:



Obrázek 19 - Koncepční mapa DevOps [8]

Tento koncept zahrnuje 4 hlavní kategorie: proces, lidé, doručování a doba zpracování. Kategorie lidé a proces musejí být zvažovány z hlediska managementu. Součástí procesu jsou koncepty, které přímo souvisejí s podnikáním. Do kategorie lidé patří dovednosti a koncepty, které jsou relevantní pro kulturu obchodní komunikace. Kategorie doručování a doba zpracování jsou zvažovány z technického hlediska. Obě kategorie jsou nezbytné pro efektivní vývoj softwaru. Průběžné doručování zabraňuje plýtvání časem během nasazení, zatímco doba zpracování zajišťuje spolehlivost v prostředí nasazení. [8]

Kategorie	**Popis**
Proces	Použití metodiky DevOps pomáhá snižovat rizika spojená s procesem vývoje a dodávky a pomáhá plnit požadavky firemních zákazníků, což

	nevyhnutelně vede ke zlepšení kvality produktu. Tohoto výsledku je dosaženo díky strukturovaným procesům spojeným s častým nasazením. Tento stav vede k časté komunikaci se zákazníky, což urychluje zpětnou vazbu, v důsledku čehož se zvyšuje spokojenost zákazníků.
Lidé	Myšlenkou DevOps je spojit lidi z vývojového týmu a provozního týmu dohromady. Pro realizaci této myšlenky hraje důležitou roli kultura komunikace v organizaci. Odstraňování bariér mezi těmito týmy vede k rozšíření dovedností vývojářů i provozních inženýrů.
Doručování	Proces doručení v DevOps je automatizovaný. Právě automatizace přispívá k frekvenci a spolehlivosti tohoto procesu. Mnoho automatizačních nástrojů je open source, což umožňuje používat metody průběžné integrace, kontrolu verzí vyvinutého softwaru, automatizované testování a mnoho dalšího.
Doba zpracování	Metodika DevOps dělá z hlavního cíle operačního týmu společný cíl, o který by měli usilovat všichni účastníci vývoje. Nasazované verze by měly být nejen časté, ale také stabilní a spolehlivé. Tato kategorie zahrnuje nejdůležitější ukazatele, které charakterizují kvalitu produktu: výkon, odolnost proti chybám, dostupnost, škálovatelnost a spolehlivost. Těchto kritérií lze dosáhnout různými způsoby, například pomocí kontejnerizace, virtualizace, cloudových služeb a monitorovacích systémů.

Tabulka 10 - Kategorie účastníků koncepční mapy DevOps [8]

Je důležité poznamenat, že kategorie doručování přímo souvisí s vývojovým týmem, zatímco doba realizace přímo souvisí s provozním týmem. [8]

Kniha Cloud Native DevOps with Kubernetes uvádí [1]:

„Nejdůležitější věcí, které je třeba ohledně DevOps porozumět, je to, že jde především o organizační, lidský problém, nikoli technický. To je v souladu s druhým

zákonem o poradenství Jerryho Weinberga: Bez ohledu na to, jak to na první pohled vypadá, je to vždy problém lidí."

Metodika DevOps pomáhá vytvořit sdílenou vizi problémů vývoje softwaru mezi vývojovým týmem, byznysem a zákazníky. V metodice DevOps neexistuje individuální odpovědnost a odpovědnost za selhání v procesu vývoje softwaru leží na týmu jako celku.

Potíž s používáním DevOps v praxi spočívá v tom, že často dochází k neshodám mezi vývojovými a provozními týmy. Hlavním úkolem provozních kolektivů je zpravidla dosažení stability a udržitelnosti vyvíjeného softwarového produktu, přičemž vývojáři se snaží urychlit proces tvorby změn rychlejším procesem dodání. K vyřešení těchto neshod je důležité pochopit, že tyto cíle se vzájemně nevylučují a lze jich dosáhnout společně, pokud všichni účastníci účinně spolupracují. [1]

3.4 CI/CD a CT

Rychlost a kvalita vývoje produktů jsou hlavní konkurenční výhody ve vývoji softwaru. Z tohoto důvodu byly tradiční vývojové modely nahrazeny novým konceptem CI/CD – průběžná integrace a průběžné doručování. Tento model pomáhá minimalizovat chyby a urychlit vývoj i kvalitu vyvíjeného produktu. [21]

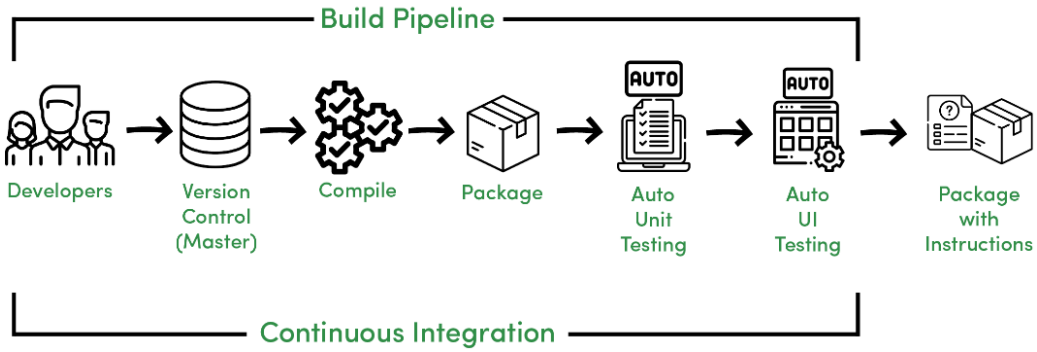
Dále budou zváženy součásti tohoto konceptu a jejich vztah k testování.

3.4.1 Průběžná integrace (Continuous Integration)

Koncept CI, který byl poprvé představen v roce 1991, se již stal de facto standardem v agilních projektech. Definice pojmu průběžné integrace dle zdroje [21]:

„Průběžná integrace je postup vývoje softwaru, kde členové týmu často integrují svou práci; obvykle se každý člověk integruje alespoň denně, což vede k několika integracím za den."

Níže je zobrazen diagram procesu CI [28]:



Obrázek 20 - Diagram procesu CI [28]

Během průběžné integrace buildy se skládají na serverech sestavení, které spouštějí automatické testy v určitých intervalech nebo po provedení každé změny kódu, načež jsou výsledky hlášeny vývojářům pomocí reportů. [21]

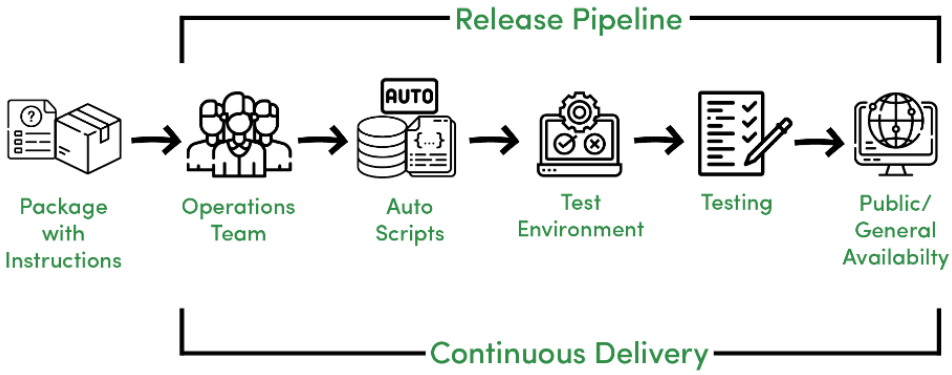
Výhody používání CI [21]:

- snižuje rizika,
- omezuje manuální rutiny,
- zvyšuje důvěru v produkt,
- včas identifikuje nedostatky,
- zkracuje dobu strávenou testováním,
- zlepšuje viditelnost projektu.

3.4.2 Průběžné doručování/nasazení (Continuous Delivery/Deployment)

Zkratku CD lze rozepsat jako Continuous Delivery nebo Continuous Deployment. Průběžné doručování (Continuous Delivery) je automatické doručení softwaru do testovacího nebo produkčního prostředí. Tento koncept předpokládá bezpečné, rychlé a udržitelné vydávání softwaru. [21]

Níže je uveden diagram procesu Continuous Delivery [28]:

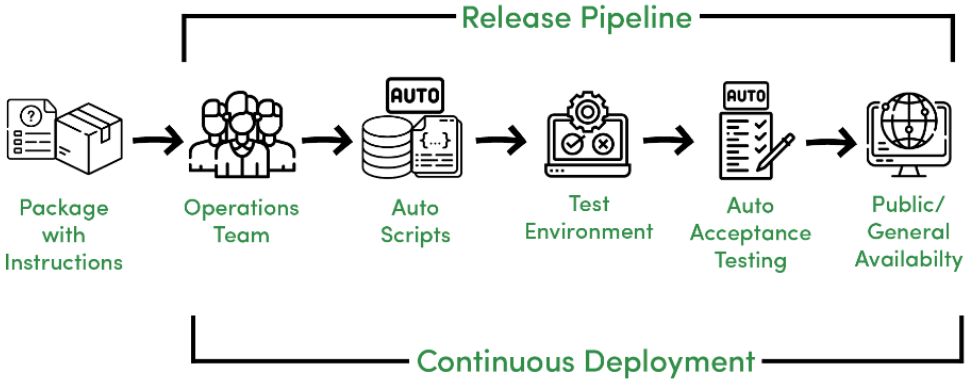


Obrázek 21 - Proces Continuous Delivery [28]

Po dokončení procesu Continuous Delivery dostane operační tým balíček s instrukcemi. Operační tým také předem připravuje automatické skripty, které se následně použijí v testovacím prostředí. Poté se provede testování a teprve v případě úspěšného dokončení testování schválí pracovník odeslání tohoto balíčku do produkčního prostředí. [21]

Průběžné nasazení (Continuous Deployment) je nasazení balíčku bez ručního zásahu. Tento koncept umožňuje snížit počet manuálních úkolů pro operační tým a tým QA v procesu nasazení vyvinutého produktu do produkčního prostředí. Během průběžného nasazení, po úspěšném dokončení automatického akceptačního testování dojde k automatickému nasazení balíčku. [21]

Obrázek níže ukazuje schéma procesu Continuous Deployment [28]:



Obrázek 22 - Proces Continuous Deployment [28]

Výhody používání CD [21]:

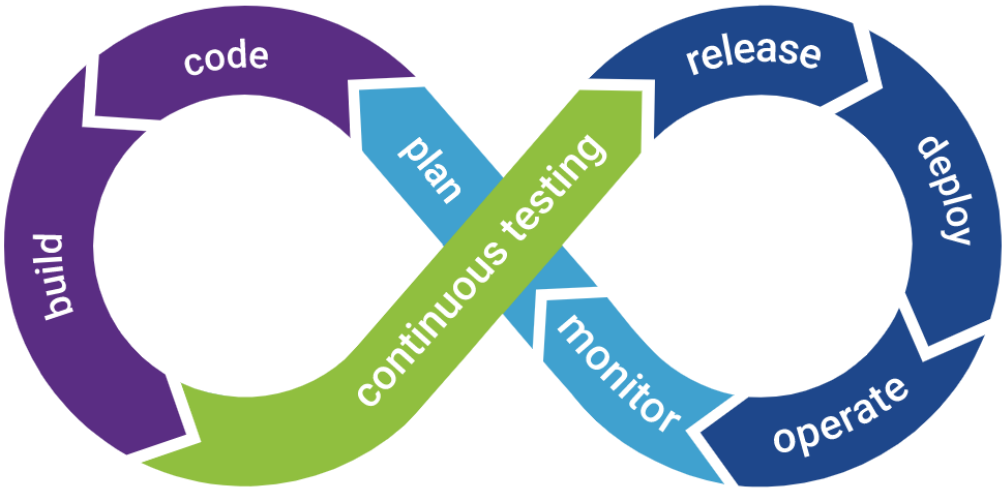
- urychluje dodávání nových funkcí,
- zrychluje zpětnou vazbu a postřehy,
- vytváří kulturu vysoké důvěry,
- zvyšuje schopnost reagovat na vnější události,
- snižuje „bolest“ při nasazení,
  buduje hlubší uživatelské vztahy,
- zlepšuje kvalitu.

3.4.3 Průběžné testování (Continuous Testing)

Koncept CI/CD také zahrnuje použití flexibilního přístupu k testování. Čím více dodávek během vývoje probíhá, tím více je potřeba automatizovat. I když mnoho moderních IT společností již používá koncept CI/CD v praxi, stále provádí ruční testování. Tento problém komplikuje nejen testování, ale celou pipeline CI/CD. [21]

Definice pojmu Continuous Testing (průběžné testování) dle zdroje [29]:

„Průběžné testování (Continuous Testing) je proces vývoje softwaru, ve kterém probíhá testování vyvíjené aplikace nepřetržitě po celý životní cyklus vývoje. Cílem CT je hodnotit kvalitu softwaru poskytováním kritické zpětné vazby předem a zajištěním vyšší kvality a rychlejší dodávky.”

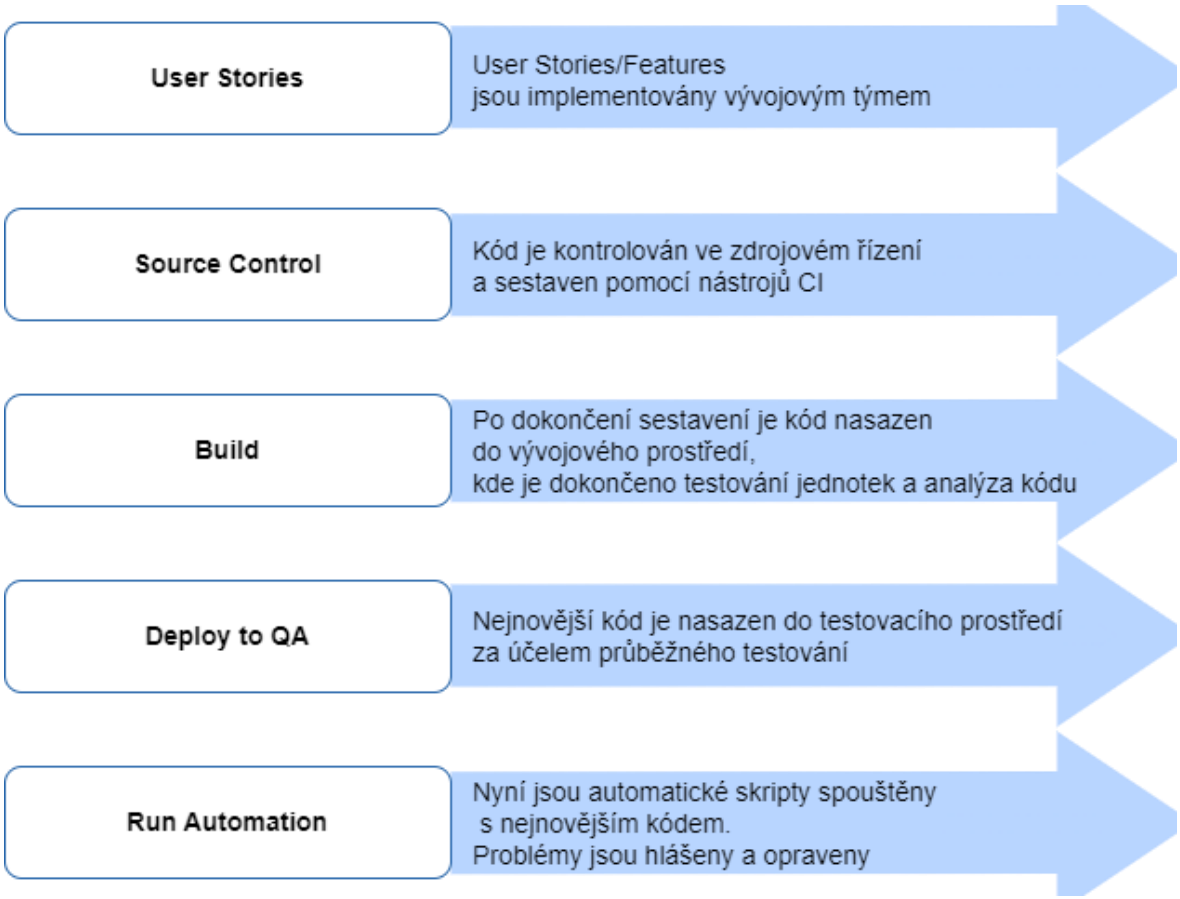


Obrázek 23 - Continuous Testing v procesu agile [29]

Podle reportu, který byl zveřejněn společnostmi RWS a Simform používá automatizované testování **od roku 2020 pouze 44 % společností na globálním trhu**. Je

důležité poznamenat, že mezi těmito společnostmi bylo v průměru pouze kolem 50 % testů automatizováno. [27]

Schéma implementace průběžného testování v praxi [31]:



Obrázek 24 - Schéma implementace průběžného testování v praxi [31]

Průběžné testování je koncept, který musí být prováděn průběžně. Výsledky testů poskytují celému týmu informace o kvalitě jeho služeb v různých fázích vývoje. [22]

Metodiky průběžného testování [30]:

Metodika	Popis
Shift-Left testing	Tato metodika upřednostňuje testování v rané fázi životního cyklu vývoje, aby se předešlo problémům, nebo snížení dopadu problémů na vývojový proces v budoucnu.

Shift-right testing	Tato metodika upřednostňuje testování ke konci životního cyklu vývoje se zaměřením na zlepšení uživatelské zkušenosti, celkového výkonu, odolnosti proti chybám a funkčnosti.
Smoke tests	Tato metodika se zaměřuje na provádění počáteční rychlé identifikace kritických problémů.
Unit testing	Testování jednotek. Tato metodika se obvykle používá ke kontrole netěsností zátěže nebo paměti v sestavení, aby se zjistily defekty v rané fázi vývoje.
Integration and messaging testing	Testování integrace a zasílání zpráv. Tato metodika pomáhá identifikovat chyby v procesu interakce několika modulů.
Functional testing	Funkční testování. Pomocí této metodiky je kontrolována shoda vyvíjeného produktu podle očekávání klienta a toho, zda jsou funkční procesy prováděny v souladu s požadavky.
Regression testing	Regresní testování. Tato metodika se používá k testování výkonu, funkčnosti a dalších kvalitativních charakteristik po opravách chyb.
User-acceptance testing	Akceptační testování. Tato technika spočívá v testování aplikace v prostředí blízkém produkčnímu prostředí. Často implementováno na straně klienta, QA týmem nebo specifickou skupinou uživatelů. Beta testování je příkladem akceptačního testování.

Tabulka 11 - Metodiky průběžného testování [30]

4 Vlastní práce

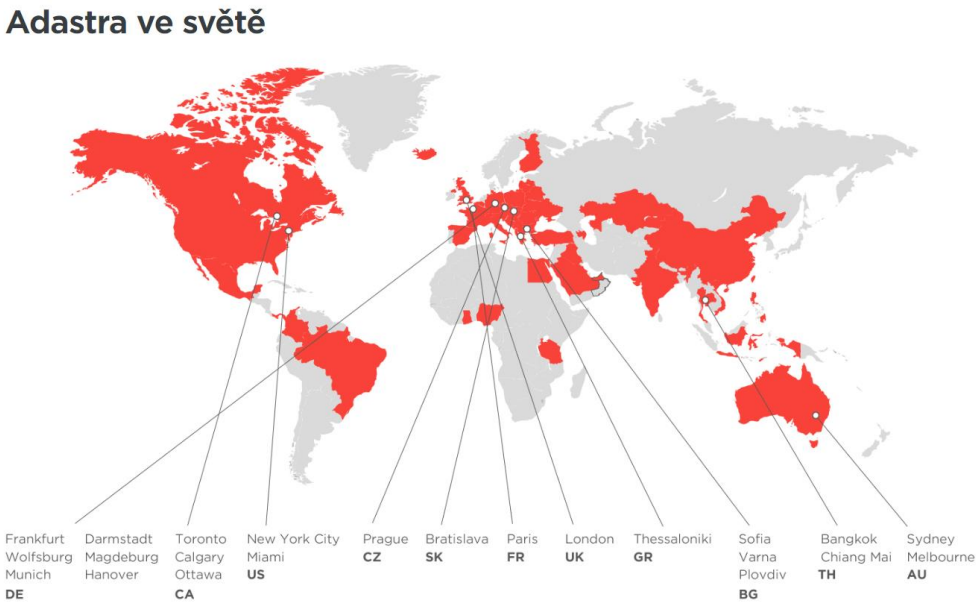
Praktická část je vytvořena na základě teoretických znalostí, získaných při zpracování literární rešerše k této práci. Analytická část vychází z analýzy společnosti Adastra, s. r. o., ve které je autorka této práce zaměstnancem. Oficiální pracovní pozice autorky je konzultant, ale její role v projektu je testovací analytik, takže její činnost přímo souvisí s tématem práce.

Projekt pro tuto diplomovou práci byl vybrán na základě možnosti reálně přispět ke zlepšení vývojového procesu, a to prostřednictvím využití metodiky průběžného testování. V rámci projektu již byla ve velké míře využita CI/CD, ale proces testování dosud nebyl kompletně automatizován. Všechny modely vytvořené v rámci této práce jsou sdíleny se členy projektového týmu.

K modelování procesů je použit open-source software Camunda Modeler 7.18.

4.1 Charakteristika společnosti Adastra, s. r. o.

Adastra je globální konzultační společnost, která poskytuje komplexní technologická řešení a služby společnostem v různých oblastech podnikání s cílem usnadnit digitální transformaci. Společnost byla založena 5. října v roce 2000. V současnosti má přes 2000 zaměstnanců, z toho více než 450 v Praze. Adastra působí ve 21 zemích.



Obrázek 25 - Adastra ve světě [46]

Adastra nabízí řešení v následujících oblastech:

- Informační management:
  - datové sklady
  - big data
  - business intelligence
  - cloud
- Aplikační vývoj:
  - vývoj mobilních aplikací
  - software
- Consulting:
  - business consulting
  - kreativní poradenství
- Pokročilá řešení:
  - umělá intelligence
  - internet of things (IoT)
  - ESG.

Technologickými partnery Adastry jsou společnosti jako Adobe, Ataccama, Google Cloud, IBM, Microsoft, ORACLE, SAP, SAS, Teradata, AWS a mnoho dalších.

4.2 Analýza současného stavu vybraného projektu

Praktická část této práce je založena na projektu Adastry pro zahraničního klienta. Z důvodu bezpečnosti a ochrany dat došlo ke změně některých údajů, avšak základní principy fungování procesů jsou zachovány.

V rámci projektu se vyvíjí cloud native mobilní aplikace, která slouží k objednání kreditních karet a užívání finančních služeb.

4.2.1 Členové projektového týmu

Do projektu jsou zapojeni:

Pozice	Popis
Frontend developer	frontend vývojář pro platformu Android

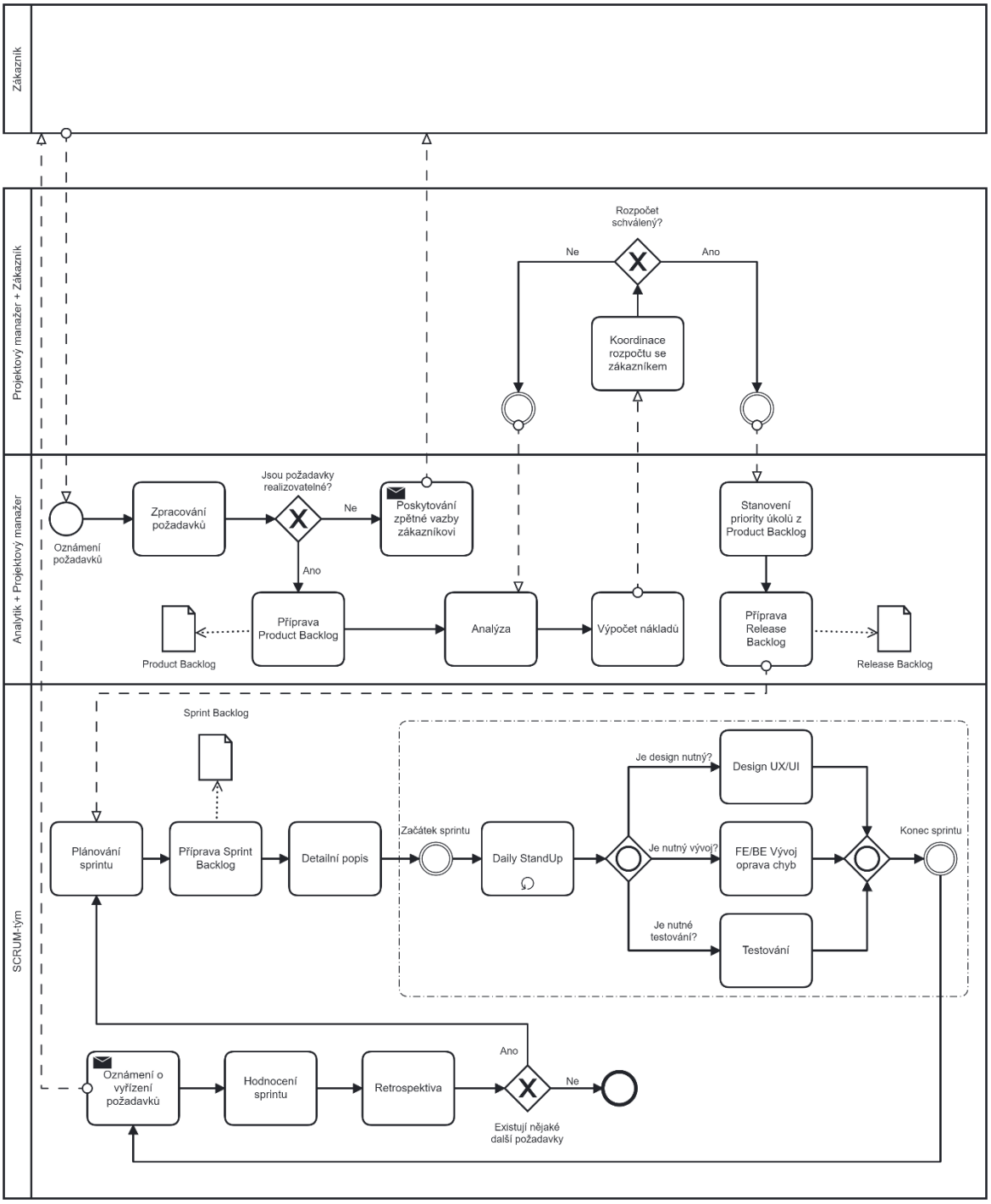
Frontend developer	frontend vývojář pro platformu iOS
Backend developer	backend vývojář nastavující IT-infrastrukturu (Azure, Kubernetes, pipelines, service bus apod.)
UI	user interface designer připravující design a wireframes pro mobilní aplikaci, podle které potom vyvíjejí frontend vývojáři
UX	user experience designer, který se stará o to, aby aplikace a její ovládací prvky byly dobře a pohodlně ovladatelné pro uživatele a aby jednotlivé prvky na obrazovce dávaly smysl
QA	tester, jehož úkolem je otestovat vyvinuté funkce, potvrdit správnost vývoje, případně hlásit chyby k opravě
PM	project manager – stará se o spokojenost klienta, zajišťuje, aby všichni v týmu věděli, co mají dělat, stará se o komunikaci s třetími stranami/vendory
Analyst	analytik – zjišťuje, co a jak má fungovat, sepisuje zadání, konzultuje s vývojáři technické možnosti a omezení, sepisuje funkční specifikaci

Tabulka 12 - Členové týmu [zdroj: autorka]

Na tomto projektu společnosti Adastra se podílí projektový manažer, analytik, dva designéři, dva frontendoví vývojáři, backendový vývojář a QA inženýr. Tým pracuje podle metodiky SCRUM. Projektový manažer komunikuje přímo se zákazníkem, získává od něj informace o jeho požadavcích a přáních týkajících se vyvíjené aplikace a vytváří tzv. product backlog, který obsahuje stručný popis všech požadovaných funkcí vyvíjené aplikace. Projektový manažer také diskutuje s analytikem o proveditelnosti požadavků a schvaluje se zákazníkem rozpočet. Analytik analyzuje úkoly v product backlogu, společně s projektovým manažerem stanoví jejich priority a vytváří release backlog. Dále se na základě release backlogu plánují sprinty. Do plánování jsou zapojeni všichni členové týmu SCRUM. V této fázi hrají hlavní roli při plánování vývojáři, designéři a tester. Na základě plánování sprintu je vytvořen sprint backlog, obsahující seznam úkolů, které budou realizovány v následujícím sprintu. V tomto projektu trvá sprint čtrnáct dní. V rámci sprintu se každý den konají ranní schůzky zvané standup. Tyto schůzky vede projektový manažer. Po ranní schůzce probíhá proces vývoje, který může zahrnovat návrh, vývoj a testování. Po dokončení sprintu je klient informován o výsledcích sprintu. Tým SCRUM vyhodnotí dokončený sprint. Proces sprintu je zakončen schůzkou, která se nazývá retrospektiva. Na této schůzce si účastníci procesu

sdělí své pocity z dokončeného sprintu a přednesou návrhy na zlepšení. V případě, že proces vývoje aplikace ještě neskončil a zákazník má nesplněné požadavky/přání na aplikaci, celý proces se opakuje znovu.

Následující schéma znázorňuje proces interakce mezi účastníky projektu v rámci metodiky SCRUM:



Obrázek 26 - Proces a metodika SCRUM [zdroj: autorka]

4.2.2 Požadavky na testování a kvalitu

Požadavky na testování a kvalitu vypracovává QA tým. Ten se skládá z QA architekta (není trvalým členem týmu) a QA inženýra. V případě potřeby může být do procesu zapojen analytik a další členové týmu.

Aktuálně používané požadavky na projekt jsou následující:

- Projekt využívá funkčních požadavků vytvořených na základě požadavků zákazníka. Z důvodu bezpečnosti a ochrany dat nebudou funkční požadavky uváděny.
- Nepřetržitost a spolehlivost: požadavky na to, jak se má aplikace chovat při různých zátěžových scénářích, jak rychle dokáže reagovat na uživatelské vstupy a jak se zachová v případě selhání komponent. Tím se kontroluje, zda aplikace nespadne v enormních situacích, při velkém zatížení a při nedostupnosti externích služeb.
- Bezpečnost: požadavky na to, jak aplikace ochrání citlivé informace uživatelů a jak bude zabráněno zneužití systému. Opatření na ochranu před hackerskými útoky a neoprávněným přístupem třetích stran. Splnění všech legislativních a aktuálních požadavků na bezpečnost (PSD2 atd.).
- Uživatelská přívětivost a design: požadavky na to, jak bude aplikace snadno použitelná a intuitivní pro uživatele, jak bude vypadat a jak bude reagovat na uživatelské vstupy.

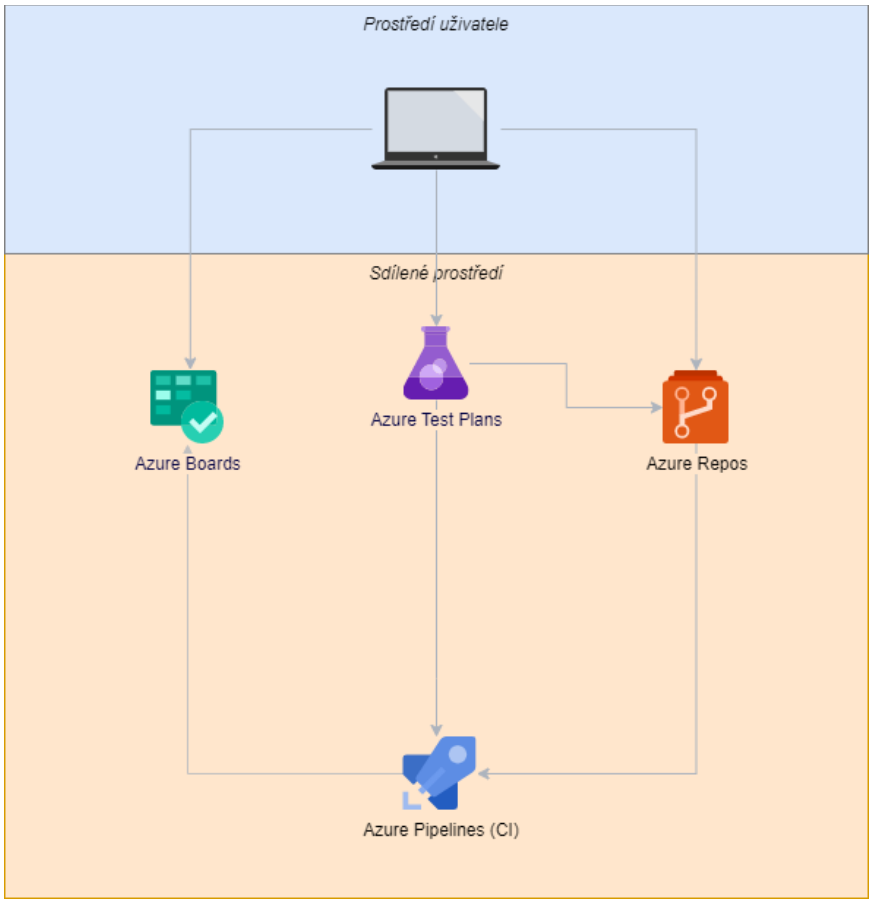
Definition of done (DoD) je soubor požadavků, které musejí být splněny, aby byla aplikace připravena k uvolnění na produkci. V případě tohoto projektu zahrnuje následující:

- kód je zkontrolován a schválen,
- všechny funkce a požadavky jsou implementovány a otestovány včetně požadovaných změn (tzv. change requests),
- všechny chyby jsou opraveny a otestovány,
- byly provedeny předběžné testy na klientské infrastruktuře a analýza možných problémů,
- automatické testy jsou propojené s produkčním prostředím a je nastavené pravidelné odesílaní reportu vývojovému tymu.

4.2.3 Infrastruktura testovacího prostředí

Infrastruktura testovacího prostředí se skládá pouze ze dvou součástí: prostředí uživatele a sdíleného prostředí. Prostředím uživatele je počítač QA inženýra, jehož prostřednictvím se řídí všechny procesy související s testováním. QA inženýr píše automatizované skripty v prostředí IntelliJ a rozhoduje, kdy a které testy budou spuštěny. Sdílené prostředí je umístěno v cloudu a postaveno na Azure DevOps Services. Azure DevOps je služba SaaS (software jako služba) od společnosti Microsoft, která poskytuje sadu potřebných nástrojů pro vývoj, testování, integraci a také nasazení softwaru. Azure DevOps Services je bezpečná, spolehlivá, snadno škálovatelná a globálně dostupná cloudová služba. Použití tohoto řešení usnadňuje nastavení potřebné infrastruktury.

Následující schéma znázorňuje testovací prostředí použité v analyzovaném projektu:



Obrázek 27 - Infrastruktura testovacího prostředí [zdroj: autorka]

Popis prvků z obrázku:

Prvek	Popis
Azure Boards	V rámci projektu se k plánování činností a odevzdávání úkolů používá aplikace Azure Boards. Tato služba poskytuje testerům aktuální informace o stavu úkolů a chybách a různé přehledy.
Azure Repos	Azure Repos je služba pro řízení verzí. Tento repozitář synchronizuje kód, který již byl zkontrolován. Kód v úložišti může pocházet od vývojářů a QA inženýrů v případě vývoje automatizovaných testů.
Azure Pipelines	Pipeline umožňuje automatizovat proces sestavení, dodání a testování. Tento proces se spouští doručením nového kódu do Azure Repos.
Azure Test Plans	Během procesu CI se provádějí určité automatické testy. Azure Test Plans pomáhají zaznamenávat proces provádění a výsledky provedených testů.

Tabulka 13 - Popis služeb Azure DevOps Services [zdroj: autorka]

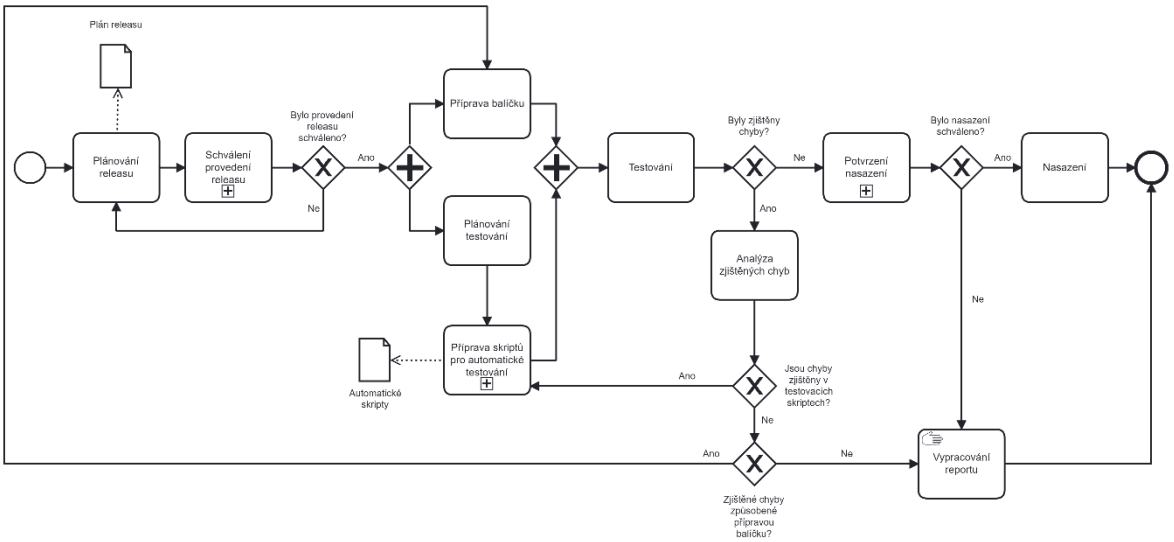
Důležité je, že díky integraci Azure DevOps Services s aplikací Microsoft Outlook budou mít vývojáři a testeři aktuální informace o průběhu testování a celém procesu integrace.

4.2.4 Proces nasazení

Release je jednou z nejdůležitějších a nejočekávanějších fází procesu vývoje softwaru. Příprava na release obvykle vyžaduje velké úsilí a spoustu času od všech, kteří se na projektu podílejí. Proces nasazení v tomto projektu začíná plánováním releasu. Plánování iniciuje a koordinuje manažer projektu. Na tomto procesu se nutně podílejí vývojáři, QA inženýr a také architekt kvality, který není stálým členem týmu. Na základě výsledků plánování je vytvořen plán releasu. Tento plán obsahuje informace o tom, které opravy nebo nové funkce budou v rámci releasu nasazeny, určuje osoby, které se na něm podílejí, a uvádí datum releasu. Jakmile je plán releasu připraven, následuje schvalovací proces, do kterého je zapojen projektový manažer a zákazník. Pokud se nepodařilo dohodnout nasazení, vrací se proces do fáze plánování. Pokud je proces schválen, vývojáři připravují release-package.

Současně s přípravou balíčku QA inženýr a architekt kvality vytvoří plán testování, poté QA inženýr vypracovává potřebné automatizované skripty. Následně architekt kvality provádí revizi kódu automatizovaných skriptů. Po dokončení tohoto procesu a přípravy balíčku se provede testování. V případě, že jsou během procesu testování zjištěny chyby, provede se analýza chyb a teprve na základě výsledků této analýzy se rozhodne, zda je možné tyto chyby v procesu nasazení opravit a balíček nasadit, nebo zda má být proces release ukončen. V případě, že nejsou zjištěny žádné chyby, vývojář provádějící nasazení schválí balíček k nasazení do produkčního prostředí a poté je provedeno automatické nasazení a proces je dokončen.

Níže je zobrazen diagram procesu nasazení:



Obrázek 28 - Proces nasazení [zdroj: autorka]

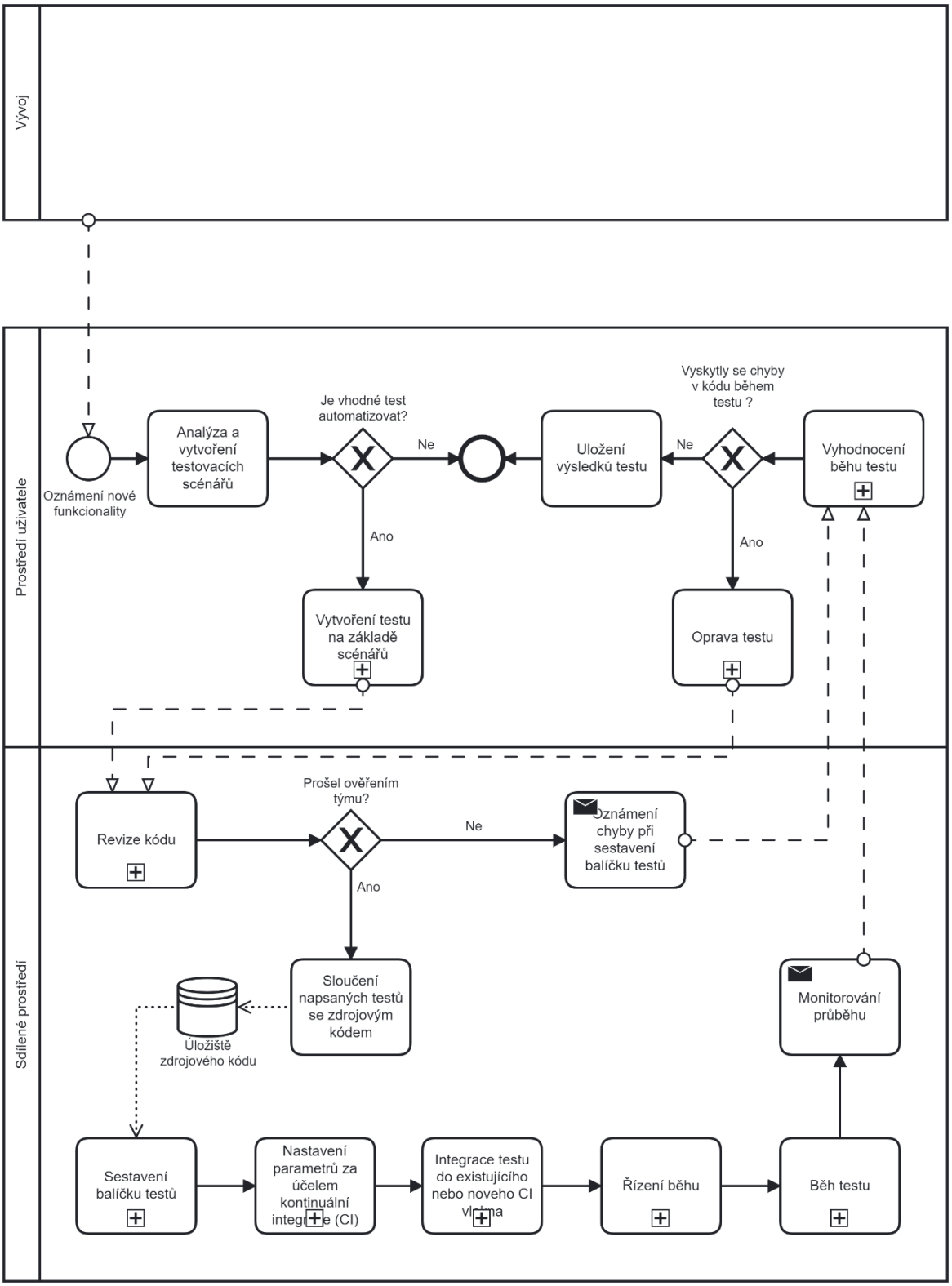
4.3 Analýza implementace automatizovaného testování v projektu

4.3.1 Proces vytváření automatického testu

Výběr testů, které budou automatizovány, závisí na různých faktorech, jako jsou náklady, časové omezení, priorita testů a technická proveditelnost. V rámci projektu se obvykle automatizují testy, které jsou opakovatelné, vyžadují vysokou úroveň přesnosti a jsou časově náročné při ručním testování. Aktuálně se zastoupení automatizovaných testů pohybuje v rozmezí od 50 % do 60 %.

Vývoj automatizovaných testů probíhá v testovacím prostředí, jehož infrastruktura byla popsána v části 4.2.3. Následující obrázek znázorňuje model automatizovaného procesu

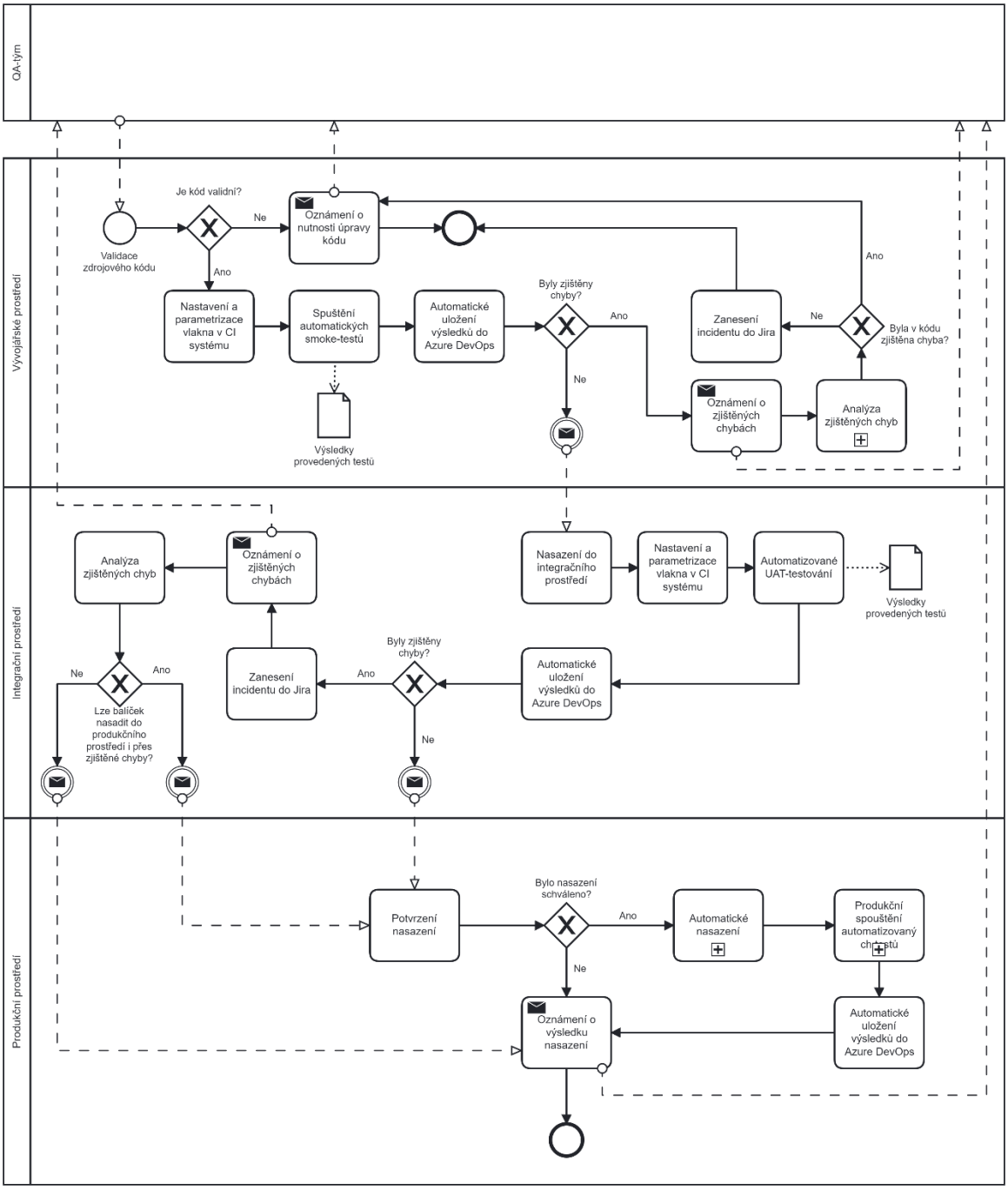
vývoje testů v rámci použití CI/CD. Tento proces zatím není v projektu plně implementován, konkrétně po přidání kódu do repozitáře se testy spouštějí ručně. Automatizace tohoto procesu je v současné době ve fázi plánování.



Obrázek 29 - Proces vytváření automatického testu za použití Azure Pipelines [zdroj: autorka]

4.3.2 Automatizované testování v rámci CT

Přestože projekt již využívá CI/CD, tato metodika je implementována pouze pro integrační a nasazovací procesy, zatímco proces testování stále zahrnuje spoustu manuálních aktivit. Níže je vyobrazen model automatizovaného procesu testování, který byl vyvinut na základě metodiky průběžného testování. Výhody a přínosy použití tohoto modelu budou popsány v kapitole „Výsledky a diskuse”.



Obrázek 30 - Proces provádění automatizovaného testování pomocí konceptu CI/CD [zdroj: autorka]

Výsledky a diskuse

Cloud technologii si každoročně jako základ pro budování vlastní IT infrastruktury vybírá stále více firem z různých odvětví. Navzdory popularitě cloud native je použití tohoto přístupu v praxi často náročné. V rámci diplomové práce byly identifikovány různé obtíže, se kterými se firmy při využívání cloud native v praxi potýkají, a to: nedostatek odborníků, obtížné zvládnutí vlastností mikroslužeb, bezpečnost, IT infrastruktura zaměřená na monolitické aplikace a mnoho dalších. Analýza však ukazuje, že výhody používání cloud native výrazně převažují nad potenciálními problémy.

V rámci analýzy odborné literatury, provedené v teoretické části, bylo potvrzeno, že testování hraje v procesu vývoje softwaru jednu z klíčových rolí. V podnikové praxi je však bohužel často věnována nedostatečná pozornost plánování a realizaci procesu testování, což vede nejen ke zhoršení kvality vyvíjeného produktu, ale též k nárůstu nákladů na projekt a zdlouhavější realizaci. Byla také zjištěna přímá souvislost mezi efektivitou testování a tím, kdy se testování objeví v procesu vývoje. Testování by mělo být do procesu vývoje zapojeno již ve fázi plánování, aby bylo možné odhalit chyby ve specifikacích a požadavcích ještě před jejich implementací do kódu.

Důležitá je rovněž automatizace procesu testování, která pomáhá snížit počet manuálních činností, čímž snižuje riziko dalších chyb způsobených lidským faktorem. Je důležité poznamenat, že automatické testování zvyšuje rychlost provádění testů a zároveň napomáhá efektivitě celého vývojového procesu. Zároveň nelze přehlížet, že ne všechny testy se dají efektivně automatizovat. Existuje mnoho faktorů, jež pomáhají vyhodnotit nutnost automatizace testů. V této práci byla použita pyramida testování, která dokonale odráží souvislost mezi úrovní testování a schopností efektivně automatizovat testy. Bylo zjištěno, že nejprve je třeba automatizovat jednotlivé a komponentní testy. Tvorba automatizovaných testů pro tyto úrovně je snadná a provádění testů je rychlé.

Při zpracování teoretické části byly analyzovány nejpoužívanější metodiky využívané k řízení IT projektů, a to: agile, DevOps, CI/CD a Continuous Testing. Specifika budování cloud native aplikací vyžadují velké množství testování. Pro zjednodušení tohoto úkolu je nejvhodnější koncept průběžného testování. Průběžné testování znamená provádění automatizovaných testů při každé změně kódu. Odhalení chyb v rané fázi vývoje šetří mnoho času a zdrojů týmu, protože čím později dojde k odhalení chyby, tím obtížnější a nákladnější

je její oprava. Z hlediska zrychlení pomáhá automatizace CI/CD výrazně optimalizovat všechny rutinní procesy sestavování.

Cílem projektu, který je analyzován ve vlastní části, je vývoj cloud native mobilní aplikace, založené na architektuře mikroslužeb. Projekt intenzivně využívá metodiku průběžné integrace a nasazení, která pomáhá urychlit proces doručování vyvinutých komponent zákazníkovi. Tým používá metodiku SCRUM, která rovněž urychluje proces vývoje, a to prostřednictvím menších, ale častých nasazení, což vede k vyšší spokojenosti zákazníků a rychlejší zpětné vazbě. V rámci znalostí získaných literární rešerší lze konstatovat, že použité techniky jsou pro tvorbu cloud native aplikace nejefektivnější.

Proces automatizovaného testování zatím nebyl integrován do celkového procesu CI/CD. V současné době probíhá plánování této fáze. V praktické části byl na základě metodiky průběžného testování vytvořen model procesu automatizovaného testování. Použití tohoto modelu pomůže urychlit proces přípravy samotných testů prostřednictvím automatické kontroly kódu, kterou zajišťuje Azure Pipelines. V rámci pipeline lze rovněž nastavit triggery pro automatické spouštění testů, což zlepší proces testování snížením počtu manuálních činností. Toto řešení také umožní nalézt víc chyb tím, že zvýší počet spuštěných testů. Je třeba také poznamenat, že průběžné automatizované testování v cloudové službě přináší další výhody, jako je snadná spolupráce, snadná mobilní dostupnost a okamžité reporty o provedených testech.

Závěr

Tato práce se zaměřuje na jedno z nejaktuálnějších témat moderního světa IT. V rámci studia odborné literatury byl prostudován koncept cloud native a základ tohoto přístupu – architektura mikroslužeb. Každá služba běží ve vlastním procesu a implementuje konkrétní business funkce. Každou mikroslužbu lze nasadit, škálovat, upgradovat nebo restartovat nezávisle na ostatních službách. To znamená, že zákazníci nejsou při aktualizaci aplikací obtěžováni. V rámci této studie byly identifikovány nejen výhody, ale také problémy, se kterými se společnosti využívající cloud native a mikroslužby mohou v praxi setkat.

Další fází bylo prostudování procesu testování, jeho role v procesu vývoje, ale především jeho automatizace. Role testování se často podceňuje a do procesu vývoje je zařazována až v závěrečných fázích, což vede ke zvýšení nákladů na opravu chyb, prodloužení doby vývoje a také ke snížení kvality vyvíjeného softwaru. Pro řešení tohoto problému byly zkoumány různé metodiky, které pomáhají efektivněji vybudovat proces vývoje-testování-údržby. Na základě vlastností vývoje cloud native aplikací je nejefektivnější metodikou CI/CD a Continuous Testing, která pomáhá začlenit proces testování do pipeline průběžné integrace a dodávky. Budování procesu založeného na Continuous Testing je účinné pouze v případě, že se aktivně využívá automatizovaného testování. Automatizované testování pomáhá odhalit chyby a další problémy v nejranějších fázích vývoje. Při standardním přístupu k procesu vývoje je to velmi obtížné.

V praktické části byl analyzován jeden z projektů společnosti, jejímž zaměstnancem je autorka práce. V rámci tohoto projektu se vyvíjí cloud native mobilní aplikace. Projekt využívá metodiku SCRUM pro řízení týmu a proces vývoje je založen na CI/CD. Pro začlenění procesu automatizovaného testování do stávajícího CI/CD pipeline byl vytvořen BPMN model založený na metodice Continuous Testing. Hlavní výhodou diagramů BPMN je snadná pochopitelnost. Notace popisuje procesy ve formě přístupné všem účastníkům projektu. Rozumí jí nejen vývojový tým, ale i zákazník. Model procesu automatizovaného testování byl vytvořen na základě stávající architektury, která umožňuje využití simulovaného procesu v praxi. V kapitole „Výsledky a diskuse” byly formulovány výhody použití navrhovaného modelu. Všechny cíle této práce byly splněny.

5 Seznam použitých zdrojů

[1] Arundel J., Domingus J. Cloud Native DevOps with Kubernetes: building, deploying, and scaling modern applications in the Cloud. – O'Reilly Media, 2019. ISBN: 978-14-920-4071-2.

[2] The Cloud Native Computing Foundation. *The Cloud Native Computing Foundation* [online]. [cit. 2022-11-20]. Dostupné z: https://www.cncf.io/about/who-we-are/

[3] Newman S. Building microservices. – O'Reilly Media, 2016. ISBN: 978-5-496-02011-4.

[4] ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. *Řízení kvality softwaru: průvodce testováním*. Brno: Computer Press, 2013. ISBN 978-80-251-3816-8.

[5] ISO/IEC 25010 Software and Data quality. *ISO 25000* [online]. [cit. 2022-11-25]. Dostupné z: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[6] SVOZILOVÁ, Alena. *Projektový management: systémový přístup k řízení projektů*. 3., aktualizované a rozšířené vydání. Praha: Grada Publishing, 2016. Expert (Grada). ISBN 978-80-271-0075-0.

[7] Taylor K., Ravichandran A., Waterhouse P. DevOps for Digital Leaders: Reignite Business with a Modern DevOps-Enabled Software Factory. – Apress, 2016. ISBN: 978-1-4842-1841-9.

[8] Leite, Leonardo, et al. "A survey of DevOps concepts and challenges." *ACM Computing Surveys (CSUR) [online]*, 2019, 1-35 [cit. 2022-12-04]. Dostupné z: doi.org/10.1145/3359981

[9] Kuchel T. et al. Key Challenges with Agile Culture--A Survey among Practitioners //arXiv preprint arXiv:2212.07218. *[online]*, 2022, [cit. 2022-12-04]. Dostupné z: doi.org/10.48550/arXiv.2212.07218

[10] Martin R. C. Clean agile: back to basics. – Pearson Education, 2019. ISBN: 978-0-13-578186-9.

[11] Jorgensen P. C. Software testing: a craftsman's approach. – Auerbach Publications, 2013. ISBN: 978-04-291-8457-4.

[12] Homès B. Fundamentals of software testing. – John Wiley & Sons, 2013. ISBN: 978-1-291-84821-324-1.

[13] Craig R. D., Jaskiel S. P. Systematic software testing. – Artech House, 2002. ISBN: 1-58053-508-9.

[14] Spillner A., Linz T. Software Testing Foundations: A Study Guide for the Certified Tester Exam-Foundation Level-ISTQB® Compliant. – dpunkt. verlag, 2021. ISBN: 978-3-96910-298-5.

[15] Majumdar R. Paul Ammann and Jeff Offutt Introduction to Software Testing. Cambridge University Press, 2008. ISBN: 978-0-521-88038.

[16] ISTQB Glossary. *ISTQB Glossary* [online]. [cit. 2023-01-11]. Dostupné z: https://istqb-glossary.page/

[17] Dustin E., Rashka J., Paul J. Automated software testing: Introduction, management, and performance: Introduction, management, and performance. – Addison-Wesley Professional, 1999. ISBN: 0-201-43287-0.
[18] VETTOR, Robert. Architecting Cloud-Native .NET Apps for Azure. Microsoft Developer Division. [online]. [cit. 2023-01-14]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/
[19] BUCHALCEVOVÁ, Alena a Jan KUČERA. Hodnocení metodik vývoje informačních systémů z pohledu testování. Systémová integrace. 2008, 15(2), 13. ISSN 1210-9479.
[20] 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.
[21] Rossberg J. Agile project management with azure DevOps: Concepts, templates, and metrics. – Apress, 2019. ISBN: 978-1-4842-4482-1.
[22] Goericke S. The future of software quality assurance. – Springer Nature, 2020. ISBN: 978-3-030-29509-7.
[23] 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.
[24] RICHARDSON, Chris. Microservices Patterns with examples in Java. United States: Manning Publications, 2019. ISBN: 978-5-4461-0996-8.
[25] SVOZILOVÁ, A. Zlepšování podnikových procesů. Praha: Grada, 2011. ISBN 978-80-247-3938-0.
[26] *State of Continuous Delivery Report June 2021* [online]. The LINUX Foundation Projects, 2021 [cit. 2023-02-05]. Dostupné z: https://cd.foundation/state-cd-report/
[27] *How to get the most out of your CI/CD Workflow using automated testing* [online]. WHITE PAPER, 2021 [cit. 2023-02-07]. Dostupné z: https://az184419.vo.msecnd.net/sauce-labs/white-papers/how-to-get-the-most-out-of-your-ci-cd-workflow-using-automated-testing.pdf
[28] *CI/CD: Continuous Integration and Continuous Delivery* [online]. [cit. 2023-02-11]. Dostupné z: https://www.geeksforgeeks.org/ci-cd-continuous-integration-and-continuous-delivery/
[29] *Continuous Testing* [online]. Synopsys [cit. 2023-02-14]. Dostupné z: https://www.synopsys.com/glossary/what-is-continuous-testing.html
[30] *What is continuous testing?* [online]. IBM [cit. 2023-02-14]. Dostupné z: https://www.ibm.com/topics/continuous-testing
[31] *What is continuous testing?* [online]. Global App Testing [cit. 2023-02-14]. Dostupné z: https://www.globalapptesting.com/blog/what-is-continuous-testing
[32] *Cloud Native Interactive Landscape* [online]. CNCF [cit. 2023-02-15]. Dostupné z: https://landscape.cncf.io/
[33] *Návrh architektury mikroslužeb* [online]. Microsoft [cit. 2023-02-15]. Dostupné z: https://learn.microsoft.com/cs-cz/azure/architecture/microservices/

[34] STRASSER, Martin. *Predicting Performance Degradations of Microservice Applications*. 2020. Master Thesis. Julius-Maximilians-Universität Würzburg. Vedoucí práce Johannes Grohmann M.Sc.
[35] *Microservices Use Cases* [online]. Alpacked [cit. 2023-02-17]. Dostupné z: https://alpacked.io/blog/microservices-use-cases/
[36] *Introduction to hybrid and multicloud* [online]. Microsoft [cit. 2023-02-17]. Dostupné z: https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/scenarios/hybrid/
[37] *Advantages and Disadvantages of Microservices Architecture* [online]. Cloud Academy [cit. 2023-02-18]. Dostupné z: https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/
[38] *The Hidden Costs of Microservices* [online]. Stack Builders [cit. 2023-02-18]. Dostupné z: https://www.stackbuilders.com/blog/the-hidden-costs-of-microservices/
[39] *Career in IT: the role of the Scrum Master* [online]. DOU [cit. 2023-02-18]. Dostupné z: https://dou.ua/lenta/articles/scrum-master-position/
[40] *Continuous Testing in CI/CD* [online]. Habr [cit. 2023-02-18]. Dostupné z: https://habr.com/ru/company/southbridge/blog/670422/
[41] *5 principles for cloud-native architecture - what it is and how to master it* [online]. Google [cit. 2023-02-18]. Dostupné z: https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it
[42] *Cloud-Native Architecture: A Guide, Definitions, Types & More* [online]. Okta [cit. 2023-02-19]. Dostupné z: https://www.okta.com/identity-101/what-is-cloud-native-architecture/
[43] *6 Things You Need to Know About Cloud Native Applications* [online]. Weave [cit. 2023-02-19]. Dostupné z: https://www.weave.works/technologies/going-cloud-native-6-essential-things-you-need-to-know/
[44] *The role of microservices in cloud-native applications* [online]. INSIGHTS [cit. 2023-02-20]. Dostupné z: https://www.ust.com/en/insights/the-role-of-microservices-in-cloud-native-applications
[45] *Microservices in the enterprise, 2021: Real benefits, worth the challenges: Results from a survey conducted by IBM Market Development & Insights* [online]. IBM [cit. 2023-02-20]. Dostupné z: https://www.ibm.com/downloads/cas/OQG4AJAM
[46] *O Adastře* [online]. Adastra [cit. 2023-02-22]. Dostupné z: https://adastra.digital/cs/o-adastre/
[47] BPMN [online]. BPMN [cit. 2023-02-22]. Dostupné z: https://www.bpmn.org/
[48] Draw.io [online]. Dostupné také z: https://app.diagrams.net/

6 Seznam obrázků, tabulek a zkratek

6.1 Seznam obrázků

Obrázek 1- Základy cloud native [18] 14
Obrázek 2 - Využití mikroslužeb v praxi [35] 17
Obrázek 3 - Porovnání monolitické architektury a architektury mikroslužeb [7] 18
Obrázek 4 - Potíže spojené s používáním architektury mikroslužeb [45] 20
Obrázek 5 - Model kvality produktu podle normy ISO/IEC 25010 [5] 26
Obrázek 6 - Diagram procesu řízení kvality [6] 27
Obrázek 7 - Koncept kvality projektu dopady na jeho cenu [6] 27
Obrázek 8 - Aktivity v oblasti vývoje softwaru a úrovně testování (V-model) [15] 29
Obrázek 9 - Metodika životního cyklu automatizovaného testování [17] 34
Obrázek 10 - Agile-strategie automatizace testování [23] 35
Obrázek 11 - Testovací pyramida [24] 36
Obrázek 12 - Použití dubléru k testování systému v izolaci [24] 37
Obrázek 13 - Životní cyklus metodiky Waterfall [11] 38
Obrázek 14 - Životní cyklus metodiky V-model [11] 38
Obrázek 15 - Obecný životní cyklus metodiky agile [11] 40
Obrázek 16 - Model klíčových výzev souvisejících s metodikou agile [9] 41
Obrázek 17 - Životní cyklus metodiky SCRUM [11] 42
Obrázek 18 - Role SCRUM mastera v projektu [39] 43
Obrázek 19 - Koncepční mapa DevOps [8] 44
Obrázek 20 - Diagram procesu CI [28] 47
Obrázek 21 - Proces Continuous Delivery [28] 48
Obrázek 22 - Proces Continuous Deployment [28] 48
Obrázek 23 - Continuous Testing v procesu agile [29] 49
Obrázek 24 - Schéma implementace průběžného testování v praxi [31] 50
Obrázek 25 - Adastra ve světě [46] 52
Obrázek 26 - Proces a metodika SCRUM [zdroj: autorka] 55
Obrázek 27 - Infrastruktura testovacího prostředí [zdroj: autorka] 57
Obrázek 28 - Proces nasazení [zdroj: autorka] 59
Obrázek 29 - Proces vytváření automatického testu za použití Azure Pipelines [zdroj: autorka] 60
Obrázek 30 - Proces provádění automatizovaného testování pomocí konceptu CI/CD [zdroj: autorka] 61

6.2 Seznam tabulek

Tabulka 1 - Klíčové komponenty BPMN, které se používají k vytvoření modelu podnikového procesu [47] 13
Tabulka 2 - Obecné vlastnosti cloud native aplikací [1] 15
Tabulka 3 - Výhody modelu mikroslužeb [3] [44] 19
Tabulka 4 - Potíže spojené s používáním architektury mikroslužeb [34] [36] [37] [38] [45] 22
Tabulka 5 - Cloud native technologie [32] 24
Tabulka 6 - Přehled testovacích artefaktů podle standardu IEEE-829 [13] 28
Tabulka 7 - Úrovně testování [15] 30

Tabulka 8 - Poddruhy funkčního a nefunkčního testování [13] [14].....32
Tabulka 9 - Typy prostředí a jejich atributy [20].....33
Tabulka 10 - Kategorie účastníků koncepční mapy DevOps [8].....45
Tabulka 11 - Metodiky průběžného testování [30].....51
Tabulka 12 - Členové týmu [zdroj: autorka] 54
Tabulka 13 - Popis služeb Azure DevOps Services [zdroj: autorka] 58

6.3 Seznam použitých zkratek

BPMN	Business Process Model and Notation
BPD	Business Process Diagram
CI	Continuous Integration
CD	Continuous Delivery
CT	Continuous Testing
CNCF	Cloud Native Computing Foundation
QA	Quality assurance
UI/UX	User Interface/User Experience
PM	Project manager
MVP	Minimum viable product
ISO	International Organization for Standardization
GUI	Graphical User Interface
API	Application Programming Interface
DoD	Definition of Done
ESG	Environmental, social, and governance