Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Testování aplikací v cloudu pro malé a střední projekty

Diplomová práce

Autor: Bc. Alexandra Ivakhnova
Studijní obor: Informační management

Vedoucí práce: Ing. Martina Husáková, Ph.D.

Hradec Králové duben 2021

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracovala samostatně a s použitím uvedené literatury.

V Hradci Králové dne 29.4.2021 Bc. Alexandra Ivakhnova

Poděkování:

Děkuji vedoucí diplomové práce Ing. Martině Husákové, Ph. D. za metodické vedení práce a věnovaný čas.

Anotace

Tato diplomová práce se zabývá možnostmi testování malých a středních projektů s pomocí cloudových technologií. Teoretická část práce se věnuje analýze relevantních literárních zdrojů za účelem poskytnutí širokého přehledu o obecných typech testování a postupech při zajištění kontroly kvality aplikací pomocí moderních přístupů softwarového vývoje. Velká pozornost je věnována vysvětlení firemních procesů při uplatnění agilních metodik a také významu pečlivého a neustálého testování produktu. Práce zdůrazňuje důležitost zavedení koncepcí CI/CD a obsahuje přehled vybraných nástrojů, nápomocných při vývoji. Za účelem praktické ukázky způsobů uplatnění cloudových technologií v testování byla provedena analýza ukázkového projektu. Závěr práce je věnován porovnání nákladů na provoz lokální a cloudové testovací infrastruktury.

Annotation

Title: Software testing in the cloud for small and larger projects

This diploma thesis research addresses the possibilities of testing small and medium-sized projects with the help of cloud technologies. The first logical part is devoted to the analysis of relevant sources in order to provide a broad overview of the general types of testing and procedures that could be used for ensuring the quality control of applications, using modern software development approaches. The aim of the second logic part is to explain the company processes using agile methodologies and emphasize the importance of careful and continuous product testing. The work points out the importance of introducing the concepts of CI / CD and contains an overview of selected tools that help in the development. In order to demonstrate practical ways of applying cloud technologies in testing, analysis of an example project is shown. At the end of the work is the demonstration of the comparison of the costs of running a local and cloud testing infrastructure.

Obsah

Seznam obrázků

Seznam tabulek

Seznam grafů

1 Úvod

Život v moderním světě je nemožné si představit bez počítačů a jejich programového vybavení. Vývojáři vytváří software pro všechny životní případy a s každým rokem počet počítačových programů a mobilních aplikací roste. Existující software se vyvíjí v čase, dostává novou funkcionalitu nebo se zbavuje neaktuálních možností, vylepšuje se, aby udržel svého uživatele a získal nového. Je skoro nemožné spočítat množství všech vytvořených programů alespoň za rok, ale je možné se podívat na částečné statistiky. App Store v lednu roku 2017 nabízel ke stáhnutí svým zákazníkům 2 200 000 aplikací třetích stran [1]. Kromě toho, za posledních několik let vzrostl počet vývojářů. V roce 2018 mluvíme o 23 milionech programátorů [2], zatímco v roce 2014 číslo sahalo k hodnotě 11 milionů [3]. Společně s těmito jevy a obecným rozšířením počítačových technologií cena vývoje u relativně jednoduchých programů klesá, a naopak roste u složitějších a komplexnějších řešení. Aplikace je produktem, od kterého uživatelé očekávají nejenom dostačující funkcionalitu, ale také kvalitu, která většinou znamená, že program odpovídá deklarovaným charakteristikám. Jednoduchý příklad: závažná odstávka bankovní aplikace může vést ke ztrátě koncového klienta. Ještě horší stav může nastat v případě úniku citlivých informací. Tato rizika vyžadují před vydáním programu do světa důkladné otestování funkcionality a tomuto účelu napomáhá zavedení automatického testování.

Proces testování programu se radikálně mění podle toho, zda mluvíme o vodopádovém přístupu či agilním vývoji, i když postupy jsou shodné. V prvním případě se program musí kompletně ověřit před vydáním, ale pokud mluvíme o agilním vývoji, testování probíhá v mnohem stresovějším a náročnějším režimu, jelikož se celá aplikace, případně infrastruktura, mění u testera přímo pod rukou. Další komplikace spočívá v nerovnoměrné zátěži na testovací stroje. Lokální infrastruktura může selhat v podmínkách kritického zatížení. Na pomoc přicházejí moderní cloudové technologie. Vysoká míra flexibility a schopnost se rychle přizpůsobit prostředí přináší uživatelům cloudu řešení problémů spojených se statickou testovací infrastrukturou. Následující kapitoly se budou zabývat

testováním softwarových aplikací a možnostmi uplatnění cloudových technologií v procesech zajištění kvality.

2 Cíl práce a metodika zpracování

Cílem diplomové práce je provést analýzu možností uplatnění cloudových technologií v testování softwarových aplikací. Součástí práce jsou literární rešerše a praktická ukázka možného řešení. Vzhledem k rozmanitosti vybrané problematiky jsou stanoveny konkrétnější úkoly:

1. vysvětlit základní principy a typy testování softwarových aplikací,
2. naznačit zařazení aktivit kontroly kvality do firemních procesů a uvnitř vývojového týmu, který používá agilní metodiky vývoje,
3. provést analýzu možností podpory testování pomocí cloudových technologií,
4. stanovit odhad finančních nákladů na testování s ohledem na různé strategie realizace a odhalit nejvhodnější řešení z finančního hlediska.

Vzhledem k tomu, že analýza nákladů není možná bez vazby na specifické požadavky projektu, v uvedených příkladech budou použita data, zjištěná ve spolupráci se zaměstnanci společnosti Quadient Ltd. (Quadient Technologies Czech s.r.o.) z pobočky v Hradci Králové.

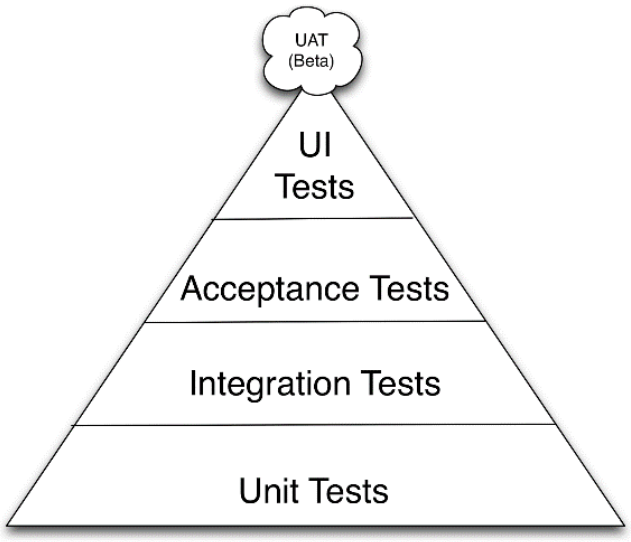
3 Obecné vlastnosti testování

3.1 Rozdělení rolí uvnitř vývojářského týmu

Kvalita počítačových programů je hlavní podmínkou konkurenceschopnosti, a proto proces analýzy celých aplikací nebo jejich určitých částí za účelem nalezení chyb není vhodné podceňovat. Je dosti komplikované stanovit nutný počet testerů, který by dokázal pokrýt všechny možné chyby a zranitelnosti, ale existuje osvědčená praxe, jejíž vhodnost je ověřena mnoha firmami.

Jedná se o určité složení vývojářského týmu, ve kterém na pět vývojářů připadá alespoň jeden tester [4]. Každá firma, dokonce i každý samostatný projekt uvnitř firmy, by měla mít možnost stanovit svůj poměr, vycházející ze specifiky vývoje. Nicméně, přenechání velkého množství práce na jednoho pracovníka a zanedbání lidského faktoru je spojeno s velkým rizikem, že týmový výsledek s časem začne postrádat kvalitu.

Je chybou říci, že proces testování leží zcela jenom na ramenech testera. Kontrolu kvality kódu začíná ještě vývojář před předáním verze na otestování. Nicméně programátor a tester většinou testují různé roviny jednoho programu: zatímco vývojář kontroluje malou a zcela oddělenou část kódu, aby se ujistil, že tato část je funkční, tester zjišťuje, zda celá aplikace správně funguje jako celek a plní uživatelské scénáře použití. Tyto roviny testování softwaru na různých hloubkách se klasicky znázorňují pomocí takzvané testovací pyramidy, která je představena na obrázku č. 1 a připomíná Maslowovu pyramidu potřeb.



Obrázek 1 Testovací pyramida
Zdroj: přístup z Internetu: https://dzone.com/articles/testing-triangle-circle-and

Podle obrázku č. 1 lze soudit, že testování se skládá z:

1. unitových testů,
2. integračních testů,
3. akceptačních testů,
4. testů uživatelského rozhraní.

Zvláštní složku představuje beta testování, které častokrát probíhá se zapojením koncových uživatelů programů.

Tato struktura má tvar pyramidy ne náhodou. Podobný způsob znázornění ukazuje na to, že by množství testů s každou další úrovní mělo klesat, přičemž základní vrstva, jak už bylo řečeno výše, je kontrolována vývojářem, zatímco další tři má na starosti tester. Existují další variace testovací pyramidy, kde se přidávají nebo odebírají složky, ale základní myšlenka a prvky zůstávají neměnné. Bez naplnění předchozí vrstvy by se nemělo pracovat s další a vždycky by se měly vyčleňovat unitové, integrační a případně systémové testy spolu s testy uživatelského rozhrání [5]. Detailnější rozbor, věnovaný různým typům testů, následuje v další kapitole.

V roce 2002 americký programátor Kent Beck ve své knize *Test-Driven Development: By Example* poprvé zmiňuje techniku, která dostala název

programování řízené testy neboli *TDD (Test-Driven Development)* [6]. Základním doporučením autora je nikdy nepsat ani řádek kódu bez unitového testu [7]. Podle Kenta Becka by proces programování měl mít tyto rysy: [8]

1. vývoj probíhá v krátkých cyklech,
2. unitový test se tvoří ještě před implementací,
3. jeden unitový test ověřuje jeden kus kódu,
4. implementace je maximálně jednoduchá a
5. po průchodu testy kód probíhá refaktorováním.

Technika vyvinutá Kentem Beckem dodnes vzbuzuje v programátorském světě žhavou diskuzi o tom, zda je programování řízené testy opravdu schopno zabránit chybám nebo je jenom ztrátou času a peněz. TDD předpokládá již ve fázi psaní kódu neustálé vylepšování provedené práce a tím prodlužování času pro implementaci. Nicméně, podobný přístup je schopný výrazně usnadnit práci testera, který bude mít více možností se soustředit na systémové chování aplikace. Kromě toho, TDD je schopno pomoci předejít jednoduchým, ale spolu s tím snadno přehlédnutelným chybám. Jako každý dobrý nápad, programování řízené testy se má používat moudře a nepovažovat se za samoúčel, který se z nápomocného nástroje stává brzdícím zatížením a jen dobře vypadá v seznamu používaných technologií.

Práce vývojáře a testera se liší nejenom prováděnou činností, ale také i pohledem na výsledek. Programátor, co například naimplementoval novou funkcionalitu potvrzuje, že aplikace umí fungovat v souladu s požadavky. Tester, který provádí svoji analýzu a v podstatě dohlíží nad kvalitou vývojářem odvedené práce, prosazuje opravu nalezených chyb a ujišťuje vedení, že se nová funkcionalita chová tak, jak má. Na první pohled se může zdát, že programátor a tester jsou dvě válčící strany, ale ve skutečnosti mají společný cíl, tj. tvorbu kvalitního softwaru. Nicméně situace, kdy tyto dvě strany mají odlišné názory na stejné chování, vůbec není tak vzácná. V tomto případě komunikace a sdílení znalostí stojí na prvním místě. V ideálním světě by si tester a vývojář měli navzájem naslouchat a neustále se rozvíjet v profesionální oblasti.

Může být tester vývojářem a vývojář naopak testerem? Může. Se zvýšením komplexity a rozsahu počítačových systémů razantně roste množství testů, které je

nemožné opakovaně procházet ručně bez enormního množství testerů. Na pomoc přichází automatizace procesu testování, kdy se automatizuje každý manuální test, kde je to technologicky možné a zároveň i rentabilní. Podobné usnadnění přináší zvýšení požadavků na testera, který musí umět navrhnout a realizovat automatické testy, jednoduše řečeno – musí umět programovat alespoň na základní úrovni. Technické dovednosti vývojáře se velice ocení při automatizaci procesu testování, zatímco testerská pečlivost a preciznost najdou uplatnění při programování, pokud se bývalý tester rozhodne věnovat čistě vývoji [9].

3.2 Základní pojmy

V této kapitole budou rozebrány některé základní specifické pojmy a techniky, které se používají při vývoji a jsou důležité při testování. Testování a jeho nejlepším praktickým technikám je věnována spousta specializované literatury a odborných článků, kde jsou detailně popsány všechny postupy a termíny, takže tato kapitola se bude zabývat jenom vybranou problematikou.

QA (Quality Assurance)

Tento pojem označuje souhrn všech způsobů monitorování vývoje a metod, které se používají pro zajištění kvality softwaru. Zkratkou QA často označují buď jednoho testera, nebo dokonce celou skupinu testerů, pracujících na projektu. Nicméně, jak vidíme z vysvětlení výše, QA je mnohem širší pojem, který zahrnuje všechny činnosti týkající se kontroly, včetně těch na vyšší úrovni, tj. stanovení cílů, zpracování plánu testování, výběr technologií, monitorování výsledků apod. QA musí splňovat obecné standardy kvality, například ISO 9000 [10, s. 7].

Do širokého pojmu QA spadá *QC (Quality Control),* což je souhrn aktivit, které garantují, že produkt splňuje stanovené požadavky a samotné testování, které se zabývá nalezením nesouladů s deklarovanými vlastnostmi. Testování je běžnou testerskou činností obnášející hledání chyb (angl. *a bug* – chyba) a dohlížení nad tím, že tyto chyby budou odstraněny [11].

Bug report

Jedná se o dokument popisující odchylky od očekáváného stavu programu, které jsou považovány za chyby. Obsahuje co nejúplnější popis kroků a souvislostí, vedoucích k nekorektnímu fungování systému. V některých případech je těžké odhalit všechny detaily, které stojí za vznikem problému, jelikož chyba se může nacházet na zcela nečekaném místě nebo musí nastat výjimečná kombinace faktorů, aby se program začal chovat nesprávně. Bug report by měl poskytnout vývojářům co nejvíce informací, jelikož to napomáhá opravě problému a tím šetří čas a finance. Kromě samotného popisu, jak nasimulovat problém, bug report může obsahovat různá kritéria na základě kterých se posuzuje jak závažný je problém, kolik sil je potřeba alokovat na jeho vyřešení a jak rychle se bug musí opravit. Příklady takových detailů mohou být: priorita a závažnost, které stanovuje tester na základě svého posouzení, které mohou být později upraveny během podrobnější analýzy programátora [10, s. 167].

Severity (závažnost)

Pojem „závažnost“ charakterizuje, jak velký negativní dopad má chyba na chování systému. Obecně můžeme závažnost dělit na několik úrovní:

- menší chyby (*minor bugs*),
- velké chyby (*major bugs*),
- kritické chyby (*critical bugs*).

Management vyvozuje z tohoto rozdělení závěr, že s růstem závažnosti chyby roste množství ztracených peněz v případě, kdyby se tento problém objevil u zákazníka. Takže teoreticky můžeme závažnost přirovnat k určité finanční částce. Je potřeba zdůraznit, že tato částka není přímo propojena s obtížností řešení problému. Oprava může trvat jenom pár minut a být triviální, ale přinese maximální užitek [10, s. 176].

Priority (priorita)

Priorita určuje čas, během kterého nalezená chyba musí být opravena. Firma může stanovit interní nařízení, kde bude určeno časové omezení na opravení chyb, zvlášť externích, které byly nahlášeny klienty. Podobná pravidla se mohou vyskytovat v Service-level agreement (*SLA*) jako závazek dodržení kvality produktu vůči klientům [10, s. 177].

Validace a verifikace

Tyto dva pojmy se často slučují dohromady. Na první pohled je těžké mezi nimi rozeznat rozdíl, jelikož oba znamenají *ověření*. Rozdíl mezi nimi existuje a není tak nepatrný, jak by se mohlo zdát. *Validace* odpovídá na otázku: Děláme *správný* produkt?; *verifikace* říká, zda produkt děláme *správně*. Vidíme, že validace kontroluje účelnost prováděné práce, resp. prověřuje, zda vývoj neprobíhá zbytečně. Verifikace probíhá spíše na nižší úrovni, když se analyzuje kvalita prováděné práce.

Podíváme se na příklad. Testerem nahlášená chyba byla opravena vývojářem a předána zpátky testerovi ke kontrole, zda se tentokrát funkcionalita chová podle předpokladu. Tester se snaží zopakovat postup, který vedl k chybě, a ujišťuje se, že žádná chyba už není. Nicméně, během své analýzy po konzultaci s kolegy se zjistí, že funkcionalita, kterou ověřuje, se už dávno nepoužívá, protože byla nahrazena jinou nebo neodpovídá požadavkům klienta. Proběhla validace a bylo zjištěno, že oprava chyby byla v podstatě zbytečnou ztrátou času. Verifikace ve světě velkého množství chyb je obvyklou testerskou činností, zatímco s validací se řadový tester až tak často nesetkává [12].

Bez revize obecných kroků, bez validace produktu existuje riziko, že se ocitneme v situaci, kdy se provádí spousta nepotřebných prací, které občas mohou být velmi náročné.

Platform matrix (cross-platform matrix)

Jedná se o souhrnný popis produktem podporovaných platforem, technologií a programů třetích stran. Slouží nejen pro interní účely, ale je také vystaven navenek a je přístupný klientům, kteří na základě tohoto dokumentu posuzují, zda je pro ně produkt vhodný. Platform matrix je možné porovnat s technickými požadavky počítačových her.

Test case (TC, testovací případ)

Pod tímto pojmem se rozumí algoritmus testování programu, tj. scénář, kterým se řídí tester a na základě výsledku provedených kroků usuzuje, zda systém funguje správně a odpovídá požadavkům. Testovací případ je detailní, krok po kroku popsaná posloupnost činností, u které je přesně definované očekáváné chování. Je samozřejmostí, že každý systém se ověřuje v předem určeném prostředí, a proto TC, který vyžaduje změnu tohoto prostředí a nastavení specifických podmínek, obsahuje tyto okolní požadavky. Jelikož je nepravděpodobné, že se najde situace, kdy jeden jediný tester celou dobu pracuje na nějakém projektu sám, všechny podklady by se měly psát jednoduchým způsobem, aby dokonce i úplně nový zaměstnanec dokázal pochopit, co se po něm žádá. TC přispívá k formalizaci testerských postupů. Je návodem k provedení manuálního otestování. Také slouží jako základna pro vytvoření automatického testu [13, s. 37].

Obecná struktura TC vypadá následovně:

Tabulka 1 Struktura testovacího případu

Akce	Očekávaný výsledek	Výsledek kroku

Zdroj: vlastní zpracování

Výsledek testu se klasicky označuje jedním z těchto stavů: [10, s. 119]

- *passed*, což znamená, že systém se chová v souladu s očekáváním,
- *failed*, což je signálem k tomu, že funkcionalita může být narušená,

- *blocked*, oznamující, že z nějakého důvodu nebylo možné provést potřebnou akci, a proto není možné posoudit, zda se program chová tak, jak má.

Kromě toho, jak už bylo řečeno dříve, TC může mít případný popis specifických podkladů a testovací soubory. Měl by mít název a číslo pro snadné dohledání. Ze stejného důvodu je možné rozdělit všechny TC do kategorií na základě toho, co ověřují.

Na základě ověření přihlašování na některý web se podíváme na příklad testovacího případu a jeho výsledky.

Tabulka 2 Příklad testovacího případu

TC č. 1, ověření přihlašovací webové stránky			
Číslo kroku	*Akce*	*Očekávány výsledek*	*Výsledek kroku*
1.	Otevřít stránku pro přihlášení	Přihlašovací stránka se načetla	Failed
2.	Provést pokus o přihlášení na web pomocí platných přihlašovacích údajů: login, heslo	Přihlašování proběhlo v pořádku, otevřela se stránka uživatelského profilu	Blocked

Zdroj: vlastní zpracování

Vidíme, že hned v kroku č. 1 se celý popsaný algoritmus otestování zastavil, jelikož, například se přihlašovací stránka vůbec nenačetla. V důsledku pádu kroku č. 1 není možné provést krok č. 2 z pochopitelných důvodů. V případě, kdyby systém na oba kroky reagoval správným způsobem, bylo by možné označit TC jako bez problému projitý, což by mělo garantovat, že přihlásit se na web jde bez potíží. Nicméně celý TC je od začátku nedostatečně komplexní: „Co by se stalo, kdyby se tester pokusil použít neexistující jméno uživatele?“ „Co by se stalo, kdyby tester zkusil neplatné heslo v páru s validním jménem?“ „Co když se uživatelovi nepodaří přihlásit, například 5x za sebou? Bude takový profil zablokovaný nebo ne?“

Testovací případ musí vycházet ze všeho, co je schopná nabídnout aplikace, a tím pádem by měl ověřit celou funkcionalitu. Podobný přístup vyžaduje od testera opravdové pochopení chování programu a pečlivost při napsání TC.

3.3 Typy testů

Program se dá testovat na různé typy zranitelností a omezení, což umožňuje rozdělení procesu testování a vznik klasifikace testů na základě rozličných vlastností. Je nutno říci, že neexistuje jediný správný způsob, jak kompletně roztřídit testy do určitých skupin, nicméně dají se najít obecně uznávané praktiky. Níže budou představeny základní kategorie a elementy.

3.3.1 Rozdělení procesu testování na základě znalosti vnitřního chování systému

Následující klasifikace vychází z toho, zda tester rozumí vnitřním procesům systému a dokáže predikovat jeho chování na základě těchto znalostí:

- černá skříňka (*black box testing*),
- bílá skříňka (*white box testing*),
- šedá skříňka (*grey box testing*).

V případě, že vnitřní logika aplikace je pro testera neznámou, mluvíme o *testování černé skříňky*. V této situaci testovací případy většinou vycházejí z očekávaných způsobů, jak funkcionalita programu bude použita uživateli. Předpokládá se například, že během nákupu přes online obchod budou klienti používat funkci vyhledávání. Na základě tohoto předpokladu tester sestavuje TC, který bude testovat oznámenou funkcionalitu ryze z uživatelského hlediska, tj. bez znalosti o tom, kde a jak jsou uloženy informace o produktech, jak se tyto informace zobrazují na stránce apod. Jestliže se o to nezajímá uživatel, stejně tak o to nemá zájem i tester.

Dalším způsobem, jak přijít na specifický scénář uživatelského chování, je explorativní testování, které zahrnuje náhodné používání programu. Vezmeme výše zmíněný případ s online obchodem. Tester přidal do košíku nějaký produkt N1, přešel na stránku platby a rozhodl si najít další produkt N2. Forma vyhledávání je dostupná i z košíku a tester ji použije, nicméně místo stránky s výsledkem

vyhledávání dostane chybu číslo 404 – Stránka nenalezena. Podobné chování není přímo očekávané ze strany uživatele, avšak se může objevit. Správně navržená aplikace by měla ukázat interní chybovou stránku a nabídnout možná řešení.

Velmi užitečnými pomocníky při vypracování testovacích případů (a nejenom při testování systému jako černé skříňky) budou intuice a komunikace s dalšími členy vývojářského týmu, kteří se buď dokážou lépe vyznat ve vnitřním světě systému a jeho případných zranitelnostech, nebo budou schopni se podívat na problém z jiného úhlu pohledu.

Jiný přístup k testování je založen na pochopení testerem nejenom výsledku, který program nabízí, ale i toho, jak tohoto výsledku systém dosáhne. V tomto případě mluvíme o *testování bílé skříňky*. Tester ví o tom, že při přidání produktu do košíku online obchodu se odpovídající počet jednotek tohoto produktu rezervuje v externím systému, který se používá jako skladovací a je společný pro kamenný a webový obchod. Tester chce zjistit, zda je možné rezervovat větší počet kusů, než je dostupný, na základě prodlevy obnovení dat na webových stránkách. Typický uživatel nad podobnou situací ani nepřemýšlí, jelikož věří, že programátoři se o všechno postarali. V tomto je rozdíl mezi testováním bílé a černé skříňky. Nicméně oba přístupy mají svoje nevýhody a rizika. Testování systému jako černé skříňky je schopno přehlédnout problémy, které vychází z technické specifikace, zatímco přístup bílé skříňky vyžaduje větší znalosti od testera a zároveň může vyvolat určitý typ slepoty, když se testují jenom složité a neobvyklé případy a ty nejjednodušší a podstatné se přehlíží. Z těchto důvodů vzniká hybrid přístupů, který obnáší jak uživatelsky orientované testování, tak i kontrolu na základě znalosti aplikace – *testování šedé skříňky*.

Jednoduchým příkladem může být kontrola, zda platba v online obchodě proběhla v pořádku. Po zaplacení se má zobrazit stránka, která potvrzuje přijetí platby a ujišťuje uživatele, že objednávka je ve stavu zpracování. Tester se v průběhu testování dostane na tuto stránku, ale místo toho, aby dokončil test, nejdříve se ujistí, jak vypadá daná objednávka v databázi, zda je opravdu ve frontě ke zpracování apod. [10, s. 70-72].

3.3.2 Rozdělení testů na základě objektu testování

Funkční testování

Tento druh testování kontroluje, zda aplikace plní všechny své funkce a dovoluje používat program v souladu s uživatelskými scénáři [10, s. 82].

Testování uživatelského rozhrání (UI testování)

Při provedení testování se kontrolují všechny prvky uživatelského rozhrání, tj. zda je stránka zobrazena korektně, zda při vložení data narození lze použít písmenka, zda zobrazená tlačítka plní svoji funkci a dávají pokyny ke zpracování dat apod. [10, s. 85]

Zátěžové testování

Aplikace s dokonalou funkcionalitou může být zcela nepoužitelná z důvodu nízké rychlosti odezvy nebo neschopnosti podporovat mnohouživatelské použití. Předejít efektu úzkého hrdla je možné pomocí zátěžových testů, které nutí program fungovat ve stresových podmínkách a ukázat problémy dříve, než nastanou během provozu [10, s. 88].

Testování bezpečnosti

Bezpečnost aplikace se dá testovat jak alokací vnitřních sil firmy, tak i pomocí profesionálních etických hackerů, kteří se pokusí napodobit hackerský útok (např. během penetračního testování), a poté všechny zranitelnosti a doporučení sdělí vývojářskému týmu. Nicméně, aspoň základy by se měly testovat uvnitř firmy: zablokovat uživatele po několika špatných pokusech přihlášení, zamezit registraci nového uživatele s příliš krátkým heslem apod.[10, s. 86]

Uživatelské testování (UX testování)

Tento typ testů často probíhá bez zapojení testerů, jelikož oni už jsou s programem dost seznámeni na to, aby dokázali posoudit, jak moc pohodlně se s ním pracuje. UX testování (angl. *user experience testing* – testování uživatelské zkušenosti) může vypadat takto: skupina osob zcela nespojených s aplikací je pozvána na otestování programu pomocí plnění předem definovaných úkolů a poté jsou vyzvání ohodnotit, jak dobře se jim pracovalo s programem [10, s. 85].

Testování kompatibility

Sem spadá analýza kompatibility programu a zařízení a jiných aplikací. Příkladem může být testování napřič internetovými prohlížeči, což je v současnosti velmi aktuální. Webová aplikace, která správně funguje v Google Chrome poslední verze, se nemusí načíst v Internet Exploreru [10, s. 86].

3.3.3 Rozdělení testů na základě času, kdy testování probíhá

Životní cyklus vývoje softwarového programu má určité fáze. Na konci každého úseku vzniká verze se specifickým označením. *Alpha* verze je určena internímu testování, *beta* verze se poskytuje externím uživatelům za stejným účelem, *release candidate* je kandidátem na vydání, pokud se neobjeví závažná chyba. Na základě toho, zda je software přístupný jenom vývojářskému týmu nebo i uživatelům, se dá testování kategorizovat následovně:

1. před předáním aplikace uživatelům (alpha testování),
2. po předání aplikace uživatelům (beta testování).

Alpha testování se většinou charakterizuje jako testování bílé nebo šedé skříňky, zatímco beta testování je více uživatelsky orientováno a vychází za předpokladu, že klient neví, jak program plní úkoly [13, s. 157].
Testy prováděné testery je možné rozdělit na dalších několik skupin na základě toho, kdy se provádí.

Smoke testy

Tato kategorie testů je určena pro kontrolu základních funkcí programu a zároveň signalizuje, zda je možné přecházet k detailnější analýze. Případ, kdy smoke testy neprochází, indikuje, že aplikace nedokáže naběhnout správným způsobem a není připravena ani pro jednoduché použití. Pád nějakého z těchto testů je důležitým signálem, který nelze ignorovat. Nicméně podobná nepříjemná situace může znamenat nejenom vznik kritické chyby, ale také i jen změnu či chybu v testovacím prostředí, kterému se program neumí automaticky přizpůsobit. Po zjištění příčiny spadlých testů a jejich opravy je potřeba znovu spustit smoke sadu a jenom pokud tentokrát žádný problém nevznikne, je možné přejít k dalšímu testování [14].

Testy, ověřující novou funkcionalitu

Jak vidíme v názvu, tyto testy kontrolují nové funkce přidané do programu. Mluvíme o vytváření a procházení nových testovacích případů, které budou časem přemístěny do kategorie regresních [13, s. 257].

Regresní testy

Regresní analýza spočívá v tom, že všechny testovací případy, které kdysi byly vytvořeny, se procházejí znovu. To, co fungovalo před měsícem, musí fungovat i dnes. Není neobvyklá situace, při které přidání nové funkcionality či vylepšení staré dokáže způsobit značné problémy a rozbít stabilní chování, na které se spoléhá [15].

3.3.4 Rozdělení testů na základě testovací vrstvy

Unitové testy, které začínají dlouhý proces testování, již byly zmíněny dříve. Teď se na ně podíváme blíž a také na další testy, existující na různých úrovních.

Unitové testy

Spočívají v testování základních komponent, ze kterých se skládá software. Unitové testy jsou automatické a mnohem kratší než všechny další vzhledem k velikosti testovacích modulů. Příkladem unitového testování může sloužit kontrola varování při vložení nepovolených znaků do pole na webové stránce. Unitové testy píšou programátoři při vývoji a tím kontrolují sami sebe na základní úrovni. Otestování komponent je rychlým způsobem, jak odhalit jednoduché chyby uvnitř samostatných prvků [16, s. 112].

Mocným nástrojem při vytváření testů jednotek pro ověření aplikací, napsaných v jazyce Java, je framework JUnit, vyvinutý Kentem Beckem ve spolupráci s Erichem Gamma. Tento aplikační rámec vyžaduje použití specifické anotace umožňující spouštět metody v určitý čas, např. před nebo po provedení testu [17].

Integrační testy

Po tom, jak byly otestovány komponenty zvlášť, je čas ověřit jejich vzájemné propojení a korektní spolupráci. Cílem integračního testování je najít defekty a nežádoucí chování, které je spojeno s chybami v interpretaci a realizaci vzájemného působení mezi jednotkami [16, s. 111].

Systémové testy

Jak můžeme vidět z názvu, systémové testy kontrolují chování programu jako celku: v této etapě se zjišťuje, zda aplikace nabývá požadované vlastnosti a plní své funkce. Testovací případy vychází z uživatelských scénářů [16, s. 111].

Uživatelské akceptační testy

Po tom, jak vývojářský tým dokončí testování a tím zaručí, že aplikace splňuje požadavky, a je připravena k použití, mohou proběhnout uživatelské akceptační testy. Je to poslední přejímací zkouška, kdy budoucí uživatelé mají možnost si

software vyzkoušet, pokusit se ho nastavit v souladu se svými potřebami a případně poukázat na chyby nebo nápady na vylepšení. Podobné vyzkoušení je relevantní v případě, kdy byla aplikace vyvinuta na zakázku pro nějakého určitého zákazníka. Na základě uzavřené smlouvy se vývojářský tým zavazuje vzít v úvahu sdílené poznámky [18].

3.3.5 Rozdělení testů na základě míry automatizace

Manuální testy

Nejjednodušší, ale zároveň nejpomalejší typ testování, který nevyžaduje od testera hlubokých znalosti v oblasti programování a automatizace. Tester buď používá scénáře testovacích případů, nebo se zabývá explorativním testováním, ale jedině manuální cestou, např. otevře aplikaci, vloží testovací údaje, zmáčkne potvrzující tlačítko apod. Výhodou je, že si ve většině případů s tímto typem práce poradí i člověk, vzdálený od informačních technologií [13, s. 166].

Automatické testy

V průběhu testování se tester setkává s velkým množstvím stále se opakujících neobyčejně jednoduchých kroků, které vyžadují jedině trpělivost a pečlivost, například schopnost přes webové rozhrání zaregistrovat nového uživatele knihovny. I když podobné testy jsou jednoduché, mohou selhat kvůli lidskému faktoru, který není možné vyloučit. Pro zjednodušení a zlevnění celého procesu testování přicházejí na pomoc automatické testy, které jsou v podstatě samostatnými programy sestavené testery a testující hlavní produkt místo nich. Automatizace je schopná významně zkrátit čas testování, ale zároveň přináší vyšší požadavky na programátorské a technické dovednosti testera [19, s. 4].

Poloautomatické testy

V případě, když část testů je zautomatizována a část se dělá testerem manuálně, mluvíme o poloautomatickém testování [13, s. 168].

Je potřeba zmínit, že při vší rozmanitosti testů je skoro nemožné zabezpečit pokrytí všech možných testovacích případů jednak kvůli účelnosti, zadruhé kvůli ještě větší mnohotvárnosti a složitosti současných informačních systémů a prostředí, ve kterém se tyto systémy používají.

4 Automatické testování

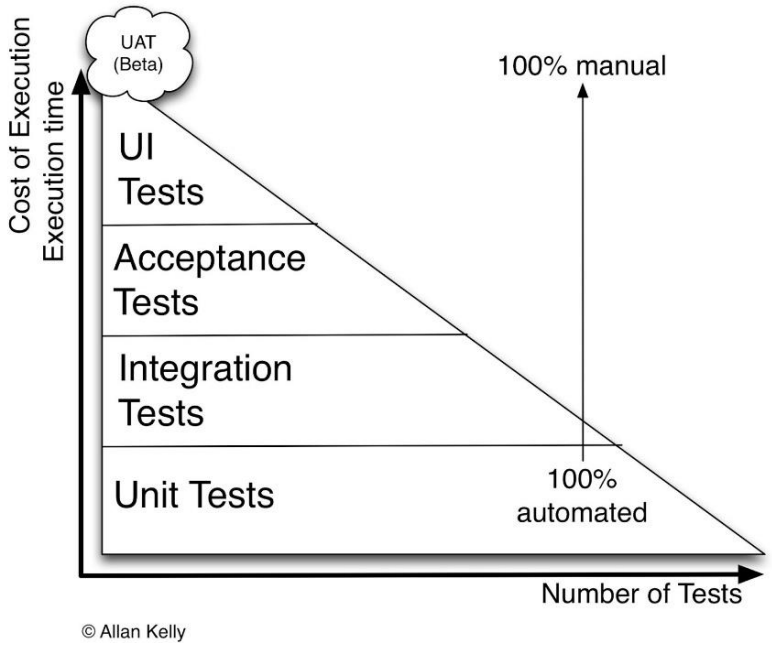
4.1 Míra a význam automatizace

Jak už bylo řečeno výše, přechod na automatické testování může značně usnadnit a zrychlit proces hledání bugů, jednoduše řečeno vylepšit ho a zlevnit. Mezi manuálním a automatickým testováním platí vazba, že se s růstem automatizace se snižuje čas potřebný na procházení testů. Testování na nejnižší, základní úrovni by se mělo automatizovat bez jakýchkoliv úvah. Aktivity jako např. rutinní ruční kontrola, kolik písmenek se vejde do všech polí, spotřebuje značnou část testerské kapacity. Přesně z tohoto důvodu se musí v ideálním světě unitové testy široce používat a mít automatickou podobu, aby nechaly prostor na hledání mnohem složitějších a neobvyklejších bugů [20, s. 251].

Integrační a systémové testy a testy uživatelského rozhrání je možné automatizovat jenom do určité míry, jelikož mohou vyžadovat provedení komplikovanějších úkolů, které je jednodušší a bezpečnější provést manuálně. Projev některých chyb může potřebovat pro své odhalení velmi specifické podmínky, které není možné předem odvodit a definovat v automatickém testu, a proto je těžké úplně eliminovat ruční testování. Každý projekt je jedinečný a jenom jeho specifika a schopnosti vývojářského týmu určují míru automatizace testovacího procesu.

Přejímací zkoušky, beta a uživatelské testování jsou manuální z pochopitelných příčin. Do procesu se zapojují lidé, kteří nemusí mít souvislost s vývojem [10, s. 258].

Popsaná situace je nejlépe znázorněna na schématu níže, kde můžeme vidět závislost mezi množstvím a úrovní automatizace testů na jejich typech. Zároveň vzniká představa o tom, kolik času a samotných testů která vrstva vyžaduje.



Obrázek 2 Struktura závislostí testů
Zdroj: Alan Kelly, přístup z Internetu: https://dzone.com/articles/testing-triangle-circle-and

Z obrázku č. 2 lze soudit, že s nárůstem složitosti testů jejich množství postupně klesá, stejně jako i míra automatizace. Nicméně komplikovanější testovací případy potřebují více času na realizaci.

Kromě značných výhod, které přináší automatické testování, vznikají i nové komplikace. Na to, aby tester dokázal napsat testovací případ, nepotřebuje tak hluboké znalosti, ale ne každý dokáže automatický test napsat či nadefinovat pomocí specifických programů. Automatizace vyžaduje nutnost zvýšení kvalifikace testerů a klade důraz na jejich osobní schopnost se rozvíjet v oblasti informačních technologií a programování a také na sdílení znalostí uvnitř týmu a projektu.

Snaha o automatické testování přináší velkou otázku: „Vyplatí se to?“ Napsání testů potřebuje více času, než procházení scénáře testovacího případu v jednom, dvou, třech kolech. Nicméně, poté se vynaložené náklady začínají vracet a šetřit čas a síly pracovníků zajištění kontroly [21, s. 2].

Předpokládejme, že ověření podmíněné funkcionality manuálním způsobem trvá jenom hodinu. Do tohoto času spadají jakékoliv přípravné práce a návrat systému do původního stavu. Napsání či definování automatického testu naopak vyžaduje přibližně dva pracovní dny a poté ještě dvě hodiny doladění a odstranění nestability, což nám dává osmnáct hodin, pokud mluvíme o osmihodinovém pracovním dnu. Na

první pohled zdá se, že investovat čas do automatizace je nerozumné, zvlášť když vezmeme v úvahu, že se během psaní testu mohou vyskytnout neočekávané problémy značně zpomalující vznik automatického testu: od nízkých programovacích znalostí autora testu do úzkých hrdel již používaných technologií [20, s. 256]. Na druhou stranu uvažujme o tom, že rozšíření a vylepšení aplikace s velkou pravděpodobností dokáže ovlivnit již otestovanou funkcionalitu. To znamená, že testovací případ se musí procházet opakovaně – v době aktivního vývoje pravděpodobně i každý den. Předpokládejme, že automatický test trvá šest minut včetně přípravy prostředí a závěrečných činností. Můžeme vytvořit dvě jednoduché rovnice času stráveného na procházení manuálního (t_m) a automatického (t_a) testu v závislosti na iteracích, s tím, že do toho započítáme i čas na automatizaci:

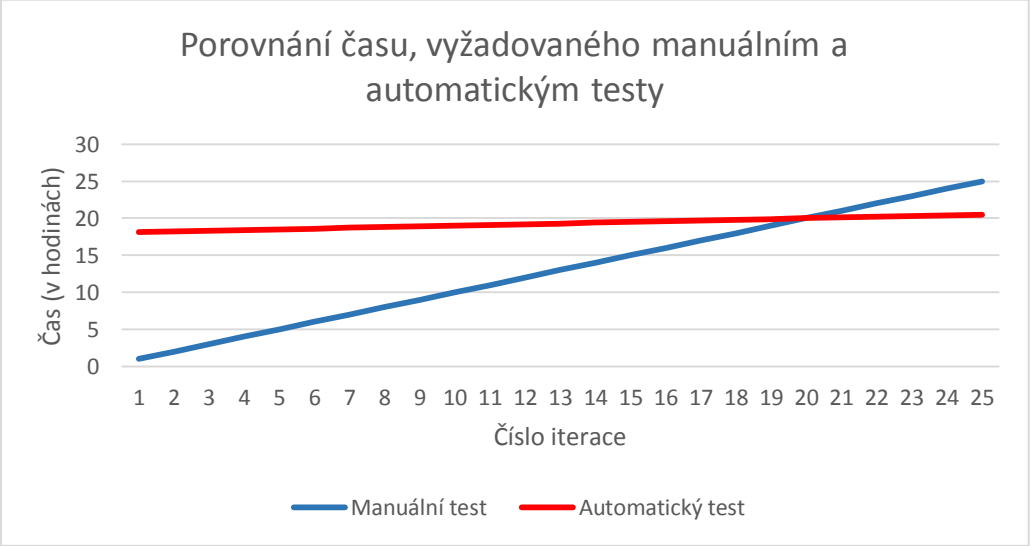
$$t_m = i$$

$$t_a = 18 + 0.1i\,,$$

kde *i* je číselným označením iteračního cyklu.

Graf č. 1 nám dovoluje graficky ocenit časové náklady obou testů.

Graf 1 Porovnání času vyžadovaného manuálním a automatickým testy



Zdroj: vlastní zpracování

Vidíme, že při 20. iteraci se čas investovaný do napsání automatického testu začíná vyplácet a při 25. běhu ušetří již čtyři a půl hodiny. Se vzrůstajícím počtem iterací se čas ušetřený automatizací bude dále zvětšovat. Za předpokladu, že se test spouští každý den, potřebujeme méně jak tři týdny na dosažení bodu zvratu. Takovým způsobem se dá prokázat, že je automatizace opravdu schopná ušetřit čas, síly a také peníze. Nepříjemným důsledkem však může být propouštění manuálních testerů, jelikož ten objem práce, který dříve zaměstnal celou skupinu manuálních testerů, po přepsání testů do automatických dokáže za stejný čas udělat jen jeden člověk.

Za zmínku stojí i možná nestabilita testů nebo technické překážky pro jejich vytváření, které nejde obejít jednoduchou cestu. Testy, které prochází jenom v padesáti procentech případů, nemusí ukazovat na chybu: příčina se může skrývat ve špatném způsobu automatizace, pomalém stroji, zásahům externích programů apod. Podobné výsledky nejsou důvěryhodné a existuje riziko, že buď při jejich pečlivé analýze dojde ke zbytečné ztrátě času, nebo se při ignorování neodhalí závažný problém. Nic neříkající závěry mohou být dosaženy i při manuálním testování, ale v menší míře, jelikož v tomto případě tester kontroluje a sleduje prostředí a je si schopný rychleji všimnout odchylky způsobující problém. U nestabilních testů by se mělo vyhodnotit, zda je vůbec potřeba je procházet, a v případě nezbytné existence těchto testů investovat čas do zvýšení jejich stability.

4.2 Nástroje nápomocné při testování webových aplikací

Tato kapitola se bude zabývat vybranými programy, které mohou být užitečné při automatickém testování webových aplikací. Řeč bude o produktech, které se používají již léta a jejich přínos je tudíž prokázaný.

4.2.1 API testování

Obecně zkrátka API (*Application Programming Interface*) znamená rozhraní, přes které se provádí komunikace na programové úrovni. U webových aplikací můžeme mluvit například o přidání interakční mapy Google Maps na stránku internetového obchodu. Webové API funguje na tom principu, že aplikace A1 posílá požadavek aplikaci A2 a očekává odpověď ve formě určité struktury, naplněné daty. Při tom se

používají formáty XML nebo JSON. Pokud přijatá zpráva neodpovídá nutnému složení, A1 vyhodnocuje takovou odpověď jako odchylku a zobrazí chybu, přičemž dokáže i částečně vysvětlit příčiny neprovedeného požadavku. V současné době nejrozšířenější architekturou API je tak zvaná architektura REST, která zase pochází z protokolu SOAP, který používá jazyk WSDL [22].

Představme si, že aplikace běží na jednom počítači – serveru, který je schopný provádět spoustu různých činností, např. pracovat s databází, odpovídat na požadavky jiných serverů, provádět výpočty apod. Komunikace s podobným zařízením pro člověka může být poněkud komplikovanější, jelikož server není schopný odpovídat zcela srozumitelným způsobem a vytvářet graficky přijatelné odpovědi s obsahem, se kterým se člověku pracuje nejlépe. Tento stroj je postaven a nakonfigurován jenom pro to, aby dokázal plnit své funkce, nehledě na to, zda programátor, tester, uživatel či kdokoliv jiný jednoduše rozumí výsledné zprávě a technické reprezentaci dat. Kromě toho, server musí také provádět komunikace s jinými systémy – posílat a přijímat data ve tvaru, přijatelném pro jinou stranu. Protokol SOAP umožňuje podobnou interakci přes výměnu zpráv v XML formátu, přičemž struktura příkazů a odpovědí musí zcela korespondovat. Pro definování pravidel, kterými se řídí server při komunikaci, se používá WSDL soubor - výčet metod dostupných k vyvolání. Nevýhodou SOAP je velikost zpráv, které mohou být nesmírně dlouhé v závislosti na tom, jaké množství dat se posílá. Výhodou je vyšší bezpečnost díky pevně určeným pravidlům [23].

Jiným způsobem komunikace je REST architektura založená na protokolu HTTP a jednotných identifikátorech zdrojů – URI, které definují cestu k objektům. Výměna dat může probíhat v XML nebo JSON formátech. Příkazy se skládají z několika částí:

- koncového bodu,
- metody,
- hlaviček,
- těla.

Jako koncový bod slouží URI, odkazující na objekt. HTTP metoda, definovaná v příkazu, ukazuje na akci, kterou je potřeba realizovat nad vybraným objektem.

Základními jsou POST, GET, PUT, DELETE, které odpovídají hlavním způsobům interakce s daty v programování:

- create (vytvořit),
- read (načíst),
- update (editovat),
- delete (smazat).

Tyto akce dostaly zkratku CRUD, ale mohou mít odlišné označení na základě konvencí jiných jazyků. Hlavičky realizují výměnu servisních dat mezi serverem a klientem, tělo obsahuje data zprávy. Po zpracování klientského požadavku server posílá odpověď a zároveň stavový kód HTTP, který je rychlým indikátorem, označujícím výsledek interakce [24].

REST architektura vyžaduje několik vlastností od webové aplikace, které byly definované Royem Fieldingem v roce 2000 [25], kdy on poprvé teoreticky shrnul principy komunikace modelu klient-server a pojmenoval vzniklý způsob jako REST.

1. Model klient-server

   Tato architektura přináší rozdělení výpočetní zátěže a úkolů mezi serverem, který poskytuje službu, a klientem, který žádá o provádění operací. Princip separace pomáhá zvýšit přenositelnost kódu a umožňuje škálování.
2. Bezstavovost

   Mezi příkazy, které server dostává od klienta, se žádná informace o jeho stavu neukládá na serveru. Všechny požadavky musí být sestaveny takovým způsobem, aby v sobě nesly všechny potřebné informace pro jejich zpracování.
3. Ukládání dat do mezi-paměti klienta

   Tato vlastnost umožňuje snížit čas nutný pro reprezentaci odpovědi ze strany serveru na straně klienta, a tím se zvyšuje výkonnost systému. Všechna data musí mít označení, zda mohou být uložena do mezi-paměti či ne. Podobný přístup pomáhá eliminovat riziko spojené s tím, že klient bude používat zastaralá data.
4. Jediné rozhraní

Unifikace rozhraní vylepšuje komunikaci a dovoluje jejím účastníkům rozvíjet se bez vzájemné závislosti; na druhou stranu tato vlastnost je možnou příčinou zpomalení odezvy, jelikož každá zpráva musí nabýt standardizované podoby, dokonce i na vrub potřeb samotných aplikací. Přepisování dat z jedné formy do jiné může trvat významný čas. Jediné rozhraní potřebuje splnění několika podmínek:

1) každý objekt nebo zdroj (resource) musí mít vlastní ukazatel a také svoji reprezentaci,
2) všechny příkazy a odpovědi musí mít dostačující data pro jejich zpracování,
3) uplatnění principu hypermedia jako aplikačního stavu (HATEOAS), který znamená, že stav zdroje je popsán v identifikátoru, atributech, tělu zprávy, hlavičkách. Objektem může být všechno, co umíme pojmenovat; jeho reprezentace je implementována pomocí textového popisu; jako identifikátor slouží URI.

5. Vrstevnatost

   V průběhu komunikace se serverem klient neví, zda on posílá a přijímá zprávy přímo od serveru nebo od jiného středního uzlu, což je podmíněno hierarchickou strukturou sítí. Použití mezilehlých bodů je schopné vyvažovat a přerozdělovat zátěž, která dopadá na server; zvyšovat bezpečnost a důvěrnost dat; zajišťovat dálkové ovládání; volat do programů třetích stran apod.

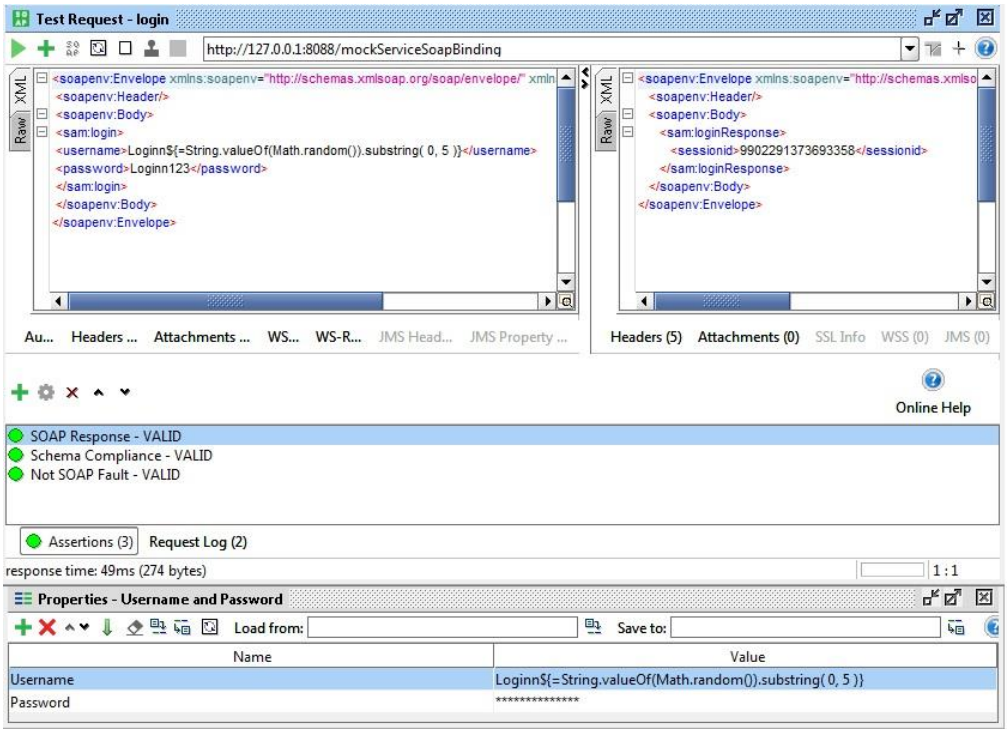
6. Vyplnění kódu na požádání (Code-On-Demand)

   Příkaz, který dostává server od klienta, může, ale nemusí obsahovat kód, který rozšiřuje základní funkcionalitu. Rozšířené požadavky se definují pomocí skriptů v JavaScript. Tato vlastnost je potenciálním zdrojem ohrožení, jelikož může obsahovat příkazy sloužící jako útok.

REST architektura je užitečná tam, kde se na aplikaci kladou vysoké nároky na rychlost odezvy a zpracování dat a také v případě nutnosti škálování při vysokém počtu uživatelů nebo při velkém množství prováděných akcí.

SoapUI

SoapUI je nástrojem, který slouží pro testování aplikací, založených na výměně dat pomocí SOAP, nicméně umí pracovat i s REST architekturou. Hned při instalaci programu jsou přístupné ukázkové testovací projekty, které dovolují vyzkoušet funkcionalitu SoapUI v plné míře za pomoci mock služby. Obrázek č. 3 ukazuje, jakým způsobem může probíhat test ověřující možnost přihlášení.



Obrázek 3 Ukázka testu v SoapUI
Zdroj: vlastní zpracování na základě aplikace SoapUI

Zleva vidíme strukturu dat, které byly odeslány na předem puštěnou virtuální službu na lokálním hostu. Dále zprava vidíme, že tato struktura byla ověřena. Jméno a heslo, které jsou definovány ve vlastnostech, jsou přijaté službou, která vrátila číslo přiděleného spojení (session). Je potřeba říci, že parametry není nutné definovat takhle přímo do příkazů, je možné použít environmentální proměnné, které jsou přístupné pro celý projekt.

SoapUI umožňuje sloučení testů do balíčků za účelem poskytnout lepší přehled o testování. Výhodou tohoto produktu je možnost jeho spuštění pomocí skriptů v příkazovém řádku bez grafického uživatelského rozhraní, což pomáhá ušetřit čas.

Postman

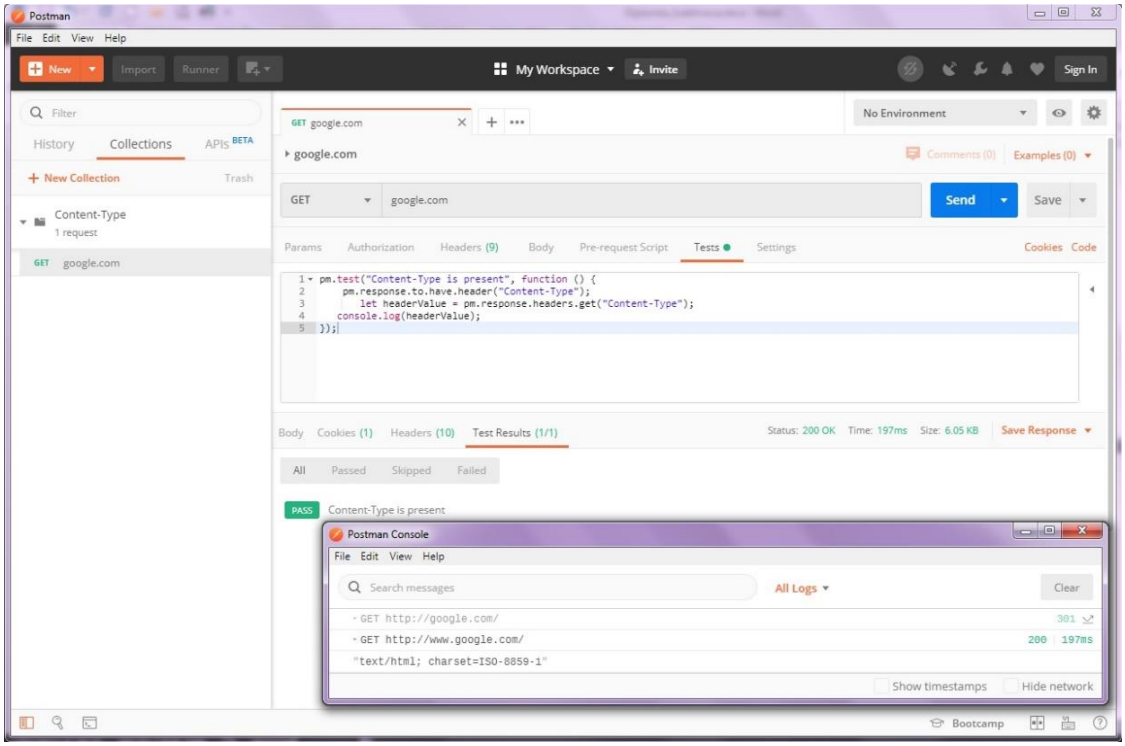
Původně Postman byl rozšířením v prohlížeči Google Chrome, ale později s růstem popularity a nabízených možností byl představen jako samostatný produkt na otestování REST API. Základní verze programu je zdarma, ale rozšířená funkcionalita vyžaduje zakoupení předplatného, takže mluvíme o softwaru typu shareware [26].

Podíváme se na příklad jednoduchého testu v Posmanu, který má za úkol zjistit, zda odpověď z webové stránky obsahuje hlavičku Content-Type, a zároveň vypsat do konzole Postmanu její obsah. Do pole „Test“ se přidá tento skript:

```
pm.test("Content-Type is present",
function () {
    pm.response.to.have.header("Content-
Type");
       let headerValue =
pm.response.headers.get("Content-Type");
   console.log(headerValue);
});
```

.

Pomocí metody GET Postman zavolá na adresu *google.com* a vrátí zprávu o tom, jak dopadl test a obsah hlavičky. Obrázek č. 4 znázorňuje tento výsledek.



Obrázek 4 Výsledek testu v Postman
Zdroj: vlastní zpracování na základě aplikace Postman

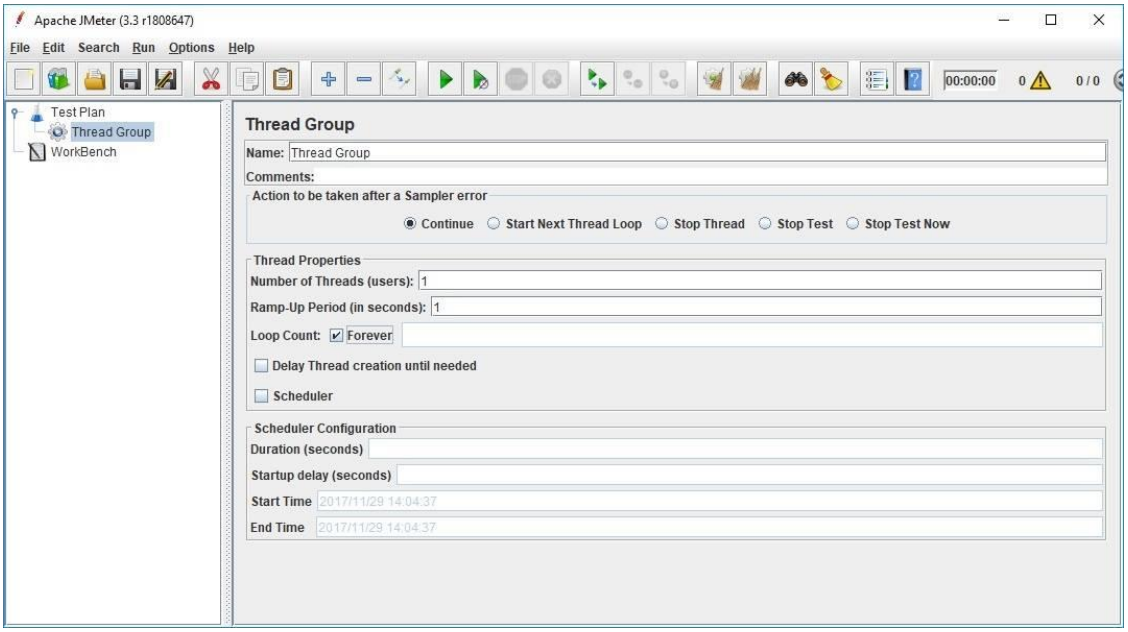
Jak můžeme vidět, hlavička Content-Type je ve zprávě a její obsahem je „text/html; charset=ISO-8859-1“. Zároveň vidíme, že Postman umí vrátit tělo odpovědi, cookies, seznam všech hlaviček, stav požadavku, dobu na jeho provedení. Stejně jako SoapUI, Postman umožňuje seskupit testy do balíčků, které mohou být spuštěny jako jeden úkol.

4.2.2 Zátěžové testování

Apache JMeter je známým a mocným programem používaným pro zátěžové testování. Původně měl pomáhat při webovém testování, ale později byl rozšířen a v současné době umožňuje testovat FTP, JMS, LDAP a jiné typy spojení. Funguje na tom principu, že posílá předem daný počet nadefinovaných příkazů buď najednou, nebo po určitých časových úsecích a tím předstírá odesílání reálných uživatelských zpráv. Nehledě na to, že základní verze nabízí spoustu možností včetně logování a výpočtu základních statistik, jeho funkcionalita může být rozšířena pomocí pluginů, které umožňují například nadefinovat různou zátěž v závislosti na času (simulace denního růstu a nočního spadu uživatelské aktivity), znázornit výsledky testování

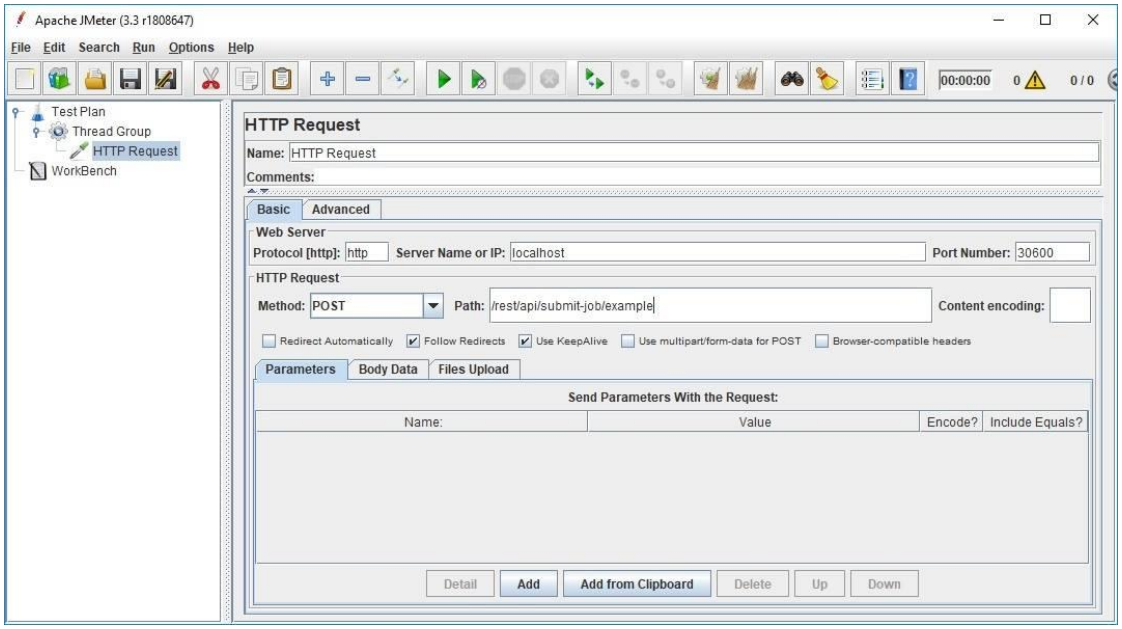
pomocí grafů a mnoho jiného. JMeter je napsán v jazyce Java a potřebuje její instalaci pro svůj běh [27].

Příklad konfigurace jednoduchého zátěžového testu pomocí JMeter je znázorněn na dalších obrázcích č. 5 a 6.



Obrázek 5 Nastavení počtu testovacích vláken v JMeter
Zdroj: vlastní zpracování na základě aplikace JMeter

JMeter umožňuje sestavovat celé testovací plány s velkým množstvím příkazů. První vlastnost, kterou je potřeba nadefinovat, je počet vláken, které imitují uživatele. Na obrázku č. 5 vidíme, že klient bude posílat jednu zprávu během jedné vteřiny; v případě chyby testování bude pokračovat a celý test bude trvat do té doby, dokud ho nezastaví tester.



Obrázek 6 Nastavení příkazu v JMeter
Zdroj: vlastní zpracování na základě aplikace JMeter

Obrázek č. 6 ilustruje, co budou dělat virtuální uživatelé nadefinovaní v předchozím kroku. V našem příkladu se budou posílat jednoduché zprávy na aplikaci, která běží s nezabezpečeným spojením HTTP na lokálním hostu na portu 30600. Příkaz bude realizovat metodu POST (vytvořit) na cestě zadané v políčku „Path“. Do testovacího plánu je možné přidat jednoduchý posluchač, který dokáže vypsat výsledky a přijaté zprávy na každý zaslaný příkaz.

4.2.3 Automatické testování uvnitř prohlížeče

Testování webových aplikací nekončí jenom posíláním příkazů a vyhodnocením přijatých zpráv. Je také potřeba provést velké množství manuálních kroků uvnitř programu, vyzkoušet a použít ho jako běžný uživatel. Na pomoc přichází nástroje, které umožňují spustit webové prohlížeče a imitovat uvnitř nich všechny klasické akce: vyplnit pole, smazat data, zmačknout tlačítko, počkat na odpověď serveru apod. Jedním z těchto úžasných pomocníků je Selenium s otevřeným kódem a podporou všech základních prohlížečů a operačních systémů. Po spuštění driveru žádaného prohlížeče, umí Selenium komunikovat s aplikací nebo stránkou přímo pomocí API nebo testovacích skriptů, které popisují akce s objekty, označené identifikátory a očekávané výsledky. Použití tohoto programu potřebuje značné

znalosti v programování a časové investice do napsání testů, ale na druhou stranu dokáže citelně usnadnit a zkrátit proces testování [28].

5 Problematika agilního vývoje

Agilní metodika vývoje je příkladem pružného přístupu k vytváření softwaru, který se liší od vodopádové technologie převážně tím, že umožňuje vnášet změny do programu bez nutnosti přepisování a schvalování na všech úrovních původního zadání. Zákazník, který se obrací na vývojářskou firmu s ne úplně definovaným úkolem, může během již zahájené práce nad produktem do jisté míry doplňovat či úplně měnit požadavky. Kromě toho je vývojářský tým schopný navrhnout lepší nebo vhodnější řešení vzniklé v průběhu vývoje. Od fungujícího agilního přístupu se očekává, že kvalita produktu bude více korespondovat s požadavky reálného světa, zrychlí dodání softwaru, zlepší přehlednost o jeho stavu a bude podporovat vývojářský tým se neustále rozvíjet. Jak vidíme, tento přístup přináší větší míru přizpůsobivosti a dokonce i kreativní svobody do procesu tvorby programu. Ačkoliv výhody jsou viditelné, zavést ho do produkce není tak jednoduchý úkol, jelikož potřebuje změnit všechny zásady firemních procesů, a dokonce i zapracovat na mezilidských vztazích a na zmírnění úrovně stresu, který přichází s potřebou pracovat v pevných časových rámcích. Je potřeba pamatovat na to, že agilní přístup se prokáže nejvíce při realizaci rozsáhlých a složitých projektů, nad kterými se plánuje pracovat v dlouhém časovém období s alokací relativně velkého počtu zaměstnanců [29].

Agilní vývoj byl poprvé definován v roce 2001, když byl skupinou softwarových inženýrů sepsán *Manifest agilního programování* za účelem usnadnění vývojového procesu [30]. Jedním z autorů, kteří se podíleli na zpracování tohoto manifestu, byl Kent Beck, již několikrát zmíněný dříve. Nehledě na to, že agilní přístup vyžaduje splnění jenom dvanácti základních principů, ve skutečnosti to znamená potřebu zavedení velmi rozsáhlých metodik alespoň částečně.

K zásadám agilního vývoje patří [30]:

1. Cílem vývojářů je kontinuální doručování kvalitního produktu.
2. Všelijaké změny úkolů jsou vítány ve všech fázích vývoje; konkurenceschopnost klienta je prioritou.
3. Funkční software se dodává v určitých časových úsecích, přičemž kratší mají nejvyšší prioritu.

4. Vedení, klient a bezprostřední účastníci vývoje jsou ve stálé komunikaci.
5. Motivace pracovníků je základním bodem projektu; vedení důvěřuje svým zaměstnancům, že ti zodpovědně přistupují ke své práci, a vytváří pro ně komfortní prostředí.
6. Osobní konverzace je nejúčinnějším typem komunikace.
7. Fungující a kvalitní software je hlavním znakem úspěchu.
8. Je třeba podporovat trvalý rozvoj a jeho stabilní tempo.
9. Je třeba se neustále snažit o vylepšení produktu.
10. Analýza úkolů a řízení pracovního procesu probíhá na základě priorit.
11. Týmy samy navrhují nejlepší řešení.
12. Týmy neustále analyzují odvedenou práci a zamýšlejí se nad tím, jak se stát efektivnější.

Jak můžeme vidět, jedná se o částečně idealistický pohled na spolupráci jak mezi klientem a developerským týmem, tak i na nižší úrovni mezi kolegy. Nicméně na bázi těchto představ vznikly metodiky, které pomáhají vytvořit určité podmínky kooperace. Řeč o některých z nich je uvedena níže.

5.1 Agilní metodiky

Firmu, ve které jsou všechny procesy dokonalé z hlediska agilního vývoje, není jednoduché najít. Hlavním problémem je, že i po zavedení agilních metodik, které mají za úkol pomáhat, ale nikoliv obtěžovat, všechny činnosti mohou v podstatě probíhat starým způsobem, jenom mají jiný moderní název. Musí být realizována jednoduchá myšlenka: vylepšení nemá být jedině pro vylepšení a fajfku na papíře, ale má přinášet reálnou hodnotu [31, s. 2].

5.1.1 Scrum

Vývojový proces se skládá z přesně daných časových období, během kterých produkt musí dostat nové vlastnosti a funkcionalitu, být otestován, stát se funkčním řešením schopným předstoupit před zákazníka. Tyto úseky se nazývají *sprinty* a trvají obvykle od jednoho do čtyř týdnů, přičemž čím kratší je sprint, tím větší je

průhlednost vývoje a jeho předvídatelnost, ale také je větší čas věnován plánování. Vlastník produktu – člověk, který má vizi o tom, jak má program vypadat – po diskuzi s klienty nebo jinými zájemci, vytváří souhrn požadavků na koncový software a přiděluje jim prioritu. Tento strukturovaný a prioritní seznam úkolů je zdrojem práce pro vývojáře a má název *backlog*. Před každým novým sprintem vývojový tým rozhoduje, čím se bude zabývat, a bere do závazku takzvané *story*, které nejsou nic jiného než úkoly z *backlogu*. Je potřeba sjednotit práce, a proto každý den probíhají synchronizační schůzky (angl. *scrum* nebo *stand-up*), kde členové týmu vypráví o odvedené práci, o tom, co plánují na další den, o případných potížích a zajímavých připomínkách. Podobná setkání jsou řízena scrum masterem. Tuto funkci může plnit vedoucí týmu, který pomáhá sledovat, zda se stíhají časové termíny a nepotřebuje-li někdo eventuální pomoc.

Plánování sprintů, jako i plánování koncového produktu, se musí umět přizpůsobit. Za účelem jeho zlepšení je vhodné provádět schůzky zpětného vyhodnocení odvedené práce, kde se sdílejí názory, kladné zkušenosti a případné problémy. Setkání mohou probíhat na dvou úrovních: přehled sprintu (*sprint review*), kde se rozebírá, jaké úkoly byly splněny, a retrospektiva, kde vývojáři diskutují o tom, jak dosáhli svého výkonu [31, s. 42-43].

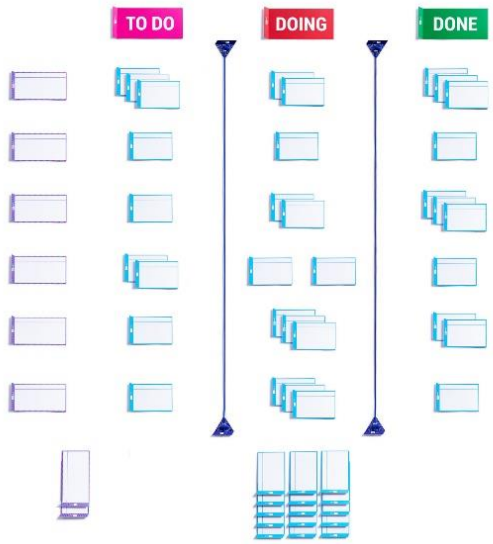
5.1.2 Systém Kanban

Hlavní myšlenkou systému Kanban je, že práce probíhá jenom nad tím úkolem, který je momentálně v prioritě. To znamená, že pokud do organizovaného provozu náhle vstoupí jiný požadavek s vyšší prioritou, ten poslední vyhrává a předchozí práce je odložena. Kanban potřebuje stanovení pracovního postupu (*workflow*) kterým prochází každý úkol, a poté zavést systém sledování tohoto toku [31, s. 324]. Vizualizace, přístupná všem účastníkům, je součástí monitorování [31, s. 326] a může být jak ve fyzické, tak i virtuální podobě pomocí takových nástrojů, jako například Jira nebo Trello [32]. V Kanbanu neexistují přesně daná pravidla, ale spíše doporučení, jak je možné sdílet informace o probíhajících procesech. Níže jsou některé z nich:

- je definován zdroj úkolů – backlog,

- pracovní postup se skládá z fází, definujících stav práce nad úkolem, například: ke zpracování, ve fázi vývoje, splněné,
- každý účastník si bere úkol, označuje ho vhodným způsobem a postupně přenáší z jedné fáze do jiné v souladu s jeho progresem,
- postup je sdílen, například s pomocí tabule [31, s. 323-324].

V praxi to může vypadat takto:



Obrázek 7 Ukázka Kanban tabule
Zdroj: přístup z Internetu: https://images-na.ssl-images-amazon.com/images/I/817LN1GY9YL._AC_SL1500_.jpg

Každý tým má právo přizpůsobit postupy v souladu se svými potřebami.

Nehledě na to, že Scrum a Kanban vycházejí z různých zásad a mají totálně odlišný přístup k časovým limitům, je možné je sjednotit do jednoho fungujícího systému, kde celý projekt je řízen pomocí Scrum, ale progres jednotlivých sprintů a neprioritních úkolů je znázorněn pomocí Kanban tabule.

5.1.3 Extrémní programování

Autorem základní myšlenky extrémního programování je Kent Beck, nicméně později jeho návrh získal rozvoj a podporu od jiných vývojářů a v současné době má velké množství postupů. Cílem této metodiky je vylepšit kvalitu kódu a přitom zkrátit dobu vydání nové verze. Důraz je kladen na realizaci nejjednoduššího a

nejlevnějšího řešení, odpovídajícího zadání, přitom bez doplnění funkcionality, která je potenciálně užitečná v budoucnosti, ale není nutná v současnosti. Vzhledem k tomu, že práce nad produktem se skládá z nepřetržitých iterací vývojových fází, vždy je zde možnost rozšíření programu, pokud to je nezbytné. Riziko setkání se s nežádoucími účinky spojené s tímto přístupem a použitím příliš jednoduchého řešení se eliminuje pomocí neustálého refaktorování a vylepšení kódu. Komunikace mezi kolegy tady přechází na novou úroveň. Zavádí se zde praxe párového programování, které na rozdíl od klasického systému s ověřením kódu jednoho programátora jiným, již ve fázi jeho sestavení, vyžaduje spolupráci dvou vývojářů: zatímco jeden programuje, druhý vedle sleduje a kontroluje jeho práci. Zpomalení vývoje se kompenzuje odhalením chyb již na začátku. Kromě toho se za účelem předcházení nepřesnostem a bugům praktikuje programování řízené testy, když do produkce jde jen ten kód, který prošel všemi unitovými testy. Sjednocení kolektivní práce probíhá s pomocí zavádění průběžné integrace podle zásad, vzniklých ve společnosti IBM ještě v šedesátých letech. V roce 1991 Grady Booch zobecnil a popsal tyto principy a také definoval formální standardizace zdrojového kódu a jeho stylu zápisu [31, s. 175 - 179].

5.2 Testování v agilním vývoji

Stálé vylepšování a zdokonalování produktu již zdánlivě dokončené práce přináší určitou míru stresu pro QA. Testeři nedostávají konečný program, kdy musí ověřit úplně všechno najednou, ale získávají verze s relativně malými funkčními změnami. Striktní časová omezení sprintů nedovolují provádět komplexní regresní testování každých pár týdnů, ale vyžadují detailní analýzu změn a nové funkcionality. Takový přístup vyžaduje vytvoření procesů pro získání rychlé zpětné vazby pro každou verzi kódu, aby vývoj probíhal co nejrychleji. V průběhu krátkého časového úseku musí vývojářský tým dostat informace o tom, jestli nové změny nenesou chyby a z tohoto vyplývá, že podstatou agilního testování je rychlé poskytování zprávy o relevantním stavu aplikace. V případě nálezu chyby je nutné ji nahlásit co nejdříve, aby se problém dostal do stavu vyřešených a vývoj mohl pokračovat dále. Přístup, při kterém se nalezené chyby ignorují, přičemž se práce na změnách nezastavuje, přináší riziko dostat se do situace, když původní příčina závady je skrytá pod

širokou sítí souvislostí a vyžaduje značně více času na její odhalení a odstranění. Nepříznivým důsledkem podobného chování může být nutnost přepsání perfektně fungujících funkcí, které ale byly postaveny na chybě a ztratily svoji hodnotu [20, s. 50, 56].

Testování v agilním vývoji nespadá do procesů, kterými se zabývají jedině testeři, jelikož vývojová a testovací fází nejsou odděleny od sebe a představují nepřetržitou činnost, která je součástí vývoje. Kvalitní program je výsledkem námahy celého týmu, a proto se vývojáři podílejí na zajištění kvality, a to převážně buď prostřednictvím programování řízeného testy nebo unitovového testování [20, s. 51-52]. Tento přístup dává testerům více času a prostředků pro explorativní procházení programů. Nicméně při kompletní analýze nových vlastností není možné říct, že jsou tyto části zcela oddělené od stabilních částí programu a nemají na ně žádný vliv, a proto je regresní testování nezbytností, které potřebuje hodně času, který často není. Za účelem řešení tohoto začarovaného kruhu přichází spoluúčast testerů během diskutování o nových požadavcích, automatizace testů a zavádění průběžné integrace a průběžného nasazení, které umožňují dokonce několikrát za den dostávat výsledky testování i těch nejmenších změn.

6 Průběžná integrace a průběžná dodávka

Tato kapitola je založena na článku Martina Fowlera *Continuous Integration* [33] a rozšířena pomocí knihy *Continuous Delivery*, jejíž autoři jsou: Jez Humble a David Farley [34].

Jednou z hlavních metodik v softwarovém inženýrství, která pomáhá sledovat kvalitu kódu již během jeho vývoje a přispívá ke zkrácení času na doručení změn, je zavedení průběžné integrace a průběžné dodávky softwaru, neboli CI/CD (*Continuous Integration & Continuous Delivery)*. Princip fungování CI/CD připomíná práci dopravníku: metodika plní integrační funkci, když od prvních změn v kódu začínají probíhat různé typy automatických testů, a to v každé fázi vývoje, s následnou dodávkou a instalací finální verze jako hotového produktu pro koncového uživatele.

CI/CD platformy, na jejichž základě je princip realizován, podporují provádění pravidelného automatizovaného sestavení projektu, aby rychle identifikovaly vady a řešily integrační problémy. Při standardním (vodopádovém) přístupu, kde vývojáři nezávisle pracují na různých částech systému, je integrační fáze konečná a při identifikaci chyb může nepředvídatelně zpomalit konec vývoje. Přechod na průběžnou integraci umožňuje snížit náročnost práce a učinit ji více průhlednější pomocí včasným a nepřetržitým odhalením a eliminacím chyb a rozporů. Souhrn vlastností dělá CI/CD mocným nástrojem, podporujícím agilní vývoj.

Metodika CI/CD se skládá ze třech částí, i když v názvu vidíme jenom dvě: kromě průběžné integrace a dodávky můžeme mluvit také i o průběžném nasazení, které je poslední etapou vývojového procesu. Dále všechny tyto tři fáze budou rozebrány detailněji.

6.1 Průběžná integrace

Princip průběžné integrace byl poprvé zmíněn autorem jazyka UML Grady Boochem v jeho díle *Object Oriented Analysis and Design With Applications* v roce 1991. Nicméně později byla jeho myšlenka přijata a rozšířena Kentem Beckem a tak průběžná integrace zapadla do principů extrémního programování s tím rozdílem, že sloučení kódu se předpokládá ne jednou za den, ale mnohem častěji v menších

dávkách. Jak už bylo naznačeno výše, podstatou CI je zajistit sloučení a kontrolu rozdrcených funkčních změn v kódu od různých vývojářů. Hned po provedení změn, ale před tím, jak se dostanou ke kolegům, je vývojář schopný relativně rychle otestovat svoji práci a dostat zpětnou vazbu o kvalitě svého řešení.

Aby bylo jasné fungování procesů průběžné integrace, je nutné vysvětlit několik specifických detailů.

1. Celý zdrojový kód musí být uložen do jednoho repozitáře, který sdílí každý účastník projektu. Důvod je pochopitelný: zabezpečit přístup ke všem zdrojům a zajistit jejich konzistenci. Dobrou praxí je také použití verzovacích systémů, které nabízí přehled o historickém vývoji softwaru s možností vrátit se do poslední bezchybné verze a provádět zálohování. Verzovací systémy bývají dvou typů: centralizované a distribuované. První jsou postaveny na schématu klient-server a mezi ně patří např. CVS, neboli *Concurrent Versions System*. Princip jeho fungování je jednoduchý. Na začátku své práce se vývojář (klient) musí připojit na server, který obsahuje centrální repozitář s kódem. Po stažení nutné verze se vytváří lokální kopie projektu, se kterou dále vývojář pracuje. Pro zpřístupnění provedených změn je nutné je nahrát zpět do hlavního repozitáře. Synchronizace kódu vždy probíhá pomocí serveru a centrálního úložiště, které je jediným zdrojem celého projektu. Tento systém se již považuje za zastaralý a postupně se nahrazuje distribuovanými systémy kontroly verzí, které přinášejí více svobody vývojářům. Rozdíl spočívá v tom, že centrální repozitář je nutný jedině v případě, když je potřeba nahrát změny, ale ve skutečnosti každý lokální stroj má svoje úložiště s kopií celého projektu. Uživatelé pravidelně synchronizují místní repozitáře s hlavním, ale pracují se svojí lokální kopií. Kromě větší flexibility s experimentováním během vývoje tento systém zvyšuje míru spolehlivosti, jelikož v případě ztráty centrálního úložiště je hlavní projekt na každém vývojářském stroji. Příkladem distribuovaného systému může být Git.

   I verzovací systémy mají své nevýhody. Jedním z hlavních problémů je narůstající objem dat zpomalující rozjíždění projektu, jelikož průběžná

integrace vyžaduje nahrávání inkrementálních změn do repozitáře s vysokou frekvencí. Nehledě na to, že verzovací systémy nikdy neukládají identické dokumenty, ale jen změny, v rozsáhlých projektech je stále riziko, že nedohledáme potřebnou verzi.

2. Musí být realizován algoritmus vytváření *buildu*, kde *build* je plně funkční program sestavený ze zdrojového kódu. Tento algoritmus předpokládá zahrnutí velkého množství činností, mezi které bez pochyb patří: práce se soubory a daty, kompilace kódu, pouštění a kontrola testů, vytváření instalačního balíčku a jeho umístění nejlépe ve sdíleném úložišti. Celý tento proces obnáší veškeré aktivity, které jsou nutné pro vytvoření životaschopného softwaru. Je nutné zmínit, že build je obecným pojmem, který může označovat verzi vydávanou na konci každého vývojového dne (*nightbuild*), ale také verzi na konci sprintu.

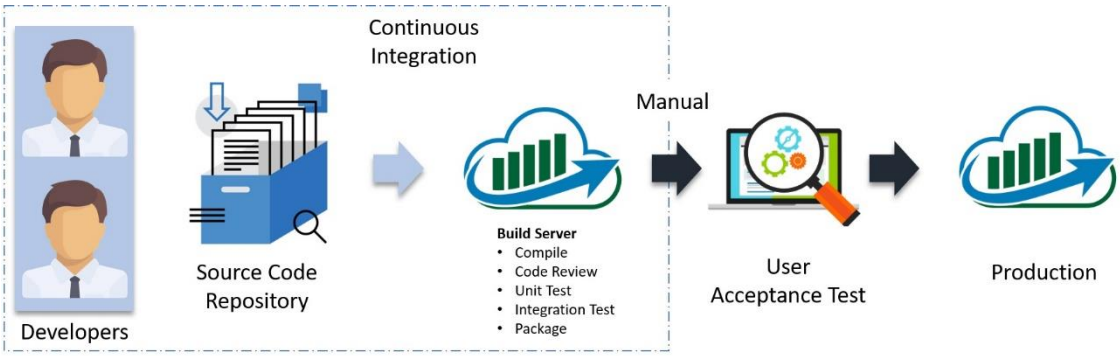
3. Doručení buildu mělo by být automatickým procesem, který se spouští pomocí jednoho příkazu buď manuálně, nebo dokonce pravidelně bez jakéhokoliv zásahu člověka. Tato automatizace pomáhá významně ušetřit čas vývojáře, jelikož ho zbavuje nutnosti plnit rutinní úkoly, které dokáže provádět stroj. Existuje celá řada softwarových prostředků, které na základě skriptů a nastavení na požádání nebo podle rozvrhu provedou všechny výše zmíněné výše činnosti a výsledek (instalační balíček) uloží na předem určené místo. K nejznámějším patří Jenkins - open-source produkt, osvědčený časem a nabízející velice širokou sadu pluginů.

4. Použití integračního serveru (*build* serveru) dovoluje přerozdělit zátěž a neomezovat práce jednoho konkrétního vývojáře. Samostatný stroj je řízen centralizovaně a jeho cílem je příprava buildu. Jeho použití přináší hned několik benefitů. První, již zmíněný, je v tom, že lokální počítač, na kterém pracuje vývojář, nespotřebuje svůj výkon. Druhý přínos není tak zřejmý. Integrační server by měl mít čisté prostředí, které neovlivní probíhající procesy, zatím co pracovní počítač vývojáře často není v perfektním stavu z tohoto hlediska. V tomto případě příprava buildu na vyhrazeném stroji může zabránit situaci, kterou můžeme charakterizovat jako syndrom „na

mém počítači to funguje“, jelikož program, fungující na dvou strojích se nejspíš spustí i na třetím.

5. Jelikož mluvíme o automatizaci procesů, je nutné zajistit vytváření a ukládání informativních logů, které jsou mocným prostředkem pro zjištění příčin případného problému. Kromě odhalení chyb logy dokážou přispět v provedení analýzy, která je schopná najít úzká hrdla nebo takové charakteristiky programu nebo procesu, které je možné optimalizovat a provádět, například rychleji nebo s menší strojovou zátěží. Kromě logů je dobrou praxí nastavit i zjednodušenou verzi zpětné vazby jako například indikátory ve tvaru jednoduchého semaforu, kde červená barva znamená, že se vytvoření buildu nepodařilo, zatímco zelená signalizuje jeho úspěšné vytvoření.

Schéma na obrázku č. 8 znázorňuje princip průběžné integrace.



Obrázek 8 Průběžná integrace
Zdroj: přístup z Internetu: https://digitize01.com/devops-tools-jenkins-ci-cd/. Upraveno

1. V prvním kroku integrace vývojáři posílají výsledek své práce do repozitáře zdrojového kódu. Tento proces je do určité míry automatický: to znamená, že spouštěč nahrávání je manuální na požadavek programátora, ale dále porovnání kódu, jeho nahrávání do úložiště a verzifikace probíhá bez účasti člověka, dokud nedochází k problémům. Možné potíže jsou nejčastěji spojeny s tím, že vývojář neměl aktuální zdroje. V tomto případě jeho řešení může například odkazovat na metodu, která byla odstraněna jeho kolegou. Druhý problém může nastat, pokud byly změny provedeny na stejném místě několika vývojáři najednou. Podobné konflikty se řeší manuálně.

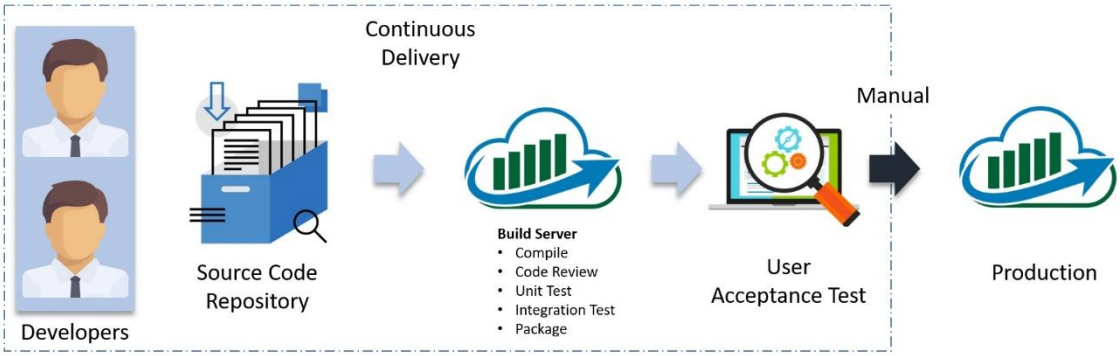
2. Po tom, jak byly změny v kódu nahrány do jednoho repozitáře, může začít proces sestavení buildu, který byl již zmíněn výše. Integrační server dovoluje spustit všechny potřebné činnosti na samostatném stroji. Jestliže základní sestavení aplikace prošlo úspěšně, začíná fáze automatického unitového a někdy také integračního testování. Další práce testerů a vývojářů je ovlivněna výsledky procházení testů, které mohou okamžitě naznačit problém. Negativní scénář může nastat na jakékoliv etapě a vyžaduje manuální zásah do procesů. V opačném případě jsou všechny fáze automatické a probíhají na pozadí.

Průběžná integrace pomocí vysoké míry automatizace pomáhá sjednotit vývojáře a jejich práce. Nicméně samostatně stojí požadavek na zavedení automatického testování. Často se stává, že projekt má bohaté pokrytí nejjednoduššími unitovými testy, které píšou sami vývojáři, ale kontrola propojení samostatných aplikačních prvků a jejich fungování probíhá manuálně. Problém přichází s růstem funkcionality, když manuální otestování softwaru postupně začíná vyžadovat více času, než má QA tým v úsecích mezi vydáním verzí. Jestli z nějakého důvodu není možné zavést automatizaci, začíná software postrádat kvalitu. Pak může být management nucen posouvat datum dodání produktu.

Další otázka zní: Je potřeba pokaždé spouštět všechny testy, pokrývající celou sadu testovacích případů? Odpověď převážně spočívá v rozměru projektu a složitosti a náročnosti testů včetně časových nákladů. Tento problém se řeší individuálně, ale obecně platí, že menší projekty nebo projekty s rychlým během testů nevyžadují jejich revizi. Větší projekty za účelem úspory času mohou použít následující strategii: všechny testy jsou zanalyzovány a zařazeny do skupin podle určitých kritérií. Základní smoke testy v tomto případě mohou tvořit první kategorii. Následné kombinace jsou určeny charakteristikami softwaru, ale princip rozdělení je následující: jestliže změny proběhly v modulu A, je nutné spustit všechny testy, které odkazují na tento modul A. Nicméně není možné testovat všechny prvky programu samostatně, jelikož modul A může mít skrytý vliv na moduly B i C podle teorie systémů, a proto tento přístup parciálního testování je vhodný jenom pro primární kontrolu a systémové testy se vyloučit nedají.

6.2 Průběžná dodávka

Průběžná dodávka je procesem, který navazuje na průběžnou integraci a je její rozšířením, které se zaměřuje na nasazení softwaru do prostředí, maximálně podobající se produkčním podmínkám. Průběžná dodávka neexistuje samostatně bez integrace, takže CI je povinnou prerekvizitou pro pokračování. Význam nasazení aplikace do testovacího prostředí spočívá v možnosti provedení kontroly kvality produktu na vyšší úrovni: aplikace je možné nainstalovat, případně propojit s dalšími systémy, provést všechny potřebné nastavení, spustit a celá vyhlášená funkcionalita bude fungovat podle očekávání. Kromě provedení systémových testů, pracujících se složitými testovacími scénáři, tato fáze má za úkol zkontrolovat i extrémní případy použití, jako například zátěžové testy, testování clusterů, provést kontrolu rezistence vůči pádům infrastruktury apod. Specifikem CD je automatizace všech těchto procesů. Obrázek č. 9 graficky znázorňuje, co je průběžná dodávka.



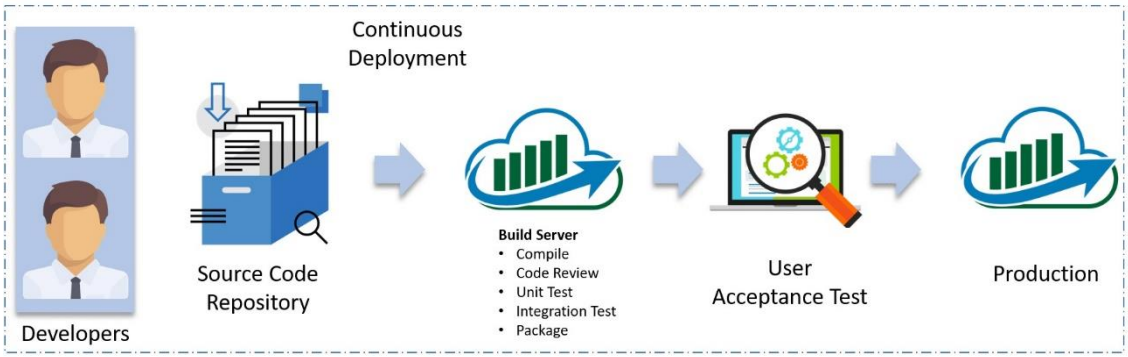
Obrázek 9 Průběžná dodávka
Zdroj: přístup z Internetu: https://digitize01.com/devops-tools-jenkins-ci-cd/. Upraveno

Ze schématu je zřejmé, že průběžná dodávka má automatické tyto procesy: sjednocení změn v kódu, sestavení buildu (což tvoří průběžnou integraci) a také nasazení produktu včetně výše zmíněného otestování. Průběžná dodávka potřebuje více technických a organizačních řešení pro zavedení: může vyžadovat poskytování nového stroje a jeho začlenění do infrastruktury a také údržbu, nucený přechod z manuálních postupů testování na automatické, eliminaci problémů spojených s technickými charakteristikami projektů apod.

6.3 Průběžné nasazení

Průběžné nasazení navazuje na průběžnou integraci a dodávku, které jsou jeho prerekvizitou. Podstatou průběžného nasazení je automatické dodání finální hotové verze softwaru a jeho příprava na produkčním prostředí zákazníka, přičemž zásah vývojáře není vyžadován. Tento přístup se příznivě liší od klasického doručení obnovení díky své vysoké rychlosti nasazení a možnosti operativního poskytování změn. Drobné a časté aktualizace pomáhají dostávat rychlejší odezvu a snižují rizika, čas a náklady na odstranění případných problémů.

Ze schématu na obrázku č. 10 lze soudit, že průběžné nasazení obsahuje všechny předchozí průběžné etapy od provádění změn v kódu do instalace a přípravy plně funkční oficiální verze softwaru v produkci. Příkladem realizace může být řešení společnosti Microsoft, která poskytuje podporu a vydává aktualizace pro Windows operační systémy a také zavedla online doručení nových verzí. Všechny tyto procesy probíhají automaticky na lokálním počítači uživatele.



Obrázek 10 Průběžné nasázení
Zdroj: přístup z Internetu: https://digitize01.com/devops-tools-jenkins-ci-cd/. Upraveno

CI/CD je obecnou koncepcí, jak lze nastavit procesy, aby vývoj probíhal co nejrychleji a zároveň přinášel kvalitní produkt, jelikož se chyby často odhalují na každém kroku. Základem je myšlenka o provedení iterativních synchronizačních úloh a neustálé testování aplikace. Za účelem šetření času a eliminace lidského faktoru co nejvíce procesů musejí být automatizované. CI/CD metodika není dogmatem, ale flexibilní strategií, která musí být přizpůsobena potřebám projektu. Nakonec i po zavedení procedurálních změn a zvýšení míry automatického

testování, manuální procházení aplikace se nedá úplně odmítnout a zároveň nelze podceňovat význam kontroly a analýzy probíhajících procesů.

Zavedení CI/CD není jednodenní záležitostí, ale vyžaduje provedení úplné a detailní analýzy již uplatněných postupů, technických a technologických prostředků, možných překážek během realizace a následného pečlivého nastavení nových procesů. Kromě toho nelze vyloučit odpor ze strany vývojářů a testerů: povinnost psaní testů může vyděsit vývojáře, zatímco u testerů, převážně manuálních, se může objevit nedostatek znalostí a způsobilostí potřebných k automatizaci. CI/CD klade větší nároky na kvalifikaci zaměstnanců. V případě automatického nasazení produktu existuje riziko selhání použitých skriptů a následky této poruchy mohou být kritické, zatímco zodpovědnost leží na vývojářích, kteří nemají plnou kontrolu nad procesem. Nutnost průběžného nasazení musí podlehnout pečlivé analýze, jelikož se může stát, že ve složitějších případech je riziko ztráty větší než výhody, které přináší rychlé dodání. Management, nejlépe ve spolupráci s vývojářským týmem, musí rozhodnout, jestli projekt potřebuje automatické nasazení nebo je vhodnější zůstat u průběžné integrace a dodávky a doručení nechat manuální.

7 CI/CD platformy

K vytvoření této kapitoly byly použity informace z oficiální dokumentace následujících produktů: Jenkins [35], TeamCity [36] a GitLab [37].

Současný trh informačních technologií nabízí širokou sadu softwaru pro každý případ použití. Podobná rozmanitost může zapříčinit dezorientaci v tom, jaké řešení je vhodné použit. Při výběru hlavního produktu je důležité si stanovit kritéria, která mohou napomoci v rozhodnutí. Níže jsou představeny některé obecné charakteristiky, které stojí za pozornost.

Možnosti integrace a podpory softwaru třetích stran

K aplikacím třetích stran patří systémy pro řízení projektů (například Jira), nástroje pro sledování incidentů a chyb (například YouTrack), platformy vedení testovacích případů (například TestRail), nástroje pro statickou analýzu kódu a jiné. Kromě toho CI/CD platforma musí být dost flexibilní a zajišťovat podporu nástrojů umožňujících automatické vydání verzí (například Maven, Gradle). Neméně důležitá je integrace s různými verzovacími systémy.

Možnosti hostingu

Softwarové nástroje se liší v infrastruktuře řízení. Cloudové nástroje jsou umístěny na straně poskytovatele, vyžadují menší náklady na nastavení a mohou být upraveny na vyžádání v závislosti na potřebách projektu. Existují také řešení pro vlastní provozování. Odpovědnost za jejich nasazení a podpora leží na vnitřních lidských zdrojích. Zatímco lokální služby poskytují větší flexibilitu v nastavení nástroje, cloudové aplikace zmírňují potíže s instalací a údržbou a nabízejí větší škálovatelnost.

Použitelnost

Některé nástroje mohou významně usnadnit proces zavedení CI/CD koncepce do

provozu pomocí jejich jasného a srozumitelného grafického rozhraní a UX. Pečlivě navržené rozhraní může pomoci ušetřit čas, převážně ve fázích prvotního nastavení a seznámení se softwarem.

Dokumentace

Nasazení nové technologie vyžaduje vysoké investice, spojené s vyvinutím funkční infrastruktury odpovídající všem požadavkům. Široká databáze znalostí, návodů a příkladů, poskytnutá jak vývojářským týmem použitého softwaru, tak i vývojářskou komunitou, je schopná usnadnit proces zavádění změn.

Možnosti rozšíření

Základní funkcionalita vybraného produktu není vždy postačující. Tento problém může nastat jak u malých projektů, tak i u rozsáhlejších. Pluginy dávají možnost zavedení lepší integrace s jinými systémy a také obecně zvyšují flexibilitu softwaru. Podpora kontejnerizace. V posledních letech se zvyšuje popularita technologie, která pomáhá spouštět aplikaci v izolovaném prostředí. Podobný uzavřený kontejner má předem nadefinované vlastnosti a také může obsahovat předpřipravený balíček potřebných nástrojů. Plugin, umožňující práci s aplikací pro konfiguraci a správu kontejnerů, jako jsou Docker a Kubernetes, usnadňuje připojení CI/CD platformy k cílovému prostředí aplikace.

Cenová politika

Náklady, spojené s pořízením a užíváním softwaru, mohou hrát rozhodující roli, zvlášť u mladých a menších projektů. Nicméně některé nástroje se řídí buď politikou svobodného a otevřeného softwaru, nebo poskytují celou funkcionalitu s limitovaným počtem uživatelů zdarma. V případě, pokud produkt je plně komerční, často se dá využit možnosti demo-verzí a některé podniky pomáhají startujícím projektům tím, že poskytují zvláštní podmínky a slevy.

Níže jsou představeny některé platformy pro podporu průběžné integrace a dodávky. Všechny nástroje plní stejný účel a mají podobnou funkcionalitu, ale liší se ve způsobech realizace.

7.1 Jenkins

Jenkins je jedním z nejznámějších nástrojů, pomáhajících organizovat a provést automatizace procesů sestavení buildu, pouštění testů nad ním a poskytování informace pro analýzu. Aplikace je napsaná v jazyce Java a potřebuje použití Java Runtime Environment (JRE). Jenkins je kompletně svobodný a otevřený software, podporující jak Windows, tak i Unix operační systémy. Základní funkcionalita je dostačující pro zavedení jednoduchého CI/CD procesu, ale složitější případy je potřeba řešit pomocí pluginů. Nevýhodou tohoto přístupu je, že nadměrné množství použitých rozšíření je schopno výrazně zkomplikovat nastavení a údržbu systému. Jenkins disponuje širokými možnostmi adaptace. Aplikace umí provést integraci s různými verzovacími systémy (například CVS, Git, Mercurial), umožňuje použití kontejnerů a také podporuje několik nástrojů pro automatizace sestavování programu, mezi nimiž jsou, například, Maven a Gradle. Kromě toho, pluginy umožňují propojit Jenkins se systémy řízení projektů pro zvýšení přehlednosti o stavu celkového progresu. Mezi potenciální nevýhody je možné zařadit častý nedostatek dokumentace, popisující použití autorských rozšíření a také grafické rozhrání, které plní svoje účely, ale může se zdát méně povedeným. Nicméně vnímání těchto vlastností je obzvláště subjektivní [35].

7.2 TeamCity

TeamCity je CI/CD platformou, vyvinutou v JetBrains a napsanou v jazyce Java. Prerekvizitou pro instalaci a spuštění je Java Runtime Environment (JRE). TeamCity je komerční řešení, ale má pružnou cenovou politiku. Celá funkcionalita se poskytuje zdarma, ale je limitován počet dostupných unikátních konfigurací pro vytváření buildů a build-agentů, kde build-agent je nástroj, umožňující paralelní provedení úkolů. Rozšíření základní licence jsou dostupná za poplatek. JetBrains nabízí korporátní balíčky, slevy pro mladé projekty, možnost využití šedesátidenní demo-verze a také zdarma licence pro open-source produkty. Mezi výhodami TeamCity je

možné zařadit poměrně snadné nastavení a intuitivní uživatelské rozhrání. Bohatá databáze znalostí je užitečným nástrojem, zvlášť pokud vývojářský tým poprvé zavádí koncepci průběžné integrace do firemních procesů. Kromě toho, JetBrains nabízí profesionální podporu pro svoji klientelu. Základní funkcionalita TeamCity je širší, než jsou klíčové možnosti systému Jenkins, a také se dá rozšířit pomocí pluginů. Jako nástroj, který musí propojovat a organizovat fungování aplikací třetích stran, TeamCity podporuje integraci s různými verzovacími systémy, s vývojovými prostředí, se systémy pro analýzy kódu a jinými [36].

7.3 GitLab

GitLab je komplexním nástrojem, nabízejícím velmi bohatou funkcionalitu, která obsahuje správu repozitářů ve verzovacím systému Git, zajištění CI/CD procesů a také podporu pro řízení projektů. Kromě toho GitLab nabízí možnost vedení vlastní wiki, systém evidence incidentů a chyb, nástroje na analýzu kvality kódu a mnoho jiných funkcí. Rozsáhlý balíček poskytnutých možností dává právo zařadit GitLab mezi DevOps (zkrátka od *Development Operations*) systémy. Tento typ nástrojů sjednocuje agilní přístup a principy průběžné integrace a dodávky a pomáhá je doplnit o vývoj a správu projektu. Takovým způsobem vzniká jediný pracovní postup, který zahrnuje všechny etapy od návrhu programu přes jeho vývoj a nasázení do správy a analýzy zpětné odezvy od uživatelů. GitLab nabízí hlavní funkcionalitu jako svobodný software, ale rozšířené možnosti vyžadují zakoupení licencí, přičemž se cena odvíjí od požadavků uživatele. Kromě toho je k dispozici i třicetidenní demo-verze na otestování produktu. Stejně jako Jenkins nebo TeamCity, GitLab nabízí spuštění několika úkolů průběžné integrace a dodávky souběžně, ale počet paralelních procesů je omezen v souvislosti s vybraným licenčním balíčkem. Široké vlastní možnosti GitLabu je možné rozšířit pomocí integrace s externími produkty [37].

Obecné porovnání zmíněných nástrojů není možné bez vazby na potřeby projektu a také lidi, které je budou používat. Každý produkt je unikátní a nabízí rozsáhlou sadu funkcí, ale každá z nich musí být přizpůsobená reálné situaci a vnitřním předpisům a finančním možnostem firmy.

8 Cloudové technologie

Cloudové technologie se staly nedílnou součástí moderního života. Se službami, využívajícími cloud, se běžný uživatel Internetu potýká všude: jsou to Google služby, sociální sítě, datová úložiště a jiné. Kromě toho, tyto technologie se používají také ve školství, řízení vztahů se zákazníky, správě firemních procesů včetně zajištění vývoje softwaru.

Jedna z definic, co je cloud, byla předložena Národním institutem standardů a technologie NIST (*National Institute of Standards and Technology*):

„*Cloud computing je modelem, umožňujícím všudypřítomný, pohodlný, na vyžádání přístup k síti ke sdílené množině konfigurovatelných výpočetních zdrojů (např. sítě, servery, úložiště, aplikace a služby), který může být rychle poskytován a uvolněn s minimálním úsilím o správu nebo interakci poskytovatele služeb*“ [38].

Cloudové technologie již získaly ocenění u softwarových vývojářů, jelikož jim poskytují možnost se soustředit na vývoji obchodní logiky aplikace a snižují lidské náklady na přípravu a údržbu požadované architektury na novém počítačovém hardwaru. Místo nákupu fyzických strojů dochází k pronájmu virtuálního prostoru a počítačů za účelem spuštění různých aplikací a služeb. V důsledku toho, že spotřebitele cloudových technologií kupují jenom nezbytně nutný výkon, tím se eliminuje ztráta z nevyužitého potenciálu vlastního hardwaru [39].

Mezi první cloud computing je možné zařadit služby od MSP (*Managed service providers*). MSP vychází z požadavků spotřebitele a vystupuje v roli dodavatele síťové infrastruktury, která slouží jako základna pro připojení počítačového a softwarového vybavení pro potřeby internetové podnikání. Síťová infrastruktura se fyzicky nenachází u uživatelů, ale u poskytovatelů, kteří se starají o její správu a údržbu. Udržování vysoké kvality dodaného systému je tak prioritou dodavatele, který zároveň hledá cesty optimalizace nákladů a má za úkol zajistit odborníky, kteří se zabývají výhradně instalací a integrací sítě. Odběratel se nemusí zabývat těmito činnostmi a zároveň dostává kvalitní produkt [40]. Na světovém trhu síťového outsourcingu nejvýznamnější roli hrají takové společnosti, jako IBM, Accenture, Cognizant, Atos nebo Capgemini [41].

Před přechodem na cloudové technologie je nutné pečlivě prozkoumat všechny aspekty s nimi spojené, zvážit výhody a případné nevýhody, které přináší této technologie. Některé jsou uvedeny níže:

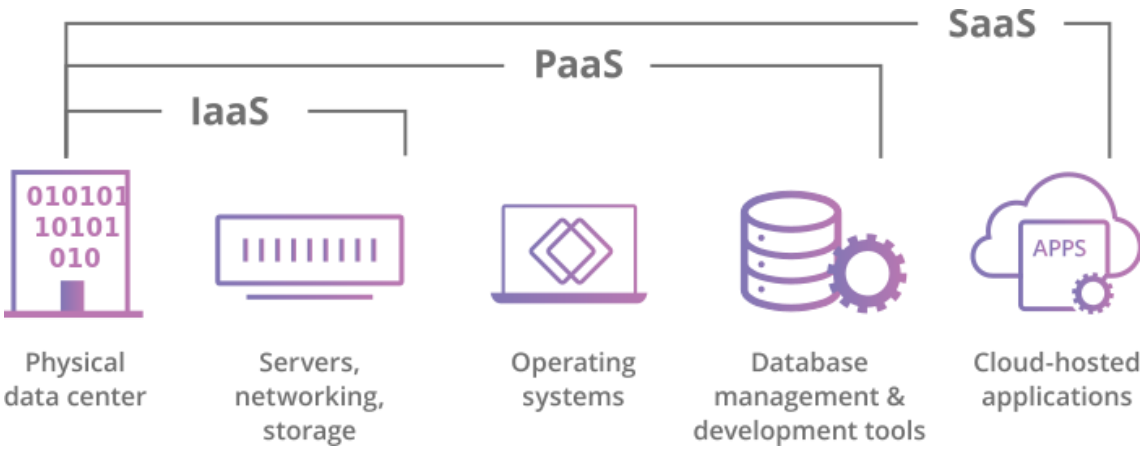
- data, software, počítačový výkon a celá infrastruktura jsou k dispozici z každého počítače, který má připojení k Internetu, což přináší vyšší míru flexibility,
- nižší náklady na provoz a podporu jak softwaru, tak i hardwaru; platba probíhá na základě spotřeby cloudových zdrojů (paměť, výkon),
- poměrně jednoduchá škálovatelnost a rychlé rozšíření již použitých zdrojů,
- pro uživatele je vždy k dispozici nejnovější verze softwaru s uspokojivou úrovní zabezpečení,
- přenos odpovědnosti za bezpečnost dat na poskytovatele služby,
- neexistují žádné náklady spojené přímo s nákupem hardwaru a nákupem licencí lokálně použitého programového vybavení [42].

Mezi základní nevýhody použití cloudových technologií patří:

- problémy přístupu ke cloudovým službám v důsledku slabého připojení k Internetu nebo jeho kompletního výpadku,
- rychlost přístupu k datům závisí na vlastní rychlosti Internetu,
- nedostatečné údaje a dohled nad personálem poskytovatele služby, který má přístup k pronajatým strojům,
- bezpečnostní problém, spojený s možností neoprávněného přístupu k datům a jejich zneužití,
- neschopnost ovlivnit proces zotavení v případě odstávky v provozu služby kvůli problémům, vzniklým na straně dodavatele,
- riziko ztráty dat během procesů migrace,
- státní legislativa a vnitřní bezpečnostní předpisy považující cloudové technologie za rizikové, jelikož firma ztrácí kontrolu nad pronajatým úložištěm dat,

- nedokonalé standardy a postupy, zajišťující propojení aplikací a služeb různých dodavatelů pomocí cloudových technologií [43].

Ke dnešnímu dni existují tři distribuční modely cloud computingu: IaaS, PaaS a SaaS. Na schématu na obrázku č. 11 je znázorněn princip, jak jsou tyto modely propojeny navzájem. IaaS balíček poskytuje čistý výpočetní výkon a úložiště dat, které mohou být přizpůsobeny potřebám uživatele podobným způsobem, jako kdyby místo cloudobých technologií byl použit fyzický server. PaaS model nabízí menší flexibilitu, ale zároveň jednodušší nastavení, jelikož místo základní infrastruktury zákazník dostává virtuální stroje s předpřipraveným hlavním softwarem, jako jsou, například, operační systémy a databáze. SaaS je nejjednodušším modelem z hlediska použití a údržby, jelikož poskytuje hotové programové vybavení, ale zároveň uživatel nemá právo příliš zasahovat do nastavení.



Obrázek 11 Distribuční modely cloudových technologií
Zdroj: přístup z Internetu: https://www.cloudflare.com/ru-ru/learning/cloud/what-is-the-cloud/

Detailnější charakteristiky každého ze zmíněných modelů jsou představeny níže.

IaaS (Infrastructure as a Service) je model poskytování infrastruktury jako služby, ve které spotřebitel využívá základní výpočetní zdroje pro zpracování a ukládání dat, vytváření sítí a další operace z důvodu nutnosti nasazování a spuštění libovolného softwaru. Uživatel sám řídí spotřebu zdrojů a vyrovnávání zátěže a může si vybrat mezi pronájmem kompletního fyzického nebo virtuálního stroje s možností ukládání dat. V případě malého zatížení, když pronajaté stroje nejsou plně

využívány, některé ze zdrojů mohou být dočasně vypnuty a tím se dá předejít zbytečným nákladům. IaaS model umožňuje pracovat s virtuálními počítači tak, jako by byly umístěny lokálně, ale na rozdíl od pevné fyzické infrastruktury dovoluje využívat benefity škálovatelnosti a zároveň eliminuje veškeré aktivity, spojené s údržbou [42], [44].

Hlavními přínosy modelu IaaS jsou: [45]

- jednoduchost implementace a použití v porovnání s nastavením a správou fyzických serverů,
- vysoká flexibilita, zajištěná pomocí přenositelnosti infrastruktury a škálovatelnosti,
- nákladová efektivita kvůli služby PAYS, nebo Pay as you go, kde uživatel platí jen tolik, kolik zdrojů používá,
- možnost nasazení do cloudu již používaných lokálních softwarových licencí prostřednictvím technologie BYOL, nebo Bring Your Own License, která umožňuje zachovat předchozí investice [46],
- vysoký stupeň spravování výpočetních zdrojů, nedostupný při použití jiných modelů cloudových technologií,
- rychlejší aktualizace stávající infrastruktury, převážně kvůli správě všech serverů na jednom místě.

Ačkoliv se infrastruktura jako služba zdá být perfektním řešením pro ty, co hledají možnosti snížit náklady na provoz vlastních serverů, existuje problém bezpečnosti dat, především z důvodů souvisejících s hardwarem poskytovatele služeb. Odběratel musí si být jist, že jsou stroje kvalitně zabezpečeny. Kromě toho, za účelem předcházení ztrátám údajů probíhá jejich replikace na několika zařízeních. Je nutné si zajistit zničení dat po ukončení spolupráce s dodavatelem. Nezbytně nutným krokem před migrací do cloudových technologií je pečlivé prozkoumání podmínek poskytování služby a uzavření dobře rozmyšlené smlouvy.

Další nevýhody a rizika spojená s použitím IaaS jsou: [43]

- je nutné investovat do serverového softwaru,

- nedostatek standardizovaného rozhraní, které by umožnilo snadný přenos dat a služeb z infrastruktury jednoho cloudu do jiného v případě změny poskytovatele,
- problémy, spojené s integrací několika platforem od různých dodavatelů.

PaaS (Platform as a Service) je model poskytování platformy jako služby, který je možné považovat za nadstavbu IaaS. Mezi závazky poskytovatele PaaS patří udržování nejenom cloudové infrastruktury, ale také operačního systému a softwaru, který slouží jako podpora uživatelských aplikací, zatímco uživatel služby PaaS je zodpovědný za vývoj, instalaci a údržbu vlastního programového vybavení. V případě IaaS modelu jsou předmětem smlouvy stroje a jejich výpočetní výkon, zatímco platforma jako služba nabízí kompletní hotové prostředí a zbavuje zákazníka nutnosti spravovat infrastrukturu [44].

Výhody, které nabízí model PaaS, jsou: [47]

- jednodušší správa prostředí v porovnání s infrastrukturou jako službou,
- relativně rychlá doba spuštění softwaru nebo aplikace,
- uživatel má nižší zodpovědnost v porovnání s IaaS modelem,
- technologie a nástroje, podporující škálovatelnost, které pomáhají optimalizovat náklady,
- možnost využití technologie BYOL u některých dodavatelů [46],
- odpadá nutnost investovat do serverového softwaru.

Výhody platformy jako služby v některých případech, zvlášť v porovnání s infrastrukturou jako službou, stávají se její nedostatky. Poměrná jednoduchost použití a nižší zodpovědnost jsou důsledky omezeného přístupu ke konfiguraci prostředí. Uživatel se musí přizpůsobit nabídce poskytovatele služby a vycházet z jeho možností, což přináší riziko narazit na úzká hrdla služby. Další technické problémy mohou nastat během migrace nebo při přechodu z vlastní infrastruktury na službu PaaS, která se strukturálně výrazně liší. Nahrazení jedné cloudové platformy jinou může být obtížné, protože platformy různých dodavatelů nemusí

být zcela nebo dokonce částečně kompatibilní. Kromě toho, ne každý provozovatel cloudu dovoluje přenést lokální licence na svoje virtuální stroje a také škálovatelnost PaaS je nižší, než u IaaS.

Na základě výše uvedených vlastností lze konstatovat, že platforma jako služba je nejvhodnější volbou pro rychle rostoucí podniky, které nemají finance pro zajištění značné počáteční investice jak do zařízení, tak i do lidských zdrojů, ale mají vysoké nároky na výpočetní výkon, přičemž bezpečnostní předpisy nebrání v používání cloudových technologií.

SaaS (Software as a Service) je model poskytování softwaru jako služby, který je nejznámějším typem využití cloudových technologií. Zatímco IaaS a PaaS jsou především určeny pro IT specialisty, s modelem SaaS se setkává každý běžný uživatel Internetu, dokonce i na denní bázi. Nejznámějším příkladem mohou sloužit služby, nabízené společností Google: Google Docs, Google Drive, Google Calendar a jiné. Kromě toho, podle modelu SaaS také fungují sociální sítě a platformy pro vytváření webových stránek. Některé poskytovatele takové služby mohou nabídnout zákazníkovi dokonce i zveřejnění hotového webu na svém serveru.

Z technického hlediska software jako služba je technologická platforma, která poskytuje dostupnost aplikace přes Internet. Uživatel nekontroluje backend infrastrukturu včetně sítě, operačních systémů, systémů pro ukládání dat, jako je, například, při klasické instalaci softwaru. Obchodní model SaaS pro konečného příjemce zjednodušuje způsob práce a instalace nástrojů na počítač a nabízí vždy nejnovější verze a aktualizace. Všechna data jsou uložená v cloudu, což řeší problém nutného udržování dat na speciálních lokálních serverech s velkou kapacitou. Značnou změnu do metod investování přináší způsoby platby: SaaS řešení se poskytují zdarma, podle spotřeby nebo také na základě předplatného na určitou dobu, přičemž cena zahrnuje veškerou údržbu aplikačního softwaru. Pokud jde o klasický model nákupu trvalých licencí, vrácení peněz není možné bez ohledu na to, zda je aplikace používána nebo ne. Mezi nejčastější způsoby využití SaaS firmami patří organizace videokonferencí, přechod na cloudovou poštu, migrace do cloudu podnikových informačních systémů, jako jsou systémy pro řízení vztahů se zákazníky, aplikace pro automatizaci obchodních procesů, služby, podporující projektové řízení a plánování času [44].

Výhody řešení SaaS pro firmy: [42], [48]

- dostupnost přes Internet bez ohledu na čas a geografickou polohu,
- žádná závislost na budování a správě vlastní infrastruktury, modernizace fyzických strojů a programového vybavení je na straně poskytovatele,
- k dispozici je poměrně rychlá škálovatelnost díky jednoduchosti zakoupení dodatečných zdrojů,
- různé úrovně uživatelského přístupu a oprávnění,
- komplexní servis a podpora ze strany dodavatele služby.

Potenciální rizika, spojená s použitím modelu SaaS, vychází z obavy z nedostatečného zabezpečení ochrany dat, což je hlavním rozhodujícím faktorem při vyhýbaní cloudovým technologiím. Řešením je volba spolehlivého dodavatele služby a pečlivě sestavená smlouva.

8.1 Možnosti využití cloudových technologií v podnikání

Cloudové technologie v podnikání je možné využit k řešení velkého množství problémů. Každá firma si nastavuje procesy a infrastrukturu podle svých požadavků a tím pádem mohou vzniknout komplexní hybridní systémy, které jsou unikátním výsledkem lidské práce. Nicméně, dá se vyčlenit několik obecných způsobů využití cloudových technologií.

Úložiště pro zálohu dat

Nejjednodušším příkladem toho, jak cloudové technologie mohou přispět bezpečnosti dat podniku, je přenos úložiště pro zálohy do cloudu. Jak už bylo řečeno dříve, provozovatele těchto služeb mají za cíl zajistit nedotknutelnost uživatelských informací, a proto se snaží maximálně eliminovat rizika ztráty. Ukládání kritických dat do vzdálených prostorů, které používají šifrování, může zaručit vysokou bezpečnost. V případě, například, vývojové společnosti, která na denní bázi používá složitou lokální infrastrukturu, celé toto řešení může byt uloženo do cloudu a v případě selhání osobního systému rychle obnoveno ze vzdáleného úložiště [49].

Elektronická pošta

Na první pohled se může zdát, že migrace elektronické pošty do cloudu je riskantní nápad. Nicméně v současné době velké množství podniků již používá cloudová řešení pro organizace firemní korespondence. Důvody této volby spočívají především ve dvou aspektech: snížení nákladů na údržbu vlastního serveru a zároveň zvýšení bezpečnosti dat. Nehledě na to, že podvědomá nedůvěra ve spolehlivost cloudových technologií je stále rozšířeným jevem, tyto obavy jsou převážně zbytečné. Detaily zajištění bezpečnosti se liší podle poskytovatele, ale mezi příznaky dobrého produktu patří použití šifrovaného spojení, zálohování, kontrola pošty pomocí vestavěného antiviru. Jeden z nejznámějších softwaru v tomto segmentu je Microsoft 365, který splňuje všechny bezpečnostní požadavky ve Spojených státech amerických [50].

Platforma pro špičkové zatížení

V případě, když druh podnikání buď má výraznou sezónnost, nebo pro něj je typické střídání období zvýšeného a rutinního zatížení, částečná migrace služeb do cloudu je schopná snížit náklady, které vznikají v důsledku prostoje. Příkladem může být lokální server internetového obchodu dětských hraček, který sotva zvládá předvánoční období. Příprava a zapojení nového stroje vyžadují investování do aktualizace již stávající infrastruktury. Druhý server je zřejmé řešení, ale problém nastává po svátcích, když nový stroj se stává nepotřebným na několik měsíců. S časem počítač stárne z technického hlediska a zároveň se kumulují náklady obětované příležitosti. Jiný scénář se odehraje, pokud správce obchodu využije cloudové technologie a pronajme na pár měsíců moderní virtuální stroj, na kterém spustí dočasný druhý server a zaplatí jenom za reálnou spotřebu. Stejný princip lze uplatnit, pokud kolísání zatížení se významně liší během dne [50].

Migrace částí nebo celé infrastruktury do cloudu

Vybudovat lokální infrastrukturu, která bude levná, odolná vůči chybám a

výpadkům a zároveň snadno udržitelná, není jednoduchá záležitost. Firma se musí sama postarat o všechny aspekty, včetně opotřebení zařízení, systému chlazení, fyzické bezpečnosti serverovny a ochrany dat, což v případě velkého podniku vyžaduje značné investice. Cloudové služby pomáhají přerozdělit zatížení a přenést aspoň část zodpovědností na poskytovatele, který je profesionálem v činnostech, které jsou jenom podpůrnými pro jejich klienty [42].

9 Cloudové technologie podporující testování během vývoje softwaru

K vytvoření této kapitoly byly použity informace z oficiální dokumentace následujících produktů: GitHub [51], GitLab [37], Google Cloud Platdorm [52], Kubernetes [53]. Ukázkový projekt X je případovou studií, která má za cíl ukázat, jak by mohlo probíhat testování malého a většího projektů v praxi. Příklad se odvíjí od reálných dat, zjištěných ve spolupráci se zaměstnanci firmy Quadient na pobočce v Hradci Králové.

V případě společnosti, která se zabývá vývojem softwaru, cloudové technologie mohou být významným pomocníkem v organizaci firemních procesů. Kromě zavedení výše zmíněných scénářů, cloud se může stát nedílnou součástí vývoje a testování aplikace, přičemž existuje velké množství odlišných způsobů integrace. Návrh infrastruktury není možný bez ohledu na konkrétní případ, a proto je potřeba stanovit balík požadavků, který bude výchozím bodem. Kromě toho, nabízené řešení, zavedené do provozu, často může vyžadovat provedení změn, převážně v důsledku buď selhání přípravné fáze, nebo měnících se okolností. Dále jako příklady budou uvedeny odlišné podmínky projektu, který se vyvíjí v čase a postupně roste.

Existují odlišné metody, jak definovat rozměr a pracnost softwarového projektu. Metoda LOC (zkrátka od *lines of code*) je založena na počítání řádků zdrojového kódu. Metoda funkčních bodů (FPA) umožňuje stanovit odhad projektu na základě analýzy funkcí, ke kterým uživatel bude mít dostup. Tyto dva přístupy jsou nejrozšířenější, ale operují s odlišnými metrikami. Fyzické počítání rozsahu kódu je nejpřehlednějším způsobem, jak ukázat velikost softwarového projektu z technického hlediska [54, s. 440]. Nicméně, tento přístup neukazuje přínos systému pro uživatele. Metoda funkčních bodů je pružnější a používá technický odhad množství zpracovaných dat a nabízených funkcí a zároveň analyzuje jejich použitelnost [54, s. 438].

Podle metody FPA je ukázkový projekt X na začátku vývoje malý v důsledku menšího počtu nabízené funkcionality.

Projekt X se zabývá vytvářením webové aplikace, která je určena pro distribuci na základě B2B konceptu, když jiná společnost po zakoupení licence integruje systém do své infrastruktury.

Cílem vývojového týmu projektu X je poskytování kvalitního softwaru a jeho pravidelná aktualizace, která umožní neustále rozvíjet a zlepšovat již jednou vyvinutou aplikační logiku. Se zaměřením na dlouhodobý rozvoj bylo zvoleno odmítnutí klasického vodopádového modelu a uplatnění koncepce agilního vývoje a metodiky Scrum. Již od začátku práce nad projektem se vývojový tým skládá nejenom z programátorů, ale také z testerů. Vývojové období pro zjednodušení kontroly progresu je rozděleno na dvoutýdenní sprinty. Vývojáři dostávají od manažerů požadavky na vylepšení, uložené jako story do backlogu, a berou je do závazku před začátkem nového sprintu na základě svých kapacit.

Aktivity na podporu zajištění kvality aplikace začínají mnohem dříve, než se nová verze produktu dostane k testerovi. Vývojové procesy jsou nastaveny v souladu se zásadami extrémního programování, což převážně znamená zavedení programování řízeného testy, provádění menšího testování před nahráním změn v kódu do repozitáře a časté sjednocení kódu. Tester kontroluje chování aplikace jako celku.

Za účelem zajištění nejlepší synchronizace, kvality aplikace a úspory času, pro projekt X byly zvoleny principy CI/CD, které vyžadují vysokou míru automatizace v testování pro jejich uplatnění. Není možné zautomatizovat všechny typy testů, zvlášť když se jedná o lidský faktor, jako v případě testování bezpečnosti nebo UX. Nicméně, manuální kontrola funkcionality se ve mnoha případech může nahradit automatickým testem. Vývoj nových vlastností probíhá v kopii stabilního zdrojového kódu, aby případné škodlivé změny nezasáhly funkční verzi. Před sjednocením verzí je nutné provést systémové testování. Testeři minimálně jednou denně spouští sadu systémových testů, aby drobné změny neovlivnily další prvky. Tyto procesy musí probíhat na vyhrazených strojích, ke kterým má přístup každý člen týmu.

Na začátku životního cyklu aplikace se vývojový tým skládá ze třech programátorů a dvou testerů.

Následující kalkulace se zabývá jen systémovým testováním v rámci procesů CI/CD a má za úkol ukázat varianty řešení tohoto úkolu.

9.1 Organizace testování v rámci CI/CD malého projektu

Nehledě na poměrně malý rozměr projektu X, spolupráce několika lidí musí být co nejlépe koordinována. Krátké synchronizační schůzky a denní scrum umožňují vytvořit přehled o situaci v reálném čase. Nicméně, kromě lidské komunikace musí být zavedeno schéma, jak sjednotit fyzický výsledek práce několika lidí, jinak řečeno jak organizovat integraci změn v kódu a zároveň vytvořit bezpečné prostředí, které umožní návrat do předchozí stabilní verze. Zmíněné problémy se dají naplnit použitím jediného verzovacího systému. V moderním světě bude nejlepší volbou distribuovaná platforma, jelikož kromě většího pohodlí v užívání poskytuje optimálnější kombinaci zabezpečení a organizaci místa. Dobrou praxí je používání platformy Git. Její popularita přináší hned několik výhod:

- vysoký stupeň kompatibility s velkým množstvím softwaru,
- dobrá dokumentace a podpora, také ze strany vývojářské komunity.

Dalším softwarem, schopným podpořit vývojový tým, jsou webové služby, jako jsou GitHub a GitLab. Tyto dva produkty nabízí podobné možnosti, ale liší se ve finanční politice a otázkách zajištění bezpečnosti. GitHub je největší platformou, poskytující hosting IT projektů, ale hlavní důraz klade na pohodlí při práci s veřejnými repozitáří vlastních a open-source projektů. V případě ziskové firmy, která nechce, aby její zdrojový kód byl přístupný pro neoprávněné uživatele, taková firma se musí počítat s tím, že funkcionalita GitHubu, zajišťující automatizaci vývojových procesů (podpora CI/CD), bude zpoplatněna 90Kč (4 USD) za uživatele/měsíc. Veřejně dostupné repozitáře jsou zdarma. V případě projektu X z důvodu obchodního tajemství není možné, aby aplikační logika byla otevřená, takže pro integraci s platformou GitHub tým z pěti členů bude mít měsíční náklady: $90 \times 5 = 450$Kč (20 USD).

GitLab má jinou politiku a nehledě na slevy pro veřejné projekty obecná pravidla financování vychází z rozsahu poskytnuté funkcionality. Základní tarif licence je zdarma, ale nabízí na realizaci CI/CD 400 minut měsíčně. Nicméně, tato limitace je

aktuální jenom v případě použití sdílených spouštěčů úkolů (*GitLab runners*), poskytnutých platformou, zatímco se práce vlastních neúčtuje a tím pádem vývojářský tým může dostat neomezený čas s menšími výdaji. Problém spočívá v tom, že personální agent vyžaduje místo a počítačový výkon, buď na lokálním serveru nebo na stroji v cloudu, ale zároveň je nastavitelný. Existují řešení, jak technicky optimalizovat vynaložení minut, ale v případě, pokud všechny způsoby selžou a provoz vlastních agentů není relevantní, vždy je možnost zakoupení dodatečného času 224Kč (10 USD)/1000 minut.

Nehledě na to, že existuje velké množství produktů podporující CI/CD, a dříve již byly také zmíněny Jenkins a TeamCity, GitLab má přednost pro projekt X. Důvodem je, že tento produkt nabízí podporu všech fází DevOps cyklu a tím pádem může mít tým všechny potřebné nástroje na jednom místě.

GitLab nabízí dvě formy použití: SaaS a lokální instalaci. V případě použití platformy na základě cloudových technologií, repozitář s celým stromem verzí se bude nacházet na serverech GitLabu. Lokální instance nabízí mnohem větší flexibilitu v nastaveních, ale vyžaduje od správce zkušenosti se systémem Linux. Výběr, kde uchovávat zdroje, zaleží minimálně na dvou faktorech:

- bezpečnostních předpisů ohledně umístění kódu,
- požadavcích ohledně nastavení.

V případě projektu X neexistují žádné limitace v tom, kde se bude nacházet a také nejsou žádné specifické požadavky na vlastnosti serveru.

Tabulka č. 3 znázorňuje průměrný odhad každodenní časové náročnosti na provedení systémových testů, kde četnost spuštění vychází z jedné povinné kontroly kvality denně a předpokladu, že vývojáři kontrolují výsledek své práce také jednou za den.

Tabulka 3 Denní odhad testovacího času pro malý projekt

Typ testů	*Délka, v minutách*	*Četnost spuštění*	*Souhrnný čas, v minutách*
Systémové	40	2	80

Zdroj: vlastní zpracování

Po sumarizaci je zřejmé, že čas strávený testováním během měsíce, činí přibližně (v minutách):

$$80 \times 25 = 2000.$$

Na základě těchto výpočtů je možné navrhnout uživatelské scénáře toho, jak bude probíhat práce týmu s GitLabem.

V případě použití GitLabu jako SaaS vývojový tým nemusí hradit žádné poplatky za software, pokud se vystačí se základním tarifem, možnostmi poskytnutých platformou agentů a 400 minutami na podporu CI/CD měsíčně. Nicméně, výpočty ukazují, že čas, potřebný pro projekt X, činí 2000 minut jenom pro systémové testování, a to vyžaduje zakoupení dodatečných možností. Zakoupené minuty se kumulují na účtu a jsou platné rok, což znamená, že je možné sestavit rovnici, která pomůže optimalizovat roční náklady:

$2000 \times 12 = 400 \times 12 + x$,

kde neznámé x je minimální počet minut, které je nutné dokoupit.

Po vyřešení rovnice dostáváme $x = 19200$, což odpovídá dvaceti nákupům dodatečného času. Odhad ročních nákladů na systémové testování při realizaci této strategie činí přibližně 4486Kč (200 USD).

Tarif Starter nabízí více možností a 2000 minut na podporu CI/CD měsíčně, ale také je zpoplatněn 90Kč (4 USD)/uživatel/měsíc, což pro pětičlenný tým odpovídá přibližně 5383Kč (240 USD)/rok. Nicméně, celková měsíční spotřeba překračuje limity, jelikož je nutné se počítat s tím, že vývojáři také budou čerpat čas na provedení úkolů, předcházejících testování, například na vytváření verze. Pokračování v odhadu a další kalkulace není relevantní, jelikož již teď je zřejmé, že základní tarif pro nenáročný tým je finančně výhodnější.

V případě vytváření lokální infrastruktury provozující CI/CD, povinnými prvky jsou počítač, který bude serverem a úložištěm repozitáře a lokální nebo celosvětová síť, která umožní vývojovému týmu přístup k serveru. Předpokládá se, že náklady na síťové připojení patří do provozních a proto nebudou zahrnuty do následující kalkulace.

GitLab uvádí minimální požadavky na hardware v případě instalace lokálního serveru, které jsou uvedeny v tabulce č. 4.

Tabulka 4 Požadavky na instalaci GitLab serveru

Komponenta	*Doporučení*
Disk	SSD nebo HDD 7200 rpm
CPU	4 jádra do 500 uživatelů
Operační paměť	4GB

Zdroj: vlastní zpracování na základě dokumentace GitLab

Tabulka č. 5 obsahuje návrh počítače na provoz lokální infrastruktury. Náklady vychází z průměrných cen na českém trhu.

Tabulka 5 Návrh serveru malého projektu

	Množství	**Náklady v Kč, bez DPH**
HP PRO 300 G3 SSD, 256GB Intel Core i5, 9400(2.9/4.1GHz 6jader/6vláken) 8GB RAM (DDR4 DIMM) Požadavky na napájení: 180W	1	12000

Zdroj: vlastní zpracování na základě internetového portálu https://www.czc.cz/

Vlastní instalace GitLab serveru a repozitářů na komerční virtuální stroj podle svojí volby není racionální, jelikož toto nahradí všechny benefity platformy, poskytované na principu cloudových technologií. Nicméně, je důležité zvážit, kde přesně budou probíhat procesy CI/CD, resp. zda lokálně nebo v cloudu. Stojí za zmínku, že obě varianty jsou podporovány jak vlastním serverem, tak i SaaS, ale agent GitLab by neměl být pouštěn na stroji, kde je běžící GitLab instance (server). V případě lokálního spuštění lze použít počítač, navržený v tabulce č. 5.

Migrace procesů CI/CD do cloudu umožňuje škálování na základě reálné zátěže. V situaci, kde nad jedním produktem pracuje několik lidí ve stejný čas a přitom sdělí výpočetní výkon jednoho integračního stroje, může dojít k prostoji v práci, jelikož ne všechny úkoly umožňují jejich paralelní zpracování a uživatel bude nucen čekat, než se počítač uvolní. Předejít tomuto problému je možné buď s pomocí nových

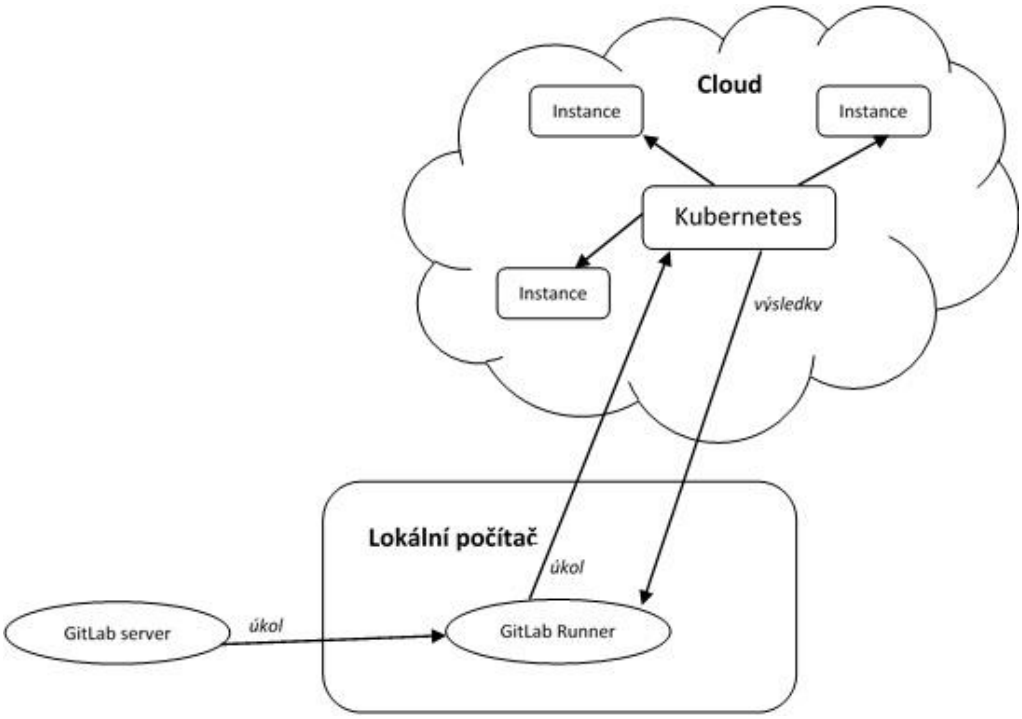
serverů, nebo pronájmu požadovaného výpočetního výkonu v cloudu. Základní myšlenkou je na požadavek vývojáře umožnit automatické spuštění úkolu na novém virtuálním stroji a tím předejít situaci, když několik serverů jsou zbytečně v provozu na případ náhlého růstu zátěže.

GitLab umožňuje spuštění úkolů v cloudu pomocí Docker kontejnerů a je schopen posílat požadavky na nástroj pro jejich orchestraci a škálování - Kubernetes. Princip jeho fungování je znázorněn v následujícím zjednodušeném algoritmu. Prerekvizitou je nastavení budoucího clusteru počítačových instancí a také uvedení povoleného množství kontejnerů na jednom stroji. Tato veličina se odvíjí od charakteristik stroje a požadavku na provoz kontejneru. Pro zjednodušení příkladu platí, že na jedné instanci současně může probíhat jenom jeden proces.

1. Kubernetes dostává požadavek №1 na spuštění kontejneru pro provedení úkolu №1,
2. Kubernetes má přehled o kapacitách a spouští kontejner №1 na stroji №1,
3. Kubernetes dostává požadavek №2 na spuštění kontejneru pro provedení úkolu №2,
4. Kubernetes ověřuje, kolik kontejnerů je spuštěno na stroji №1 (maximálně povolené množství) a spouští kontejner №2 na novém stroji №2.

Po vyplnění úkolů Kubernetes uvolňuje stroje, což znamená, že uživatel služby platí jenom za reálnou spotřebu.

Podobná technologie vyžaduje určitou přípravu na lokálním stroji, jelikož propojení GitLab serveru a Kubernetes je realizována vyhrazeným vlastním GitLab agentem. To znamená, že nehledě na to, kde je umístěn GitLab server, je nutné zahrnout do infrastruktury počítač, který bude provozovat řízení škálování, vizte obrázek č. 12. Ve výsledku finanční náklady budou obsahovat výdaje na podporu vlastního stroje a cloudového výkonu zároveň.



Obrázek 12 Propojení serveru GitLab a cloudu
Zdroj: vlastní zpracování

Projekt, nad kterým pracuje pětičlenný tým a na provedení systémových testů denně potřebuje přibližně 80 minut (vizte tabulka č. 3), sotva se setká s problémem dlouhého čekání na uvolnění serveru. Přesunutí zátěže do cloudu v tomto případě není potřebné a finančně nevýhodné. Za účelem pouštění vlastních GitLab agentů se doporučuje použít fyzický stroj, například navržený v tabulce č. 5.

Souhrnná tabulka č. 6 znázorňuje přibližné náklady na systémové testování pro různé varianty použití možností GitLabu během prvních třech let. Výpočty vycházejí z použití základního tarifu, který se poskytuje bez poplatků; také je nutné počítat s dokoupením chybějících minut v případě použití sdílených agentů, poskytnutých platformou. V ceně jsou zahrnuty přibližné roční náklady na elektřinu, které vycházejí z charakteristiky stroje (vizte tabulka č. 5) a průměrné ceny elektřiny, která činí 4,34 Kč/1 kWh [55]. Server a vlastní GitLab agent nesmí být na jednom stroji.

Tabulka 6 Odhad souhrnných nákladů na systémové testování pro malý projekt, v Kč

1. rok	**Runner GitLab**	**Lokální runner**
GitLab SaaS	4486	18843,312
Lokální server	23329,312	37686,624
2. rok		
GitLab SaaS	4486	6843,312
Lokální server	11329,312	13686,624
3. rok		
GitLab SaaS	4486	6843,312
Lokální server	11329,312	13686,624
Náklady za 3 roky		
GitLab SaaS	13458	32529,936
Lokální server	45987,936	65059,872

Zdroj: vlastní zpracování

Tabulka č. 6 jednoznačně ukazuje, že pro potřeby malého a nenáročného projektu X je finančně nejvýhodnější strategií použít server GitLab jako SaaS a zároveň organizovat systémové testování s pomocí cloudových služeb nabízených platformou. V případě, pokud konfigurace připravených agentů není vyhovující, nejlepší variantou je kombinace SaaS varianty a vlastního agentu.

9.2 Vývoj a testování aplikace středního projektu

Projekt X se úspěšně rozvíjel v čase a dostal se do fáze, když jeden pětičlenný tým se stal součástí čtyř týmů, což v souhrnu dává 20 vývojářů a 8 testerů. Kromě toho, s růstem funkcionality, dokumentace a celkové sofistikovanosti aplikace se změnil čas potřebný na provedení testování. Nový každodenní odhad znázorňuje tabulka č. 7, kde četnost spuštění vychází z předpokladu, že během sprintu vývojáři zkontrolují novou funkcionalita desetkrát, zatímco testeři spustí balíček systémových testů minimálně jednou denně.

Tabulka 7 Denní odhad testovacího času pro střední projekt

Typ testů	*Délka, v minutách*	*Četnost spuštění*	*Souhrnný čas, v minutách*
Systémové	310	2	620

Zdroj: vlastní zpracování

Čas strávený testováním během měsíce činí přibližně (v minutách):

$$620 \times 25 = 15500.$$

Níže je uvedena rovnice, pomocí které je možné zjistit chybějící minuty:

$15500 \times 12 = 400 \times 12 + x$,

kde neznámé x je minimální počet minut, které je nutné dokoupit.

Po vyřešení rovnice dostáváme $x = 181200$, což odpovídá sto osmdesáti dvou nákupům dodatečného času při použití základního tarifu. Odhad ročních nákladů na systémové testování při realizaci této strategie činí přibližně 40821Kč (1820 USD).

V případě použití lokálního serveru GitLab pro větší projekt je doporučeno použít výkonnější stroj. Návrh počítače je znázorněn v tabulce č. 8.

Tabulka 8 Návrh serveru středního projektu

	Množství	**Náklady v Kč, bez DPH**
Lenovo ThinkStation P520c TWR SSD, 256GB Intel Xeon, W-2123(3.6/3.9GHz 4jádra/8vláken) 16GB RAM (DDR4 DIMM) Průměrný výkon: 250W	1	27000
		27000

Zdroj: vlastní zpracování na základě internetového portálu https://www.czc.cz/

Za účelem spuštění vlastního agentu GitLab je možné použít stroj z tabulky č. 5. Nicméně, v důsledku většího počtu vývojářů a úkolů, které jsou schopní řešit současně během jednoho vývojového cyklu, a také vzhledem k dlouhému běhu balíčku systémových testů, je potřeba zvážit, kolik lokálních testovacích strojů potřebuje celý projekt.

Dobrou praxí je spuštění systémových testů jednou za den, nejlépe v noci, aby testeři dostali výsledky testování hned další ráno a měli čas předat případné chyby na opravu. Podobná aktivita musí být automaticky opakovatelná a zároveň nesmí

být přerušena nebo být posunutá v čase v důsledku testování nové funkcionality, jelikož může odhalit problémy v stabilní verzi.

Z vlastní praxe je známo, že většinovou kontrolu chyb nad vyvíjejícími se verzemi týmy provádějí na konci pracovního dne, když byly doladěny poslední změny. Nicméně, není možné vyloučit lidský faktor a možnou flexibilní pracovní dobu a také technické problémy. Představme si, že se vývojář z týmu A rozhodl spustit systémové testy nad verzí svého týmu v devět večer. Není možné použít hlavní testovací server, jelikož to posune pravidelné testování, takže je použit další stroj. Půlhodiny poté se vývojář z týmu B rozhodl otestovat novou funkcionalitu, nad kterou pracuje jeho tým a musí se počítat s tím, že při použití druhého testovacího serveru v nejlepším případě výsledky dostane až kolem osmé hodiny ráno. To znamená, že jiný kolega z týmu B, který začíná svůj pracovní den v šest ráno, bude muset čekat na výsledky testování. Aby se předešlo takovým zbytečným čekáním, je vhodné použít tři testovací stroje: jeden je vyhrazen pro testery a dva jsou pro vývojáře. Problém je v tom, že uvedená situace nemusí být běžná, zatímco tři počítače budou vždy aktivní na případ náhlého růstu zátěže, nehledě na to, zda testování probíhá či ne. Pravděpodobnost toho, že tři nebo čtyři týmy budou potřebovat spustit testovací balíček najednou, je velmi malá, takže další rozšíření infrastruktury není relevantní.

Náklady na provoz zbytečných strojů lze minimalizovat pomocí cloudových technologií. GitLab uvádí charakteristiky počítačů v Google Cloud Platform, na kterých probíhají procesy CI/CD v případě použití agentů, poskytnutých tímto systémem. Stejné stroje je možné použít i pro vlastní potřeby v případě migrace systémového testování do Google Cloud Platform. Tabulka č. 9 obsahuje vlastnosti stroje typu n1-standard-2 a také ilustruje přibližnou měsíční kalkulaci v závislosti na množství použitých počítačů v clusteru. Předpokládá se, že každý stroj je používán šest hodin denně a dvacet pět dní v měsíci. V důsledku použití služby Pay as you go výsledné náklady mohou být jak nižší, tak i vyšší. Data z tabulky č. 9 jsou orientační.

Tabulka 9 Odhad měsíčních nákladů na provoz cloudové infrastruktury středního projektu

	Množství instancí v jednom clusteru	Cena, v Kč
N1-standard-2 vCPUs: 2 (2 jádra) 8GB RAM Windows licence Umístění ve Finsku 6 aktivních hodin denně	1	318,44
	2	636,88
	3	955,32

Zdroj: vlastní zpracování na základě dokumentace Google Cloud Platform

Souhrnná tabulka č. 10 znázorňuje přibližné náklady na systémové testování středního projektu X pro různé varianty použití možností GitLabu během prvních třech let. Stejně, jako v případě malého projektu, výpočty vycházejí z použití základního tarifu, který se poskytuje zdarma; také je nutné počítat s dokoupením chybějících minut v případě použití agentů poskytnutých platformou. V ceně jsou zahrnuty přibližné roční náklady na elektřinu, které vycházejí z charakteristiky stroje pro spuštění vlastního agentů GitLab runner (vizte tabulka č. 5), průměrného výkonu lokálního serveru na úrovni 250W (vizte tabulka č. 8) a průměrné ceny elektřiny, která činí 4,34 Kč/1 kWh [55]. Server a vlastní GitLab agent nesmějí být na jednom stroji. V lokální infrastruktuře se používají tři testovací počítače. Pro řízení clusteru v cloudu je nutné připravit samostatný stroj s agentem GitLab runner. Tabulka č. 10 obsahuje dvě kalkulace pro cloudové řešení: pro maximální zatížení, když v provozu jsou tři instance, a také pro minimální výkon s použitím jednoho stroje. Předpokládá se, že každá instance je v provozu šest hodin denně dvacet pět dní měsíčně.

Tabulka 10 Odhad souhrnných nákladů na systémové testování pro střední projekt, v Kč

1. rok	Runner GitLab	Lokální runner	Testy v cloudu, 3 instance	Testy v cloudu, 1 instance
GitLab SaaS	40821	56529,936	30307,152	22664,592
Lokální server	77325,6	93034,536	66811,752	59169,192
2. rok				
GitLab SaaS	40821	6843,312	18307,152	10664,592
Lokální server	50325,6	30034,536	27811,752	20169,192
3. rok				
GitLab SaaS	40821	6843,312	18307,152	10664,592
Lokální server	50325,6	30034,536	27811,752	20169,192
Náklady za 3 roky				
GitLab SaaS	122463	70216,56	66921,456	43993,776
Lokální server	177976,8	153103,608	122435,256	99507,576

Zdroj: vlastní zpracování

Tabulka č. 10 ukazuje, že náklady na provoz třech instancí v cloudovém clusteru jsou nejnižší bez ohledu na to, kde bude rozhodnuto provozovat server GitLab. Menší zatížení umožňuje používat menší počet instancí, což sníží náklady. Provoz lokální infrastruktury bez zapojení cloudových technologií je finančně nevýhodný.

10 Shrnutí výsledků

Kapitola č. 9 obsahuje praktickou ukázku použití cloudových technologií v softwarovém testování. Byly stanoveny vlastnosti příkladového projektu s ohledem na jeho vývoj v čase včetně firemních procesů a časových nákladů na provedení systémového testování. Na základě těchto charakteristik pro každý případ byla navržena infrastruktura, potřebná k provedení kontroly kvality aplikace z hlediska testera. Byla provedena kalkulace základních finančních nákladů pro každou strategii a na jejich analýze byla vybrána nejvhodnější varianta.

Pro malý projekt, který používá pro podporu CI/CD platformu GitLab a nevyžaduje umístění zdrojového kódu lokálně, bylo zjištěno:

1. migrace systémového testování do vlastní cloudové infrastruktury není finančně výhodná. Důvod spočívá ve specifických požadavcích na nastavení možností škálování výkonu v cloudu, definované systémem GitLab,
2. nejpříznivější volbou pro tento projekt je použití produktu GitLab jako SaaS v kombinaci s agenty, které jsou předem dané platformou a jejich práce je realizována pomocí Google Cloud Platform.

Pro střední projekt, který používá pro podporu CI/CD platformu GitLab a nevyžaduje umístění zdrojového kódu lokálně, bylo zjištěno:

1. migrace systémového testování do vlastní cloudové infrastruktury na základě Google Cloud Platform v kombinaci s produktem GitLab jako SaaS je finančně výhodná,
2. mezi nejméně výhodné strategie patří použití přednastavených agentů GitLab, provozovaných na základě Google Cloud Platform v kombinaci s provozováním vlastního serveru.

Je nutné zmínit, že závěry jsou relevantní výhradně pro příkladový projekt X. Nicméně, postupy na provedení podobné analýzy jsou schopné být nápomocnými pro jakoukoliv aplikaci, která se nachází ve fázi vývoje. Důležité je překonat předsudky vůči cloudu a zvážit jeho možnosti.

11 Závěry a doporučení

V této práci byla provedena analýza relevantních zdrojů za účelem sjednocení a poskytnutí přehledu o softwarovém testování s použitím cloudových technologií. V důsledku vysokého zájmu o uplatnění metodik agilního vývoje, byla provedena jejich analýza, která umožnila ukázat význam testování a jeho zaražení do firemních procesů. Na ukázkovém příkladu byly znázorněny varianty použití cloudových technologií v testování a jejich přínos, včetně finančního.

Obsah práce splňuje stanovené dílčí cíle:

1. byly vysvětleny základní principy a typy testování softwarových aplikací,
2. byl poskytnut přehled o zařazení aktivit kontroly kvality do firemních procesů a uvnitř vývojového týmu, který používá agilní metodiky vývoje,
3. byla provedena analýza možností podpory systémového testování pomocí cloudových technologií,
4. byly zformulovány požadavky ukázkového projektu a na jejich základě byla provedena analýza finančních nákladů pro různé strategie alokace zdrojů pro podporu testování.

Cloudové technologie jsou relativně mladé a zatím se používají za účelem testování softwarových aplikací jen v nejmodernějších společnostech. Jedním z hlavních důvodů je nedostatek důvěry ohledně bezpečnosti dat v cloudu. Tento problém se dá vyřešit volbou spolehlivého poskytovatele služby.

Konstatovat, že cloudové technologie plně nahradí časem ověřené principy, není možné. Nicméně, provozovatelé cloudových služeb se neustále zlepšují a již v současné době nabízí širokou sadu nápomocných nástrojů. Hlavním doporučením této diplomové práce je zvážit všechny možnosti ohledně organizace vlastní testovací infrastruktury a neodmítat cloudové technologie z důvodu jejich neobvyklosti. Dominantním benefitem cloudu je jeho jednoduchá škálovatelnost, která umožňuje výrazně snížit náklady na provoz vlastního pevně daného testovacího clusteru.

Seznam použité literatury

1. Wikipedie, otevřená encyklopedie. Přístup z Internetu: https://en.wikipedia.org/wiki/App_Store_(iOS)#Number_of_iPad_applications Citováno dne: 15.01.2020

2. Daxx Team. How Many Software Developers Are in the US and the World? Přístup z Internetu: https://www.daxx.com/blog/development-trends/number-software-developers-world Citováno dne: 15.01.2020

3. Abel Avram. IDC Study: How Many Software Developers Are Out There? Datum publikace: 31.01.2014. Přístup z Internetu: https://www.infoq.com/news/2014/01/IDC-software-developers/

4. Mahesh Chand. How Many Software Developers Are There In The World? Datum publikace: 15.01.2019. Přístup z Internetu: https://www.c-sharpcorner.com/article/how-many-software-developers-are-there-in-the-world/

5. Ham Vocke. The Practical Test Pyramid. Datum publikace: 26.02.2018. Přístup z Internetu: https://martinfowler.com/articles/practical-test-pyramid.html

6. Wikipedie, otevřená encyklopedie. Přístup z Internetu: https://en.wikipedia.org/wiki/Kent_Beck Citováno dne: 17.01.2020

7. Portál Wiki. Never Write a Line Of Code Without a Failing Test. Přístup z Internetu: http://wiki.c2.com/?NeverWriteaLineOfCodeWithoutaFailingTest Citováno dne: 18.01.2020

8. BECK, Kent. Test-Driven Development By Example. 1st ed., Addison-Wesley Professional, 2002, 220 s. ISBN-10: 9780321146533.

9. Sushma S. The Difference in Perspective of “Testers” and “Developers”. Datum publikace: 1.09.2020. Přístup z Internetu: https://www.softwaretestinghelp.com/the-difference-in-perspective-of-testers-and-developers/amp/#A_Good_Tester_and_A_Good_Developer

10. KULIKOV, Sv. Testirovanije programmnogo obespečenija. Bazovy kurs. 3. vyd., datum publikace: 16.10.2020, 298 s. Přístup z Internetu: http://svyatoslav.biz/software_testing_book/

11. STF. Software Quality Assurance vs Software Quality Control. Datum publikace: 6.09.2020. Přístup z Internetu: https://softwaretestingfundamentals.com/sqa-vs-sqc/

12. STF. Verification vs Validation. Datum publikace: 10.09.2020. Přístup z Internetu: https://softwaretestingfundamentals.com/verification-vs-validation/

13. SAVIN R. Testirovanie Dot Kom. Moskva: Delo, 2007, 312 s. ISBN 978-5-7749-0460-0

14. Dymovoe testirovanie ili Smoke testing. Přístup z Internetu: http://www.protesting.ru/testing/types/smoke.html Citováno dne: 19.01.2020

15. Regressionnoe testirovanie ili Regression testing. Přístup z Internetu: http://www.protesting.ru/testing/types/regression.html Citováno dne: 19.01.2020

16. PATTON, Ron. Software Testing. 1st ed., Sams Publishing, 2001, 389 s. ISBN 0-672-31983-7

17. Oficiální dokumentace aplikačního rámce JUnit. Přístup z Internetu: https://junit.org/junit4/javadoc/4.12/org/junit/package-summary.html Citováno dne: 20.01.2020

18. What Is User Acceptance Testing (UAT): A Complete Guide. Datum publikace: 1.09.2020. Přístup z Internetu: https://www.softwaretestinghelp.com/what-is-user-acceptance-testing-uat/

19. DUSTING Elfriede., RASHKA Jeff, PAUL John. Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance. Perevod na russky jazyk. Lori, 2003, 567 s. ISBN-10: 0201432870

20. CRISPIN Lisa, GREGORY Janet. Agile Testing. A Practical Guide for Testers and Agile Teams. Perevod na russky yazyk. Moskva, I.D. Viljams, 2010, 464 s. ISBN: 978-5-8459-1625-9 (rus). ISBN: 978-0-321-53446-0 (eng)

21. Grebenyuk Viktor Mikhaylovich. Test automation rationale. Moscow State Technical University of Radioengineering, Electronics and Automation, MSTU MIREA. Internet-časopis NAUKOVEDENIE №1, 2013

22. Oficiální dokumentace produktu SoapUI. Přístup z Internetu: https://www.soapui.org/learn/api/soap-vs-rest-api/ Citováno dne: 21.01.2020

23. PAUTASSO Cesare. REST vs. SOAP (Making the Right Architectural Decision) - 1st International SOA Symposium, Amsterdam, October 2008. Přístup z Internetu: https://www.slideshare.net/cesare.pautasso/rest-vs-soap-making-the-right-architectural-decision-1st-international-soa-symposium-amsterdam-october-2008-presentation

24. Oficiální dokumentace produktu SoapUI. Přístup u Internetu: https://www.soapui.org/learn/api/rest-request-method-verbs/ Citováno dne: 22.01.2020

25. FIELDING, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

26. Oficiální dokumentace produktu Postman. Přístup z Internetu: https://learning.postman.com/docs/getting-started/introduction/ Citováno dne: 23.01.2020

27. Oficiální dokumentace produktu JMeter. Přístup z Internetu: https://cwiki.apache.org/confluence/display/JMETER/Home Citováno dne: 23.01.2020

28. Oficiální dokumentace produktu Selenium. Přístup z Internetu: https://www.selenium.dev/documentation/en/ Citováno dne: 23.01.2020

29. KNIBERG Henrik. Agile Product Ownership in a nutshell. Datum publikace: 25.10.2012. Přístup z Internetu: https://blog.crisp.se/2012/10/25/henrikkniberg/agile-product-ownership-in-a-nutshell

30. BECK Kent. Manifesto for Agile Software Development. Rok publikace: 2001. Přístup z Internetu: https://agilemanifesto.org/principles.html

31. STELLMAN Andrew & GREENE Jennifer. Learning Agile. Understanding Scrum, XP, Lean, And Kanban. 1st ed., O`Reilly, 2015, 397 s. ISBN: 978-1-449-33192-4

32. PETERSON Aleks. Trello vs. JIRA: Choosing an Agile Project Management Tool. Datum publikace: 20.05.2019. Přístup z Internetu: https://technologyadvice.com/blog/information-technology/trello-vs-jira-choosing-an-agile-project-management-tool/

33. FOWLER, Martin. Continuous Integration. Datum publikace: 1.06.2006. Přístup z Internetu: https://www.martinfowler.com/articles/continuousIntegration.html

34. HUMBLE Jez, FARLEY David. Continuous Delivery. Perevod na russky jazyk. Moskva, I.D. Viljams, 2011, 428 s. ISBN: 978-5-8459-1739-3 (rus). ISBN: 978-0-321-60191-9 (eng)

35. Oficiální dokumentace produktu Jenkins. Přístup z Internetu: https://www.jenkins.io/doc/ Citováno dne: 24.01.2020

36. Oficiální dokumentace produktu TeamCity. Přístup z Internetu: https://www.jetbrains.com/help/teamcity/teamcity-documentation.html Citováno dne: 24.01.2020

37. Oficiální dokumentace produktu GitLab. Přístup z Internetu: https://docs.gitlab.com/ Citováno dne: 28.09.2020

38. MELL Peter, GRANCE Timothy. The NIST Definition of Cloud Computing. NIST Special Publication 800-145. 2011.

39. What is Platform-as-a-Service (PaaS)? Přístup z Internetu: https://www.cloudflare.com/ru-ru/learning/serverless/glossary/platform-as-a-service-paas/ Citováno dne: 29.09.2020

40. Kate C. What Is an MSP (Managed Service Provider)? Datum publikace: 7.05.2020. Přístup z Internetu: https://www.msp360.com/resources/blog/what-is-an-msp/

41. Wikipedie, otevřená encyklopedie. Přístup z Internetu: https://en.wikipedia.org/wiki/Managed_services Citováno dne: 30.09.2020

42. Benefits Of Cloud Computing. Přístup z Internetu: https://www.salesforce.com/products/platform/best-practices/benefits-of-cloud-computing/ Citováno dne: 01.10.2020

43. LARKIN Andrew. Disadvantages of Cloud Computing. Datum publikace: 7.08.2019. Přístup z Internetu: https://cloudacademy.com/blog/disadvantages-of-cloud-computing/

44. WATTS Stephen, RAZA Muhammad. SaaS vs PaaS vs IaaS: What's The Difference & How To Choose. Datum publikace: 15.06.2019. Přístup z Internetu: https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/

45. Microsoft Azure. What is IaaS? Přístup z Internetu: https://azure.microsoft.com/en-us/overview/what-is-iaas/ Citováno dne: 03.10.2020

46. PUTTA Harshitha. Simplified Bring-Your-Own-License experience using AWS License Manager. Datum publikace: 26.03.2020. Přístup z Internetu: https://aws.amazon.com/ru/blogs/mt/simplified-byol-experience-using-aws-license-manager/

47. Microsoft Azure. What is PaaS? Přístup z Internetu: https://azure.microsoft.com/en-us/overview/what-is-paas/ Citováno dne: 04.10.2020

48. Microsoft Azure. What is SaaS? Přístup z Internetu: https://azure.microsoft.com/en-us/overview/what-is-saas/ Citováno dne: 04.10.2020

49. REED Jessie. An Introduction to Cloud Backup: What, Why, and How. Datum publikace: 12.12.2018. Přístup z Internetu: https://www.nakivo.com/blog/introduction-cloud-backup/

50. KANDEYEVA Alisa. 3 apparent advantages to deploy infrastructure in a public cloud. Přístup z Internetu: https://www.sim-networks.com/blog/benefits-of-public-cloud#advantage-1 Citováno dne: 10.10.2020

51. Oficiální dokumentace produktu GitHub. Přístup z Internetu: https://docs.github.com/en Citováno dne: 20.10.2020

52. Oficiální dokumentace produktu Google Cloud Platform. Přístup z Internetu: https://cloud.google.com/docs Citováno dne: 21.10.2020

53. Oficiální dokumentace produktu Kubernetes. Přístup z Internetu: https://kubernetes.io/docs/home/ Citováno dne: 25.10.2020

54. TRENDOWICZ A., JEFFERY R. Software Project Effort Estimation. Cham: Springer International Publishing, 2014, XXII, 469 s. Print ISBN: 978-3-319-03628-1 Online ISBN: 978-3-319-03629-8

55. Nezávislý portál plynu a elektřiny energie123.cz. Přístup z Internetu: https://www.energie123.cz/ Citováno dne: 14.11.2020

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor: **Bc. Alexandra Ivakhnova**

Studium: I1700687

Studijní program: N6209 Systémové inženýrství a informatika

Studijní obor: Informační management

Název diplomové práce: **Testování aplikací v cloudu pro malé a střední projekty**

Název diplomové práce AJ: Software testing in the cloud for small and larger projects

Cíl, metody, literatura, předpoklady:

Cílem diplomové práce je zjistit pro jaké (převážně z hlediska velikosti vývojářského týmu) projekty je výhodnější používat cloudové řešení pro testování aplikací. Osnova diplomové práce: Úvod Podstata a principy softwarového testování Testovaí technologie Specifika testování aplikací při agilním vývoji Cloudové technologie Analýza velikosti projektů vzhledem k nasazení do cloudu Kalkulace Závěr

1. S. Nachiyappan, S. Justus, Cloud Testing Tools and its Challenges: A Comparative Study, Procedia Computer Science, Volume 50, 2015, Pages 482-489, ISSN 1877-0509 2. Akerele O., Ramachandran M., Dixon M. (2013) Testing in the Cloud: Strategies, Risks and Benefits. In: Mahmood Z., Saeed S. (eds) Software Engineering Frameworks for the Cloud Computing Paradigm. Computer Communications and Networks. Springer, London 3. Muhammad Younas, Dayang N.A. Jawawi, Imran Ghani, Terrence Fries, Rafaqut Kazmi, Agile development in the cloud computing environment: A systematic review, Information and Software Technology, Volume 103, 2018, Pages 142-158, ISSN 0950-5849 4. Nalini Subramanian, Andrews Jeyaraj, Recent security challenges in cloud computing, Computers & Electrical Engineering, Volume 71, 2018, Pages 28-42, ISSN 0045-7906 5. Tilley S., Parveen T.(2012) SMART-T: Migrating testing to the Cloud. In: Software Testing in the Cloud. SpringerBriefs in Computer Science. Springer, Berlin, Heidelberg

Garantující pracoviště: Katedra informačních technologií,
Fakulta informatiky a managementu

Vedoucí práce: Ing. Martina Husáková, Ph.D.

Datum zadání závěrečné práce: 15.10.2019