Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Využití nástrojů CI/CD při vývoji software

Diplomová práce

Autor: Bc Viktor Pešek
Studijní obor: Aplikovaná informatika

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.
Odborný konzultant: Ing. Jakub Šlambora, Software Architect, Unicorn Systems

Hradec Králové duben 2022

Prohlášení:

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

vlastnoruční podpis

V Hradci Králové dne 26.4.2022 Jméno a Příjmení

Poděkování:

Děkuji vedoucímu diplomové práce profesoru Antonínu Slabému za metodické vedení práce a Ing. Jakubu Šlamborovi za odborné vedení při zpracovávání diplomové práce. Děkuji mé rodině a přítelkyni za podporu při studiu.

Anotace

Diplomová práce se zabývá představením problematiky Continuous Integration a Continuous Delivery a zefektivněním vývojového cyklu zavedením automatizace různých dílčích úkolů v rámci procesu vývoje a dodávání softwaru. V rámci této diplomové práce je provedena analýza zmíněných procesů při vývoji aplikací OPC a STA v softwarové firmě Unicorn. V analytické části jsou představeny problémy, se kterými se týmy spolupracující na vývoji potýkají. V následující praktické části jsou tyto problémy vyřešeny a jsou představeny úpravy v Continuous Integration a Continuous Delivery.

Annotation

Title: Using CI / CD tools in software development

The diploma thesis deals with the introduction of the issues of Continuous Integration and Continuous Delivery and streamlining the development cycle by introducing the automation of various subtasks within the software development and delivery process. As part of this diploma thesis, an analysis of the mentioned processes in the development of OPC and STA applications in the software company Unicorn is performed. The analytical part presents the problems that the teams cooperating in the development face. In the following practical part, these problems are solved and adjustments in Continuous Integration and Continuous Delivery are introduced.

Obsah

1 Úvod ... 1
2 Cíl práce ... 2
3 Metodika zpracování ... 3
4 Životní cyklus softwaru ... 4
4.1 Metodiky vývoje softwaru ... 4
4.1.1 Vodopádový přístup ... 4
4.1.2 Agilní přístup ... 5
4.2 Fáze vývojového cyklu SDLC ... 8
4.2.1 Analýza požadavků ... 9
4.2.2 Plánování ... 10
4.2.3 Návrh ... 10
4.2.4 Vývoj ... 11
4.2.5 Testování ... 11
4.2.6 Nasazení ... 12
4.3 Continuous integration ... 12
4.3.1 Testování aplikace ... 13
4.4 Continuous delivery ... 17
4.5 Continuous deployment ... 18
4.6 Aplikace řešící Continous Integration a Continous Delivery ... 18
4.6.1 Jenkins ... 18
4.6.2 TeamCity ... 19
4.6.3 CircleCI ... 19
4.6.4 GitLab CI ... 19
4.7 IaC (Infrastructure as Code) ... 20
5 Analýza stávajícího řešení ... 22

5.1 Použité technologie ... 22
5.1.1 Docker ... 22
5.1.2 Jenkins ... 23
5.1.3 Azure Kubernetes Service (AKS) ... 24
5.1.4 Azure Container Registry (ACR) ... 24
5.1.5 Kubernetes ... 24
5.1.6 Helm ... 26
5.1.7 GitLab ... 27
5.2 Architektura systémů ... 27
5.3 Složení týmů ... 29
5.4 Docker obraz ... 29
5.5 Pravidla verzování ... 30
5.6 Unit testy ... 31
5.7 Vývojové prostředí ... 31
5.8 Uložení balíčků k nasazení ... 32
5.9 Continuous Integration a delivery na projektu OPCSTA ... 33
5.9.1 Sestavení aplikace a vytvoření Docker obrazu ... 34
5.9.2 Nasazení na vývojové prostředí ... 35
5.9.3 Nahrání Docker obrazu k zákazníkovi ... 36
5.9.4 Nahrání Helm Chartů k zákazníkovi ... 36
5.10 Nasazení k zákazníkovi ... 37
5.11 Aktuální nedostatky CI/CD ... 40
5.11.1 Notifikace z Jenkins do Slack ... 40
5.11.2 Řazení spouštění aplikací ... 41
5.11.3 Nezabezpečená hesla v Git repozitáři ... 41
5.11.4 Manuální spouštění CI pipeline ... 41

5.11.5 Manuální přesun verze k zákazníkovi 42
5.11.6 Aplikace se po pádu znovu nerestartuje 42
6 Implementace změn v CI/CD 43
6.1 Aktualizace verze Jenkins CI 43
6.1.1 Použití Ansible pro nasazení Jenkins CI 44
6.2 Zabezpečení hesel aplikací 47
6.2.1 SealedSecrets 48
6.3 Jenkins CI pipeline 51
6.4 Integrace Slack notifikací do Jenkins 55
6.5 Pořadí nasazování aplikací 56
6.6 Verzování Helm Chartů 60
6.7 Přesun Docker obrazů k zákazníkovi 61
6.8 Automatický restart aplikace po pádu 62
6.9 Demonstrace změn v CI/CD 63
7 Závěr 66
8 Seznam použité literatury 67
9 Přílohy 71

Seznam obrázků

Obrázek 1 Ganttův diagram, zdroj: [3] 5
Obrázek 2 Scrum proces, zdroj: [4] 6
Obrázek 3 Kanban tabulka, zdroj [7] 7
Obrázek 4 Fáze SDLC, zdroj: [8] 9
Obrázek 5 Proces Continous Integration, zdroj: vlastní zpracování 13
Obrázek 6 Testovací pyramida, zdroj: [19] 15
Obrázek 7 Continuous delivery proces, zdroj: vlastní zpracování 17
Obrázek 8 Continuous deployment proces, zdroj: vlastní zpracování 18
Obrázek 9 Stávající podoba CI/CD pipeline, zdroj: vlastní zpracování 27
Obrázek 10 Komunikace aplikací, zdroj: vlastní zpracování 29
Obrázek 11 Subskripce Azure, zdroj: vlastní zpracování 32
Obrázek 12 Adresářová struktura balíčků Helm, zdroj: vlastní zpracování 33
Obrázek 13 Vydání verze v Jenkins CI, zdroj: vlastní zpracování 35
Obrázek 14 Nasazení aplikace na vývojové prostředí, zdroj: vlastní zpracování 35
Obrázek 15 Vstupní formulář k nahrání Docker obrazu, zdroj: vlastní zpracování 36
Obrázek 16 Vstupní parametry nahrání Helm Chartu, zdroj: vlastní zpracování 37
Obrázek 17 GitLab pipeline, nahrání Helm Chartů 37
Obrázek 18 Gitlab repozitáře, zdroj: vlastní zpracování 38
Obrázek 19 Pipeline pro instalalci u zákazníka, zdroj: vlastní zpracování 39
Obrázek 20 Jenkins Plugin Management, zdroj: vlastní zpracování 44
Obrázek 21 Jenkins Plugins, zdroj: vlastní zpracování 44
Obrázek 22 Dekódování Base64, zdroj: https://www.base64decode.org/ 48
Obrázek 23 Nová struktura git repozitáře, zdroj: vlastní zpracování 52
Obrázek 24 Detail nove CI pipeline, zdroj: vlastní zpracování 53
Obrázek 25 Jenkins Pipeline, zdroj: vlastní zpracování 55
Obrázek 26 Ukázka notifikace z Jenkins do Slack, zdroj: vlastní zpracování 56

Obrázek 27 Ukázka inicializačního kontejneru, zdroj: [43] 57
Obrázek 28 Ganttův diagaram posloupnosti nasazení, zdroj: vlastní zpracování 58
Obrázek 29 Skripty kontrolující dostupnost mikroslužeb, zdroj: vlastní zpracování 58
Obrázek 30 Ukázka události aplikace, zdroj: vlastní zpracování 63
Obrázek 31 Nastavení verze aplikace na 2.5.0, zdroj: vlastní zpracování 63
Obrázek 32 Vydání verze v Jenkins, zdroj: vlastní zpracování 64
Obrázek 33 Docker obraz u zákazníka, zdroj: vlastní zpracování 64
Obrázek 34 Merge request v GitLab s verzí 2.5.0, zdroj: vlastní zpracování 65
Obrázek 35 Slack notifikace, zdroj: vlastní zpracování 65

Seznam kódů

Ukázka kódu 1 Aplikace v env-prod.json s atributy pro instalaci 40
Ukázka kódu 2 Dockerfile pro Jenkins CI, zdroj: vlastní zpracování 45
Ukázka kódu 3 Ansible playbook pro Jenkins CI, zdroj: vlastní zpracování 47
Ukázka kódu 4 Instalace kubeseal, zdroj: vlastní zpracování 49
Ukázka kódu 5 Instalace SealedSecret kontroleru, zdroj: vlastní zpracování 49
Ukázka kódu 6 Soubor se Secret objektem, zdroj: vlastní zpracování 49
Ukázka kódu 7 Příkaz na vytvoření SealedSecret, zdroj: vlastní zpracování 50
Ukázka kódu 8 Soubor se SealedSecret, zdroj: vlastní zpracování 51
Ukázka kódu 9 Ověření dešifrování, zdroj: vlastní zpracování 51
Ukázka kódu 10 Skript pro kontrolu dostupnosti aplikce, zdroj: vlastní zpracování 60
Ukázka kódu 11 Dockerfile pro init kontejnery, zdroj: vlastní zpracování .. 60
Ukázka kódu 12 Logování inicializačního kontejneru, zdroj: vlastní zpracování 60
Ukázka kódu 13 Chart.yaml, zdroj: vlastní 61
Ukázka kódu 14 Konfigurace Liveness Probe, zdroj: vlastní zpracování 62

1 Úvod

Tato diplomová práce se zabývá problematikou postupného dodávání softwarového řešení zákazníkovi. V rámci diplomové práce bude představena problematika Continuous Integration a Continuous Delivery. Tyto dvě metodologie spolu úzce souvisí a přispívají k automatizaci integrace kódu a vydávání nových verzí softwaru a zjednodušení dodávek softwarového produktu. V rámci diplomové práce budou představeny teoretické aspekty CI a CD a bude zanalyzováno současné řešení používané při vývoji aplikací OPC a STA ve společnosti Unicorn. Výstupem analýzy bude seznam nedostatků stávajícího CI a CD procesu. V rámci praktické části proběhne úprava stávajícího procesu vývoje zavedením automatizace opakujících se úkolů. Manuální činností může vznikat lidská chyba, kterou se automatizace snaží eliminovat. Praktickým výstupem této diplomové práce bude optimalizovaný proces CI a CD, který bude řešit nedostatky nalezené v analytické části diplomové práce.

2 Cíl práce

Cílem diplomové práce je poskytnout čtenářům vhled do oblasti Continuous Integration a Continuous Delivery. Popsat, jaké způsoby řešení se ve světě používají. Na základě zpracovaných témat v teoretické části bude v praktické části provedena analýza řešení CI/CD při vývoji informačních systémů OPC a STA ve společnosti Unicorn. Bude představena aktuální situace při vývoji softwaru, jaké jsou procesy vývoje, jak vývoj probíhá a jaké jsou úskalí spojené s implementovaným řešením Continuous Integration a Continuous Delivery. V další kapitole autor navrhne řešení, které se bude snažit eliminovat nalezené problémy z analytické části diplomové práce s použitím znalostí z teoretické části.

3 Metodika zpracování

Výstupem diplomové práce je představení problematiky Continuous Integration a Continuous Delivery. V praktické části bude vytvořena analýza stávajícího fungování CI/CD na projektu OPC/STA vyvíjené softwarovou společností Unicorn. Budou definovány nedostatky a požadavky na vylepšení stávajícího procesu vývoje softwaru. V praktické části budou poté tyto body implementovány a otestovány.

4 Životní cyklus softwaru

Kapitola popisuje životní cyklus softwaru od analýzy a vývoje produktu, přes testování až po uvedení do produkce.

4.1 Metodiky vývoje softwaru

Následující kapitola popisuje jednotlivé metodiky vývoje softwaru a rozdíly mezi nimi.

4.1.1 Vodopádový přístup

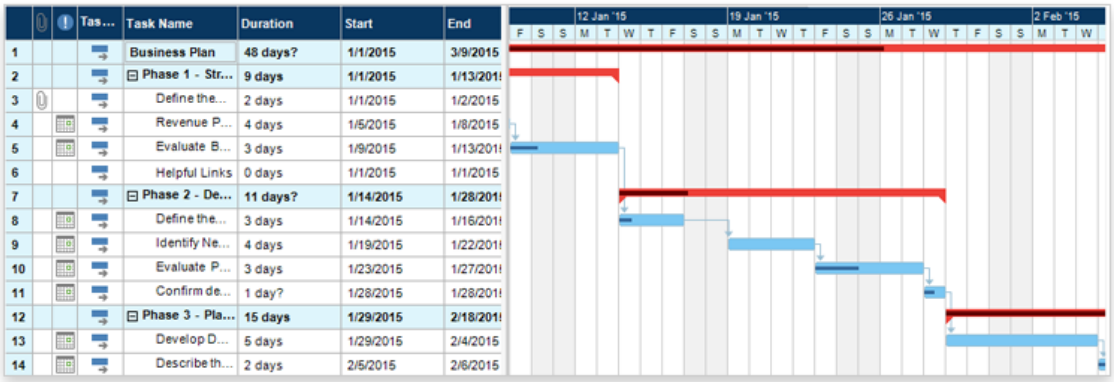
Vodopádový přístup je lineární přístup k vývoji softwaru, kdy se na začátku projektu shromažďují požadavky zúčastněných stran a zákazníků a poté se vytvoří plán postupného projektu, který těmto požadavkům vyhovuje [1]. Model vodopádu je pojmenován takto, protože každá fáze projektu kaskádovitě přechází do další a postupně klesá jako vodopád. Jednotlivé fáze lze definovat takto [2]:

1. Shromáždění a zdokumentování požadavků
2. Návrh
3. Vývoj a jednotkové testování
4. Výkonnostní testování
5. Uživatelské testování (UAT)
6. Oprava vzniklých problémů
7. Dodávka finálního produktu

Jedná se o důkladnou a strukturovanou metodiku, která se používá již dlouho, protože se osvědčila. Mezi průmyslová odvětví, která model vodopádu pravidelně používají, patří stavebnictví, IT a vývoj softwaru. Termín „vodopád“ se však obvykle používá v kontextu softwaru. [1] Ve skutečném vodopádovém vývojovém projektu každá z těchto fází představuje odlišnou fázi vývoje softwaru a každá fáze obvykle končí dříve, než může začít další. Mezi nimi je prodleva, kdy se například musí čekat na schvalování návrhu zákazníkem [2].

Ganttovy diagramy jsou preferovaným nástrojem pro projektové manažery pracující ve vodopádu. Použití Ganttova diagramu umožňuje mapovat dílčí úkoly,

závislosti a každou fázi projektu v průběhu životního cyklu projektu. Ganttův diagram se běžně používá v projektovém managementu. Pro tuto činnost lze využít software ProjectManager.com, který nabízí výše uvedené funkce. Tento software je jedním z nejpopulárnějších a nejužitečnějších nástrojů pro zobrazení aktivit v čase. V levé části grafu je seznam aktivit a v horní části je vhodná časová stupnice. Každá aktivita je reprezentovaná pruhem (pozice a délka pruhu odráží datum zahájení a trvání a datum ukončení aktivity) [3].



Obrázek 1 Ganttův diagram, zdroj: [3]

4.1.2 Agilní přístup

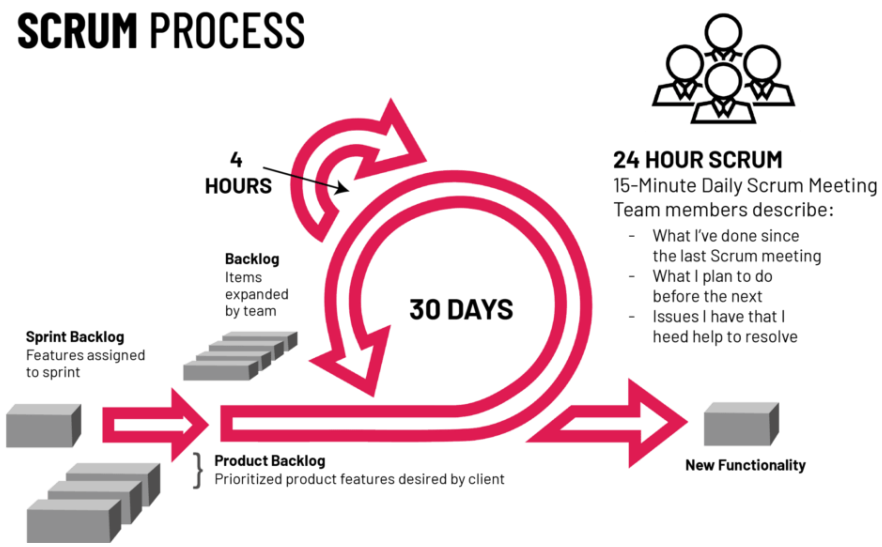
Agilní vývoj softwaru odkazuje na metodiky vývoje softwaru soustředěné kolem myšlenky iterativního vývoje, kde se požadavky a řešení vyvíjely ve spolupráci mezi menšími samostatnými týmy. Agilní metody nebo agilní procesy obecně podporují disciplinovaný proces projektového řízení, který podporuje týmovou práci, sebeorganizaci a odpovědnost. Jsou to soubory postupů, které mají umožnit rychlé dodání vysoce kvalitního software. Zahrnují také obchodní přístup, který sladí vývoj softwaru s potřebami zákazníků a cíli společnosti. Hlavní hodnotou v agilním vývoji je to, že umožňuje týmům dodávat hodnoty rychleji, s vyšší kvalitou, předvídatelností a lepší schopností reagovat na změny. Nejpoužívanější metodiky jsou Scrum a Kanban [4].

4.1.2.1 Scrum

Scrum je podmnožinou agilních metodik. Od ostatních agilních metodik se liší specifickými koncepty a postupy. [4] Scrum se adaptabilní, rychlý, flexibilní a efektivní. Je navržen tak, aby zákazníkovi přinášel hodnotu po celou dobu vývoje

projektu. Primárním cílem je uspokojit potřeby zákazníka prostřednictvím transparentnosti komunikace, kolektivní odpovědnosti a neustálého pokroku. Vývoj pochází z obecné představy o tom, co je třeba vybudovat vypracováním seznamu charakteristik, seřazených podle priority, které chce vlastník produktu získat. [5] Scrum výrazně zvyšuje produktivitu a zkracuje čas potřebný na k porovnání s klasickými „vodopádovými" procesy. [4] Role ve Scrumu jsou [4]:

- **Scrum Master** – Udržuje proces, je zodpovědný za hladký průběh vývoje, odstraňuje překážky, které mají dopad na produktivitu a organizuje setkání týmu.
- **Product Owner** – Udržuje požadavky, je jediným zdrojem pravdy, je rozhraním mezi zákazníkem a týmem. Úzce spolupracuje s týmem na vytváření technických požadavků.
- **Tým** – Je sebeorganizující skupina lidí, kteří vyvíjejí, testují produkt, je zodpovědný za dodávání produktu, členové týmu rozhodují, jak si rozdělí práci, velikost týmu je zpravidla mezi 5 a 9 členy.



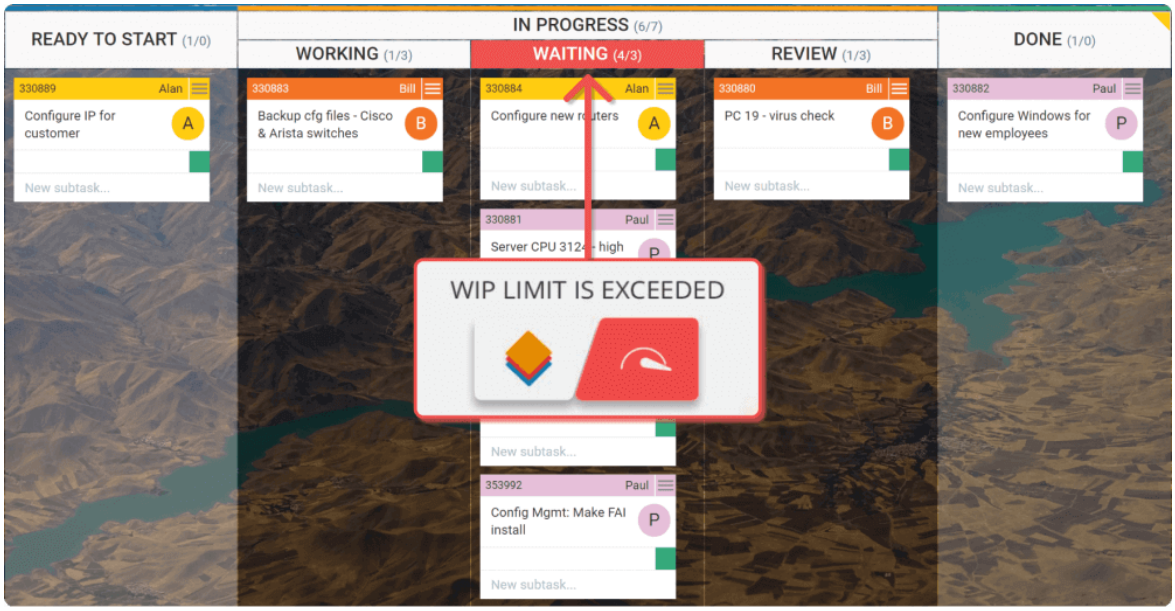
Obrázek 2 Scrum proces, zdroj: [4]

4.1.2.2 Kanban

Kanban je populární agilní metodika vývoje softwaru a řízení lidských zdrojů, pomocí vizualizace úkolů v grafické podobě. [6] Jeho cílem je pomoci vizualizovat práci, maximalizovat efektivitu. Pochází z oblasti výroby, ale začal se uplatňovat

i v agilním softwarovém vývoji. Slovo „kanban“ pochází z japonštiny a znamená „vizuální tabule“. Používala ho automobilka Toyota již v 60. letech 20 století. [7] Kanban se zaměřuje na rozvoj orientovaný na služby (service oriented approach). Vyžaduje hluboké porozumění potřebám zákazníků. Při využívání Kanban metodiky je důležité soustředit se na 3 hlavní principy. [7]

- **Zaměření se na potřeby zákazníka** – středobodem každé organizace by mělo být poskytování hodnoty zákazníkovi. Pochopení potřeb a očekávání zákazníků přivádí pozornost ke kvalitě poskytovaných služeb.
- **Řízení práce –** správa práce zajišťuje, že je posílena schopnost lidí samostatně se organizovat. To umožní soustředit se na požadované výsledky bez mikro-řízení lidí.
- **Pravidelná kontrola** – Jakmile jsou dané služby vyvinuty, vyžadují neustálé hodnocení. Prostřednictvím pravidelných kontrol služeb a hodnocení podporuje Kanban zlepšování poskytovaných výsledků.



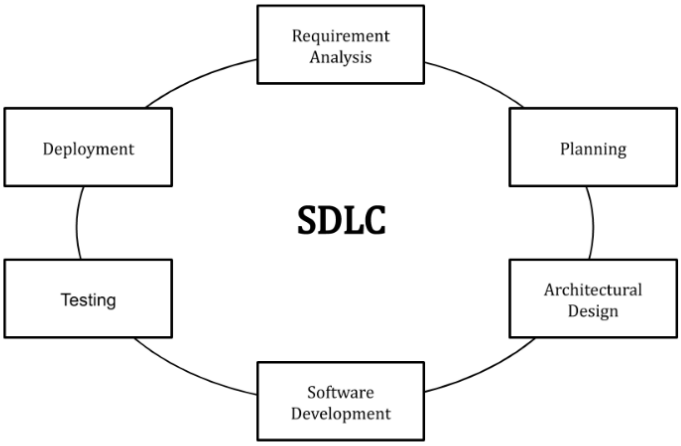
Obrázek 3 Kanban tabulka, zdroj [7]

4.2 Fáze vývojového cyklu SDLC

V minulých kapitolách byly představeny metodiky vývoje softwaru. Jak bylo vysvětleno, existují směry vodopádového a agilního přístupu k vývoji softwaru. Všechny popsané metodiky zahrnují fáze, které budou představeny v nadcházející kapitole. Zkratka SDLC znamená Software Development LifeCycle a odkazuje na metodiku s jasně definovanými procesy pro vytváření vysoce kvalitního softwaru. Podrobně se metodologie SDLC zaměřuje na tyto fáze vývoje softwaru: [8]

- Analýza požadavků
- Plánování
- Softwarový design
- Vývoj
- Testování
- Nasazení

Životní cyklus vývoje softwaru (SDLC) je proces, který produkuje software vysoké kvality a nízké ceny v co nejkratším čase. Poskytuje strukturovaný tok fází, které pomáhají rychle produkovat vysoce kvalitní software, který je dobře otestovaný a připravený k produkčnímu použití. Níže jsou popsány jednotlivé fáze vývoje softwaru. U každé fáze bude nastíněno, jakým způsobem je řešena u popsaných metodik.



Obrázek 4 Fáze SDLC, zdroj: [8]

4.2.1 Analýza požadavků

Během první fáze jsou od zákazníka shromážděny všechny relevantní informace potřebné k vyvinutí produktu. Zákazník spolu s analytiky sestaví požadavky na funkčnosti aplikace, které spolu konzultují. V případě nejasností je svolána schůzka k dalšímu projednání. Dalším krokem je určení náročnosti produktu na hardwarové a síťové zdroje[1]. Jakmile jsou všechny požadavky shromážděny je provedena technická analýza kvůli ověření proveditelnosti vývoje produktu. Do procesu analýzy se mohou v dalších iteracích projektu zapojit i lidé jako vývojáři, testeři, kteří mohou při vývoji starší verze aplikace nalézt aplikační chybu, kterou v další verzi produktu opraví. [8–10]

Při použití vodopádového přístupu je na tuto fázi kladen důraz, aby byly všechny požadavky velmi dobře zdokumentované a jasně definované. Jeho použití je vhodné pro menší projekty, kde jsou dobře pochopeny požadavky na funkčnost. [11] Jak již bylo řečeno, agilní metodiky jsou iterativní. Stejně tomu je i u analýzy požadavků. Tato fáze je prováděna v každé iteraci. [9]

[1] V dnešní době nedostatku křemíku pro výrobu procesorů je toto důležité řešit s předstihem, pozn. autora.

4.2.2 Plánování

V této fázi tým stanovuje náklady a zdroje potřebné k implementaci analyzovaných požadavků. Vytvořením efektivního plánu pro nadcházející vývojový cyklus teoreticky zachytí problémy dříve, než ovlivní vývoj. Snad nejdůležitější je, že fáze plánování stanoví harmonogram projektu, který může mít klíčový význam, pokud se jedná o komerční produkt, který musí být odeslán na trh do určité doby. Této fázi se také říká fáze proveditelnosti. [8–10]

Hlavním rozdílem agilních metodik oproti vodopádovému přístupu je v rozdělení hlavního vývojového procesu na více cyklů (sprintů). Každá iterace projde všemi fázemi vývoje a na jejím konci dojde ke zhodnocení výsledků. Provedením zhodnocení se může tým poučit z chyb v minulém sprintu a v příštím se jich vyvarovat. [12]

4.2.3 Návrh

Návrh vzniká z požadavků definovaných v první fázi SDLC. Požadavky se přemění do více logické struktury, která může být později implementována v programovacím jazyce, který se používá v dané architektuře. Vzniká návrhový plán, tzv. Design Specification. Všechny zúčastněné strany pak tento plán přezkoumají a nabídnou zpětnou vazbu a návrhy na změny/zlepšení. Je důležité mít plán pro shromažďování a začlenění vstupů zúčastněných stran do tohoto dokumentu. Selhání v této fázi téměř jistě povede v nejlepším případě k překročení nákladů a v nejhorším případě k úplnému kolapsu projektu. Po dokončení této fáze manažer připraví finální verzi tohoto dokumentu a ta je použita v dalších fázích SDLC. [8–10]

Ve vodopádové metodice je návrh vytvářen najednou a díky nedostatku zpětné vazby od zákazníka během této a vývojové fáze je docela časté, že týmy vytvoří nepotřebné nebo málo používané funkcionality, které vedou ke ztrátě času, úsilí i peněz. Naopak agilní metodika je založena, jak bylo řečeno dříve, na přírůstkovém (iterativním) dodávání softwaru. Na začátku je tedy vytvořen prvotní návrh a ten je v dalších fázích přizpůsobován potřebám zákazníka. Dokud není systém připraven k použití jsou všechny fáze SDLC opakovány. Je časté, že týmy pracují na více fázích najednou. [13]

4.2.4 Vývoj

Vývojová část je část, ve které vývojáři vytvářejí kód a sestavují aplikaci podle dřívějších návrhů a specifikací. Vývojáři následují pokyny pro implementaci tak, jak je definuje organizace, a používají různé nástroje jako jsou kompilátory, debuggery a interprety [10]. Tyto nástroje jsou obvykle integrované v IDE (Intergrated Development Environment), programu poskytujícímu základní nástroje pro tvorbu kódu. Nejznámějšími jsou například: IntelliJ (IDEA, WebStorm, PHPStorm, PyChart, …), NetBeans, Visual Studio Code, Eclipse, Visual Studio. Dalšími nástroji můžou být například SonarLint a ESLint. Tyto nástroje analyzují kód a upozorňují vývojáře na nebezpečné konstrukce či špatná jména proměnných.

V minulé kapitole bylo u agilního vývoje zmíněno, že probíhá rychleji a dodávky softwaru jsou častější, nežli tomu je u vodopádového přístupu. V tomto se tyto dva přístupy od sebe liší. Ve vodopádovém přístupu obvykle trvá déle, než je dostupná první verze softwaru. Agilní metodika se naopak snaží vyvinout první demo verzi co nejrychleji, aby bylo možné pružně reagovat na zpětnou vazbu od zákazníka. [2]

4.2.5 Testování

Jakmile je k dispozici sestavená aplikace, je uvolněna k otestování. V této fázi je vyvinutý software otestován a případné nalezené chyby jsou nahlášeny vývojářům, aby je opravili. Na konci této fáze je produkt, který odpovídá požadavkům a očekáváním zákazníka. Na testování se podílejí testeři firmy, která produkt vyvíjí. V další fázi je produkt otestovaný i uživateli, kteří jej v budoucnu budou používat. [8–10]

U testování platí podobné rozdíly jako tomu bylo u předcházející fáze. Testování v agilní metodice je obvykle pružnější oproti vodopádovému přístupu. Vodopádový přístup vyžaduje důkladnou dokumentaci testovací fáze, naproti tomu agilní přístup toto nevyžaduje. Zde je tedy velký prostor pro úsporu času, ale za cenu menší a méně důkladné dokumentace. Testování v agilní metodice probíhá částečně už ve fázi vývoje. U vodopádového přístupu toto nastává až na konci projektu. V případě nejasnosti požadavků od zákazníka je agilní metodika více efektivní

oproti vodopádové. Obecně u větších systémů/projektů se doporučuje spíše agilní metodika testování. [14]

4.2.6 Nasazení

Když je produkt otestován přichází fáze nasazení na produkční nebo testovací zákaznické prostředí, podle nastavení očekávání se zákazníkem. V případě testovacího prostředí je vytvořena replika produkčního prostředí a zákazník spolu s vývojáři provádí testování. Pokud aplikace podle zákazníka odpovídá očekávání, je testování ukončeno a aplikace nasazena na produkční prostředí. [8–10]

Jak bylo řečeno u přechozí kapitoly o testování, nasazení je opět prováděno u vodopádového přístupu po dokončení fází předešlých. U agilních metodik je proces nasazení frekventovanější, jelikož je cílem dodávat postupně menší kvantitu změn. [13]

4.3 Continuous integration

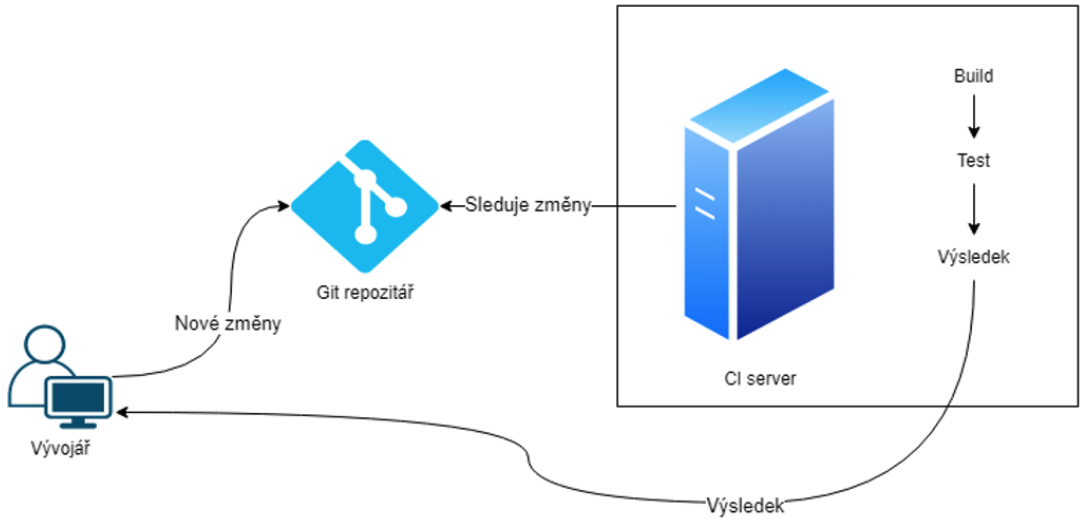
Dodání softwarové aplikace nebo systému dnes zahrnuje několik týmů vývojářů pracujících na samostatných součástech aplikace. Výsledná aplikace musí také komunikovat s okolními aplikacemi nebo službami třetích stran. Výsledkem je, že vývojáři obecně potřebují integrovat svou práci s komponentami vytvořenými jinými vývojovými týmy a s dalšími aplikacemi a službami. Díky tomu je integrace nezbytným a složitým úkolem v životním cyklu vývoje softwaru. [15] V minulosti vývojáři pracovali v týmu po delší dobu izolovaně a poté bylo náročné začlenit jejich změny se změnami od ostatních vývojářů. To vedlo k tomu, že se hromadily chyby a dlouhou dobu čekaly na opravení. Tyto faktory ztížily rychlé doručování nových verzí zákazníkům. Se zavedením průběžné integrace mohou vývojáři častěji přispívat svými změnami do sdíleného repozitáře[2] použitím verzovacího systému. [16]

[2] Repozitář je označením pro složku, kam se ukládají soubory k vytvořeným kódem, je uložen na serveru a vývojáři se k němu připojují.

Proces Continuous integration (dále CI) se obvykle spouští po každém commitu do repozitáře zdrojových kódů. Obvykle je složen z kroků popsaných v seznamu níže. Podle potřeb daného týmu/aplikace může být rozšířen o další kroky.

1. Sestavení aplikace (build)
2. Otestování aplikace automatickými/jednotkovými/integračními testy (test)
3. Statická analýza kódu

Díky tomuto řešení vývojář po krátké době od publikování změny vidí, zda jeho změna nerozbila část systému, na které zrovna pracoval, a dokáže rychleji odhalit a opravit případné chyby. [17] V dalších kapitolách budou představeny jednotlivé nástroje, pomocí kterých je proces CI implementován.



Obrázek 5 Proces Continous Integration, zdroj: vlastní zpracování

4.3.1 Testování aplikace

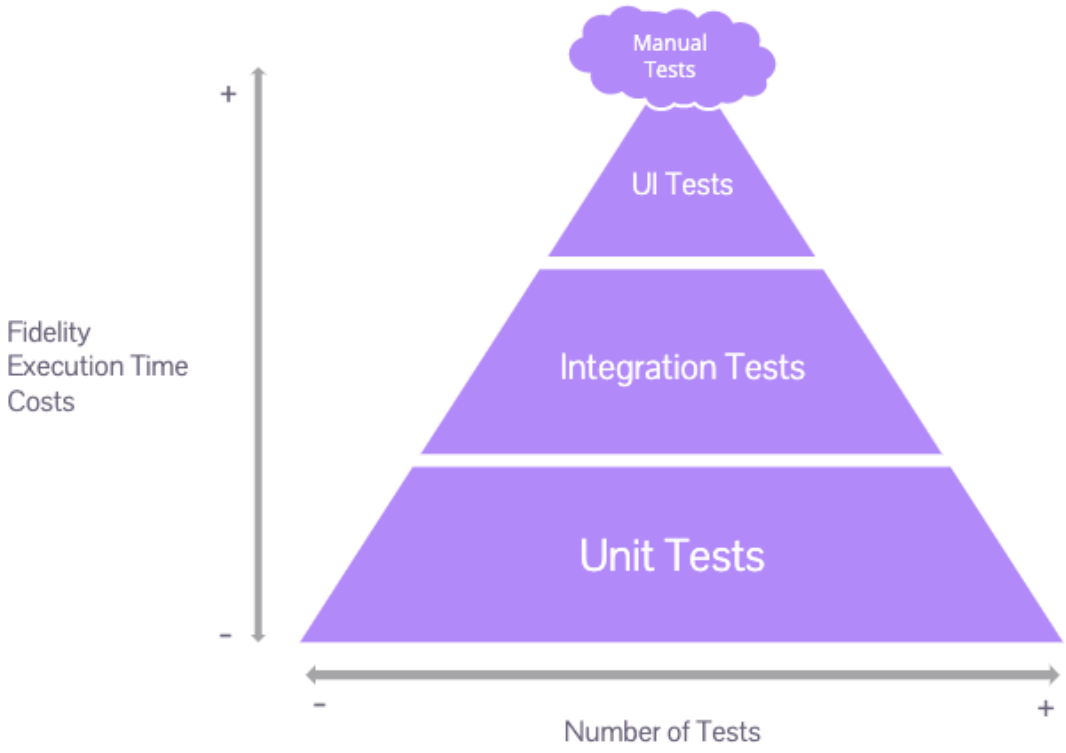
Než se software dostane do produkčního provozu, vyžaduje testování. Jak se vyvíjel vývoj softwaru, tak dozrávaly i přístupy k testování softwaru. Místo toho, aby tým měl veliké množství testerů, vývojové týmy přešly k automatizaci největší části svého testovacího úsilí. [18] Není nutné čekat až bude aplikace nasazena v testovacím prostředí a někdo zodpovědný za klikání na spoustu tlačítek nezjistí,

že aplikace nefunguje podle očekávání. Dnes existují procesy, které umožňují spouštět automatizované testy předtím, než se aplikace dostane do produkce. [19]

Tradičně bylo testování softwaru velké množství manuální práce, která byla prováděna nasazením aplikace do testovacího prostředí a následným klikáním do uživatelského rozhraní, kde bylo zjišťováno, zda je něco nefunkční. Tyto testy byly často realizovány testovacími skripty, aby bylo zajištěno, že testeři budou provádět konzistentní kontrolu. [20]

Testování softwaru při CI procesu je velmi důležité, jelikož testy nejsou prováděny člověkem, ale strojem. Vývojář tedy upraví a uloží kód a v rámci CI procesu se mu dostává rychlé zpětné vazby na jeho změnu. V případě, že změna prošla všemi testy, je poté připravena na sloučení do hlavní větve a nasazení na testovací/produkční prostředí. V případě, že z nějakého důvodu kód neprošel testovací fází CI procesu, je vývojář notifikován o tomto stavu a může opravit své chyby. [21]

Automatizované testy můžeme rozdělit do několika skupin podle tzv. testovací pyramidy. Tato pyramida je návodem, jaké testy psát a jak je psát. [19]



Obrázek 6 Testovací pyramida, zdroj: [19]

4.3.1.1 Unit testy

Základem testovací sady budou jednotkové (unit) testy. Tyto testy zajišťují, že testovaná jednotka (testovaný předmět) funguje, jak má. [20] Význam pojmu „jednotka“ záleží na kontextu daného programovacího jazyka. Pokud je používán funkcionální programovací jazyk, tak bude jednotkou daná funkce. Test zavolá funkci s rozdílnými parametry a bude zjišťovat, zda vrací očekávané výsledky. V objektově orientovaném jazyce může jako jednotka vystupovat metoda nebo celá třída. [20] Jednotkové testy se mohou psát pro kteroukoliv část softwaru a je ideální, aby každá třída softwaru byla pokryta jednotkovými testy. [19]

4.3.1.2 Integrační testy

Všechny netriviální aplikace se budou integrovat s některými dalšími částmi (databáze, souborové systémy, síťová volání do jiných aplikací, ...). Při psaní jednotkových testů jsou to obvykle části, které jsou vynechávány, aby byla zajištěna

lepší izolace a rychlejší běh testů. Rozdíl mezi integračními a jednotkovými testy je v tom, že integrační testy jsou použity v případě, že testovací scénář pokrývá více než jednu jednotku/funkci/třídu. Integrační testy testují jednotlivé komponenty a to, jak tyto komponenty mezi sebou fungují. Závislosti, které nemusejí být testovány, budou nahrazeny zástupnými objekty tzv. „mocky“. Mock znamená vytvoření falešné externí nebo interní služby, která může zastoupit tu skutečnou a pomůže testům běžet rychleji a spolehlivěji. [19, 20, 22]

4.3.1.3 API testy

Tato skupina se též v angličtině nazývá Contract tests. Tyto testy se používají hlavně ve světě mikroslužeb. V moderních vývojových organizacích je toto cesta, jak škálovat vývojové síly rozdělením vývoje systému mezi více týmů, které vyvíjí samostatné volně provázané mikroslužby. Rozdělení sytému na více malých služeb (mikroslužeb) znamená, že vytvořené služby spolu musejí komunikovat pomocí definovaných rozhraní. API testy ověřuji, že nedojde k porušení kontraktu/smlouvy mezi jednotlivými REST API mikroslužeb. Je potřeba mít jistotu, že ve změnách, které vývojář provedl, nebyly změny i ve struktuře dat, které aplikace poskytuje nebo přijímá. Rozdílem oproti integračním testům je zaměření na výměnu dat mezi klienty a poskytovateli. Pokud nějaký test neskončí úspěšně, víme že došlo ke změně datové struktury, kterou poskytovatel vrací a klient přijímá. [19, 20]

4.3.1.4 Testy uživatelského rozhraní

Většina aplikací disponuje uživatelským rozhraním. Typicky se jedná o webové rozhraní v kontextu webových aplikací. Testy uživatelského rozhraní ověřují, že UI aplikace pracuje správně. Vstup od uživatele by měl spouštět správnou akci, data by měla být prezentována uživateli a stav uživatelského rozhraní by se měl podle očekávání změnit. Testování aplikace od jednoho konce k druhému znamená pouštění testů oproti uživatelskému rozhraní. [19, 20]

4.3.1.5 End-to-End testy

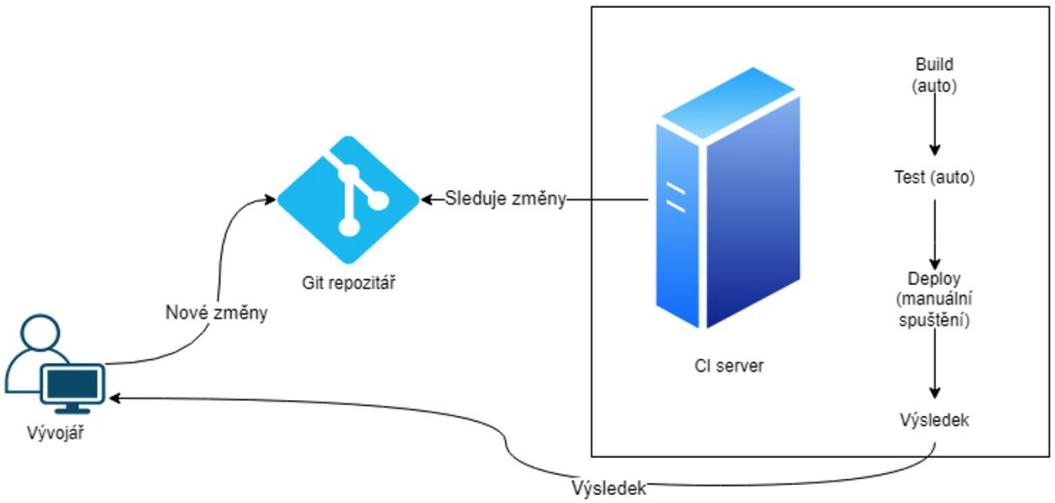
Testování nasazené aplikace pomocí jejího uživatelského rozhraní je také jedním ze způsobů, jak otestovat aplikaci. Tyto testy dávají týmům největší jistotu, že aplikace funguje, jak má. Na rozdíl od testů uživatelského rozhraní berou end-to-

end testy v potaz i kontext uživatele a jeho cestu systémem. Proto je třeba, aby byly připojeny k systémové logice aplikace. End-to-end testy přicházejí s vlastními problémy. Často padají anebo končí v neúspěšném stavu z neočekávaných a nepředvídatelných důvodů. Čím je uživatelské rozhraní propracovanější, tím mají větší tendenci být nekvalitní. Tyto testy vyžadují hodně času stráveného úpravami kvůli problémům s časováním animací, neočekávaným vyskakujícím dialogům atp. [19, 20]

4.4 Continuous delivery

Continuous delivery (dále CD) je rozšíření v předchozí kapitole představeného CI. Je to postup vývoje softwaru, kdy se všechny změny kódu automaticky připravují na vydání. Moderní vývoj softwaru je rozšířen o kontinuální dodávání, což znamená automatické nasazení vydané verze aplikace do testovacího a produkčního prostředí po fázi sestavení. Při správném použití budou mít vždy vývojáři sestavený a otestovaný artefakt připravený k nasazení. Sestavení a otestování je realizováno v CI procesu. [23]

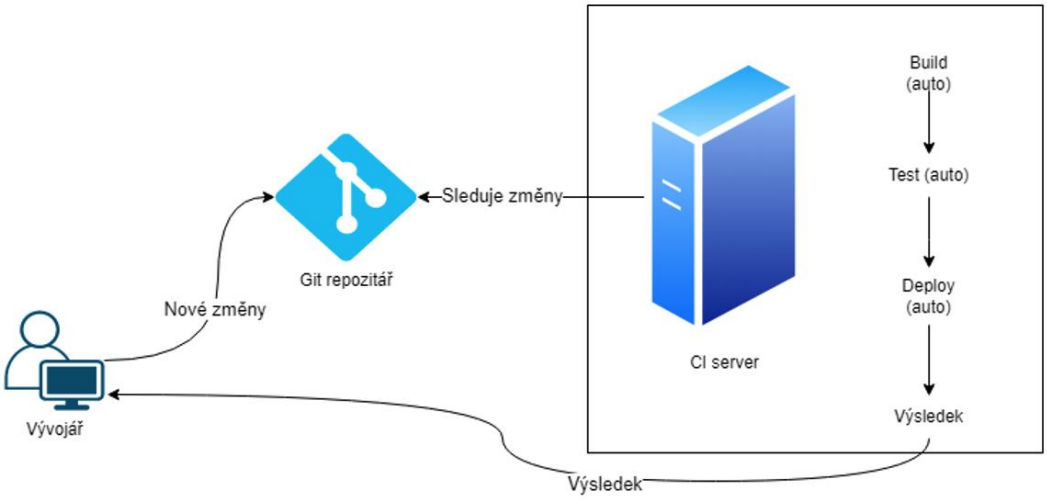
Proces nasazení aplikace může být stále manuálně spouštěn, ale po spuštění by už neměl vyžadovat žádnou lidskou interakci, a měl by nasadit aplikaci do požadovaného prostředí. [24] Na obrázku níže je zobrazen průběh CD procesu. Celý proces je navázán na CI.



Obrázek 7 Continuous delivery proces, zdroj: vlastní zpracování

4.5 Continuous deployment

Continuous deployment jde ještě o krok v před než předchozí CD. V tomto přístupu je každá změna projde všemi fázemi CI a CD automaticky nasazena do produkčního prostředí. Neexistuje žádný lidská zásah a pouze test v neúspěšném stavu může zabránit tomu, aby byla aplikace nasazena na produkční prostředí. [24]



Obrázek 8 Continuous deployment proces, zdroj: vlastní zpracování

4.6 Aplikace řešící Continous Integration a Continous Delivery

Procesy CI a CD jsou základem moderního vývoje softwaru. I když CI i CD technicky nevyžaduje specifické nástroje, většina týmů používá server pro nepřetržitou integraci, který jim pomáhá zefektivnit procesy. Tento server v podstatě spravuje sdílené úložiště kódu a sestavuje nově příchozí kód od vývojářů. Po nalezení nového kódu zahájí sestavení CI a vývojáři pošle výsledky sestavení. Níže jsou uvedeny některé příklady CI/CD serverů. Většina nástrojů nabízí podobné funkcionality. Rozhodnutí, jaký nástroj použít, záleží na preferencích týmů. [25]

4.6.1 Jenkins

Jenkins je automatizační open-source server, který lze použít k automatizaci nejrůznějších úkolů souvisejících s vytvářením, testováním a dodáváním nebo nasazováním softwaru. Jenkins lze nainstalovat prostřednictvím nativních systémových balíčků, Dockeru nebo dokonce spustit samostatně na jakémkoliv

počítači s nainstalovaným Java Runtime Environment (JRE).[3] Jenkins nenabízí žádnou vlastní formu provozování v cloudu[4]. [26]

4.6.2 TeamCity

TeamCity je CI/CD server vyvíjený společností JetBrains. Produkt je dostupný v bezplatné i placené verzi. Bezplatná verze je omezena počtem 100 konfigurací pro sestavení a maximálně třemi agenty, na kterých se spouštějí jednotlivé sestavení. Lze tedy mít spuštěné maximálně 3 úlohy najednou. Provozovat lze TeamCity v cloudu u JetBrains i na vlastních strojích. [27]

4.6.3 CircleCI

CircleCI je automatizační nástroj pro tvorbu CI/CD procesů. Každý den je v CircleCI spuštěn celosvětově necelý milion sestavení a denně podporuje přibližně 30 000 organizací. Nástroj je stejně jako TeamCity dostupný v bezplatné a placené verzi. Nabízí pouze provozování v cloudu u CircleCI. Je nabízena i možnost vlastního provozu, ale ta na stránkách produktu není standartně nabízena. Bezplatná verze je omezena 6 000 minutami běhu sestavení měsíčně a maximálně 30 paralelními spuštění sestavení. [28]

4.6.4 GitLab CI

GitLab je převážně známý jako webové úložiště repozitářů pro verzovací sytém Git. GitLab byl rozšířen o podporu procesů CI a CD. Pro využití CI a CD procesů je nutné mít své aplikace uloženy v GitLab repozitářích. Celý proces CI a CD se odehrává v prostředí webového rozhraní GitLab na stránce www.gitlab.com. Lze sjednat i provozování na vlastních hardwarových prostředcích. Obě varianty jsou dostupné v placených i bezplatných verzích. Bezplatná verze je limitována 400 minutami běhu CI a CD procesů a maximální velikostí repozitářů 5 GB. [29]

---

[3] JRE je běhové prostředí nutné pro spuštění aplikací napsaných v jazyce Java.

[4] Cloud provoz znamená instalaci aplikace třetí stranou a následnou správu této aplikace třetí stranou. Odpadá tedy nutnost mít vlastní tým řešící infrastrukturu, provoz a nasazování aplikace.

4.7 IaC (Infrastructure as Code)

Historicky byla správa IT infrastruktury manuální proces. Správci by fyzicky umístili servery a nakonfigurovali je. Teprve poté, co byly stroje nakonfigurovány na správné nastavení požadované operačním systémem a aplikacemi, by tito lidé aplikace nasadili. [30]

Infrastructure as code je proces správy infrastruktury prostřednictvím kódu namísto manuálních procesů. Pomocí IaC se vytvářejí konfigurační soubory, které obsahují specifikace infrastruktury, což usnadňuje úpravu a distribuci konfigurací mezi prostředími. [31]

Tyto soubory jsou strojově i lidsky čitelné a často uchovávané ve verzovacím systému. [30] Bez tohoto procesu by týmy musely inicializovat každé prostředí pro aplikace ručně. Z prostředí by se staly tzv. „snowflakes“ (vločky). Tento pojem znamená, že na daném prostředí je nahrána unikátní nereprodukovatelná konfigurace. Při ztrátě této konfigurace jsou často ztraceny hodiny manuální konfigurační práce. [32]

Při používání IaC existují dva způsoby, jak uchovávat konfigurace. Existuje tzv. deklarativní a imperativní proces Infrastructure as Code.

4.7.1.1 Deklarativní

Deklarativní přístup definuje požadovaný stav systému, včetně toho, jaké zdroje potřebujeme a jaké vlastnosti by měly mít. Nástroj IaC je pak nakonfiguruje za nás. Deklarativní přístup také vede seznam aktuálního stavu našich systémových objektů, což usnadňuje správu infrastruktury. [31]

4.7.1.2 Imperativní

Imperativní přístup definuje konkrétní příkazy potřebné k dosažení požadované konfigurace a tyto příkazy je třeba provést ve správném pořadí. [31]

Mnoho nástrojů IaC používá deklarativní přístup a automaticky zajistí požadovanou infrastrukturu. Nástroje IaC jsou často schopny fungovat v obou přístupech, ale mají tendenci preferovat jeden přístup před druhým. Preferovanější přístup je deklarativní. Nevyžaduje totiž specifikování procesu instalace a konfigurace. Tato abstraktnost poskytuje větší flexibilitu, jako jsou optimalizované

techniky, které může poskytovatel infrastruktury použít. Pomáhá také snižovat čas potřebný k údržbě imperativního kódu, jako jsou různé skripty pro nasazení, které mohou časem nabývat na velikosti. [31, 32]

Hlavním přínosem, který IaC přináší je rychlost. Infrastructure as code umožňuje rychle nastavit kompletní infrastrukturu pomocí spuštění skriptu. Toto lze udělat pro každé prostředí od vývojového až po produkční. Celý životní cyklus vývoje softwaru se tím zjednoduší. Dalším bodem, který IaC přináší, je konzistentnost konfigurací. Lidé jsou omylní a manuální procesy vedou k chybám. IaC všechny konfigurační soubory udržuje ve verzovacím systému, který je jediným zdrojem pravdy. Toto garantuje, že stejné konfigurace budou použity pro každé nasazení a je možnost garantovat, že každé prostředí (vývojové, testovací, produkční) bude obsahovat stejné konfigurace pro běžící aplikace. Tímto se minimalizují chyby způsobené odlišností jednotlivých běhových prostředí. Díky použití verzovacího systému je aplikován i třetí přínos a to auditovatelnost. Díky změnám v repozitáři lze vidět, kdo a kdy co změnil. [30]

5 Analýza stávajícího řešení

Diplomová práce se zaměřuje optimalizaci procesů CI a CD při vývoji dvou informačních systémů dodávaných společností Unicorn Systems. Konkrétně se jedná o systémy OPC (Outage Plannig Coordination) a STA (Short Term Adequacy). Informační systém OPC slouží ke shromažďování plánovaných odstávek a seznamy prvků sítě všech zúčastněních provozovatelů přenosových soustav v Evropě. Informační systém STA zprostředkovává deterministickou a pravděpodobnostní kalkulaci krátkodobých diagnóz vyvážení energetické sítě. Výsledky této diagnózy mohou obsahovat doporučení k nápravným akcím pro optimalizaci přeshraniční výměny energie.

5.1 Použité technologie

Pro lepší představu budou v této kapitole uvedeny a představeny technologie použité ve stávajícím řešení CI/CD při vývoji informačních systémů OPC a STA.

5.1.1 Docker

Docker poskytuje možnost zabalit a spustit aplikaci v izolovaném prostředí nazvaném kontejner. Kontejnery obsahují vše potřebné ke spuštění aplikace a není nutné nic instalovat na hostitelském systému. Poskytuje jistotu integrity aplikace při spuštění na různých systémech. V Dockeru jsou důležité tyto termíny: [33]

5.1.1.1 Docker image (obraz)

Docker obraz je šablona s pokyny pro vytvoření Docker kontejneru. Je často založen na základě jiného Docker obrazu. Uživatel může vytvářet své obrazy na základě jiných nebo si vytvářet své nové obrazy. K vytvoření vlastního obrazu je zapotřebí vytvořit soubor s názvem ***Dockerfile*** ve kterém jsou definovány kroky pro vytvoření obrazu a jeho následné spuštění. Každý krok obrazu vytvoří jednu vrstvu. Při úpravě Docker obrazu se poté ukládají jen ty vrstvy, které byly změněny. Díky tomu je dosaženo malých velikostí a lehkosti aplikací. [33]

5.1.1.2 Docker container (kontejner)

Kontejner je spustitelná instance Docker obrazu. Pomocí rozhraní Docker lze spustit, vypnout a smazat Docker kontejner. Kontejner je definován svým obrazem a také všemi možnostmi konfigurace, které mu lze poskytnout při vytvoření nebo spuštění. Po odstranění kontejneru zmizí všechny změny jeho stavu, které nejsou uloženy v trvalém úložišti. Všechny data by tak měla být ukládána mimo kontejner. [33]

5.1.1.3 Docker registry (registr)

Registr uchovává Docker obrazy. Docker Hub je veřejný registr, který může používat kdokoliv a Docker je ve výchozím nastavení nakonfigurován tak, aby vyhledával Docker obrazy právě na Docker Hub. Lze provozovat nebo si objednat vlastní registr pro Docker obrazy. [33]

5.1.2 Jenkins

Pravděpodobně nejznámější software pro kontinuální integraci. Open source nástroj používaný k automatizaci procesů, ke kterým dochází při vývoji softwaru. [17] Na projektu se tento nástroj používá pro běh CI procesů v rámci vývoje aplikací OPC a STA.

5.1.2.1 Jenkins Pipeline

Jenkins Pipeline je sada pluginů, která podporuje implementaci CI a CD v Jenkins. Definice této Pipeline je zapsána v textovém souboru nazývaném Jenkinsfile, který lze uchovávat v git repozitáři společně s aplikačním kódem. [26]

5.1.2.2 Jenkinsfile

Jak bylo představeno dříve, Jenkinsfile je textový soubor obsahující definici Jenkins Pipeline. Tento soubor je obvykle uložen v kořenovém adresáři git repozitáře a je psán v programovacím jazyce Groovy. V rámci tohoto souboru jsou definovány některé klíčové bloky [26]

- Pipeline – hlavní blok obalující všechny ostatní bloky [26]

- Stage – v tomto bloku je umístěna série akcí, která má za úkol realizovat část pipeline [26]
- Step – blok uvnitř každé stage direktivy, obsahuje vykonání příkazu [26]
- Environment – zde se specifikují proměnné ve tvaru klíč – hodnota, které jsou dostupné v celé pipeline [26]

5.1.3 Azure Kubernetes Service (AKS)

V rámci Azure je dostupná služba AKS, která nabízí Kubernetes cluster pro provoz aplikací ve vývojových prostředí. [34]

5.1.4 Azure Container Registry (ACR)

Služba od Azure nabízející Docker Registry pro uložení Docker obrazů a jejich následné použití v rámci služby AKS. [35]

5.1.5 Kubernetes

Orchestrační služba typu open-source pro běh Docker kontejnerů. Usnadňuje deklarativní konfiguraci, automatizaci nasazování a následnou správu Docker kontejnerů. Při provozu kontejnerizovaných aplikací může docházet k výpadkům. Pokud například dojde k pádu kontejneru, tak je zapotřebí spustit nový kontejner. Toto právě systémově řeší Kubernetes. Stará se o škálování aplikací, poskytuje vzory pro nasazení a další. [36]

5.1.5.1 Kubernetes objekty

Kubernetes do jeho prostředí (clusteru) instaluje různé typy objektů. V následujících kapitolách budou vybrané objekty, které se používají na projektu OPC a STA, představeny. Jednotlivé objekty jsou v souborech ve formátu YAML. Objekty jsou uloženy v Kubernetes systému. Kubernetes poté používá tyto entity pro reprezentaci stavu clusteru. [36]

5.1.5.1.1 Namespace

Základním objektem je namespace (jmenný prostor). Tento typ přináší mechanismus izolace skupiny objektů v rámci jednoho clusteru. Jména dalších

prostředků/objektů musí být unikátní v rámci jednoho jmenného prostoru, ale mohou existovat stejné názvy v jiných jmenných prostorech. [36]

5.1.5.1.2 Deployment

Požadovaný stav aplikace se popisuje v objektu Deployment (nasazení). Při uložení tohoto souboru mechanismus Kubernetesu změní aktuální stav v clusteru tak, aby odpovídal požadovanému stavu. Lze tvořit nové objekty nebo je také mazat, adaptovat nastavení aplikací, velikosti přidělených hardwarových prostředků atp. [36]

5.1.5.1.3 Pod

Pod je nejmenší nasaditelná jednotka Kubernetes. Skládá se z jednoho nebo více Docker kontejnerů se sdíleným úložištěm, síťovými prostředky a specifikací, jak kontejnery spustit. Konfiguraci objekt Pod přebírá od objektu Deployment popsaný výše. [36]

5.1.5.1.4 Secret

Secret je objekt, který obsahuje citlivá data jako jsou hesla, tokeny nebo klíče. Tyto informace by měly být dodány do konfigurace Podů nebo do obrazu kontejneru. Použitím Secret objektu není nutno citlivá data dávat do aplikačního kódu. Přes reference jsou při aplikaci objektu do Kubernetesu vloženy přímo do konfigurace daného Podu. [36]

5.1.5.1.5 ConfigMap

Konfigurační mapa, jak již název napovídá, slouží v uložení konfigurace. Na rozdíl od Secret objektu se zde ukládají pouze nedůvěrná data ve formátu „klíč-hodnota“. [36]

5.1.5.1.6 Service

Použitím objektu Service je možné zpřístupnit aplikaci v rámci jmenného prostoru dalším aplikacím. Každý objekt Pod dostane přidělenou IP adresu, pod kterou je dostupný. Takto lze komunikovat s aplikací bez použití Service objektu. Jenže pokud je tento Pod smazán a vytvořen znovu, je zapotřebí tuto IP adresu

změnit. Díky použití Service objektu se o toto nikdo starat nemusí a Kubernetes zařídí, aby Pod byl dostupný například pod názvem ***my-app-svc***. [36]

5.1.5.1.7 Ingress

Tento objekt rozšiřuje dostupnost aplikací mimo cluster. Jak bylo řečeno v předchozí kapitole objekt Service zpřístupní Pod v rámci jednoho jmenného prostoru. Ingress tento objekt zpřístupní uživatelům mimo Kubernetes cluster. [36]

5.1.6 Helm

Helm je šablonovací nástroj pro vytváření Kubernetes souborů popisujících objekty Deployment, Pod, Service, Ingress a další, které jsou následně vytvořeny v Kubernetes clusteru. Nástroj je psán v programovacím jazyce Go a používá šablonování pomocí hodnot v {{ }}. Například tedy když chceme na dané místo použít hodnotu ***port***, tak bude mapování vypadat takto: ***port: {{ .Values.port }}***. Díky tomuto není potřeba mít YAML soubory pro Kubernetes objekty uložené duplicitně pro odlišné prostředí, ale lze mít vícero souborů s hodnotami, které se do těchto šablon pouze vloží při nasazení. [37]

5.1.6.1 Helm Chart

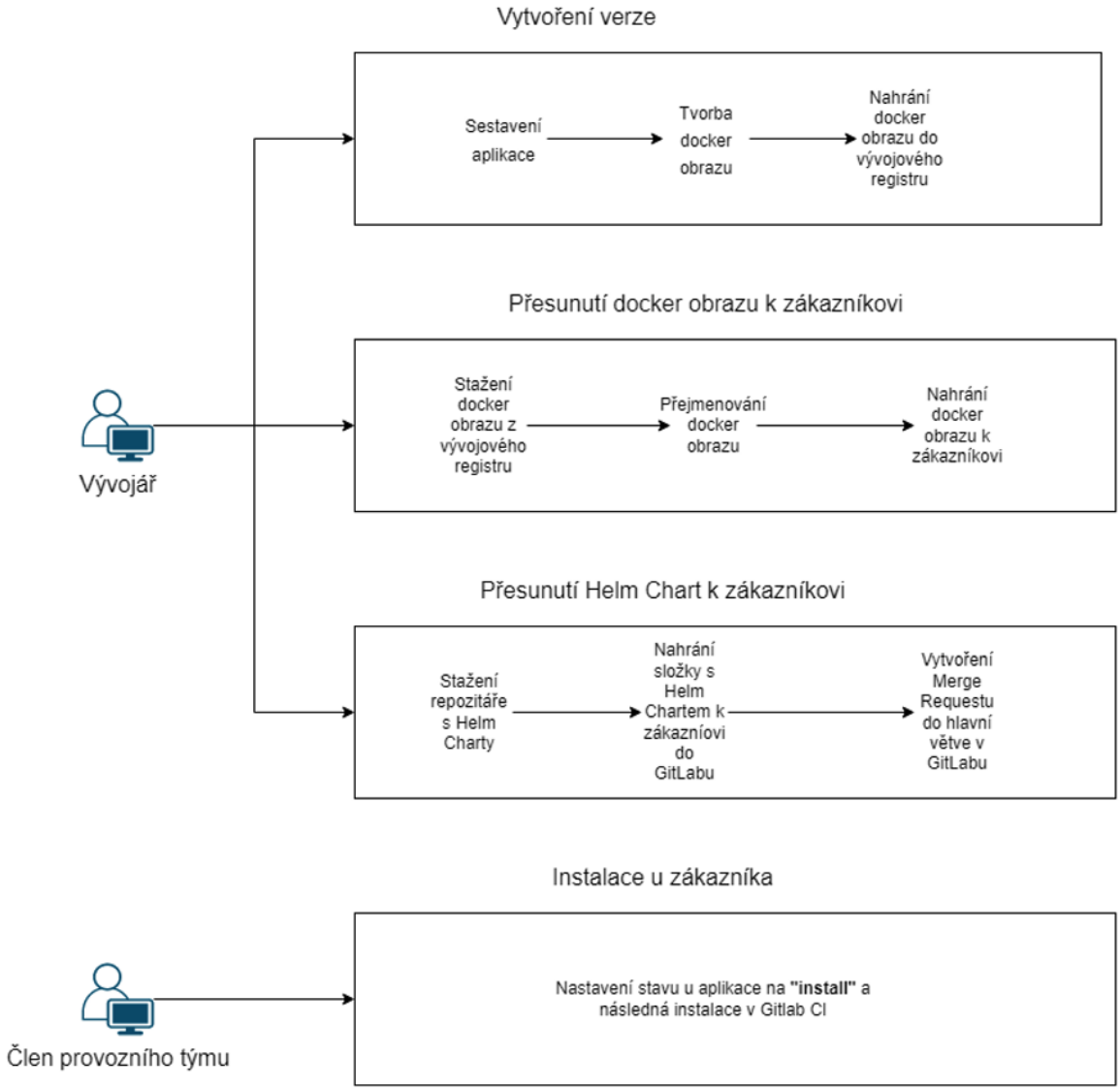
Hlavním pojmem Helmu je tzv. Chart. Tento pojem znamená složku se soubory, které popisují skupinu Kubernetes objektů. Jeden Chart je obvykle jedna aplikace, databáze, nebo jiná služba. Ve složce najdeme tyto soubory [37]

- Chart.yaml – soubor obsahující informace o dané aplikaci [37]
- README.md – soubor popisující k čemu je aplikace určena a další informace o chartu [37]
- tempates/ - složka se šablonami, které po vložení hodnot vygenerují validní Kubernetes soubory [37]
- values.yaml – soubor s hodnotami, které budou dosazeny do šablony [37]

5.1.7 GitLab

Pro uložení Helm balíčků k nasazení u zákazníka je použit GitLab a GitLab CI poté k nasazení aplikací do Kubernetes clusteru pomocí série skriptů psaných vývojovým týmem.

Na obrázku níže je ilustrována stávající CI/CD pipeline. Jedná se o sérii několika manuálních spuštění pipeline v Jenkins CI serveru.



Obrázek 9 Stávající podoba CI/CD pipeline, zdroj: vlastní zpracování

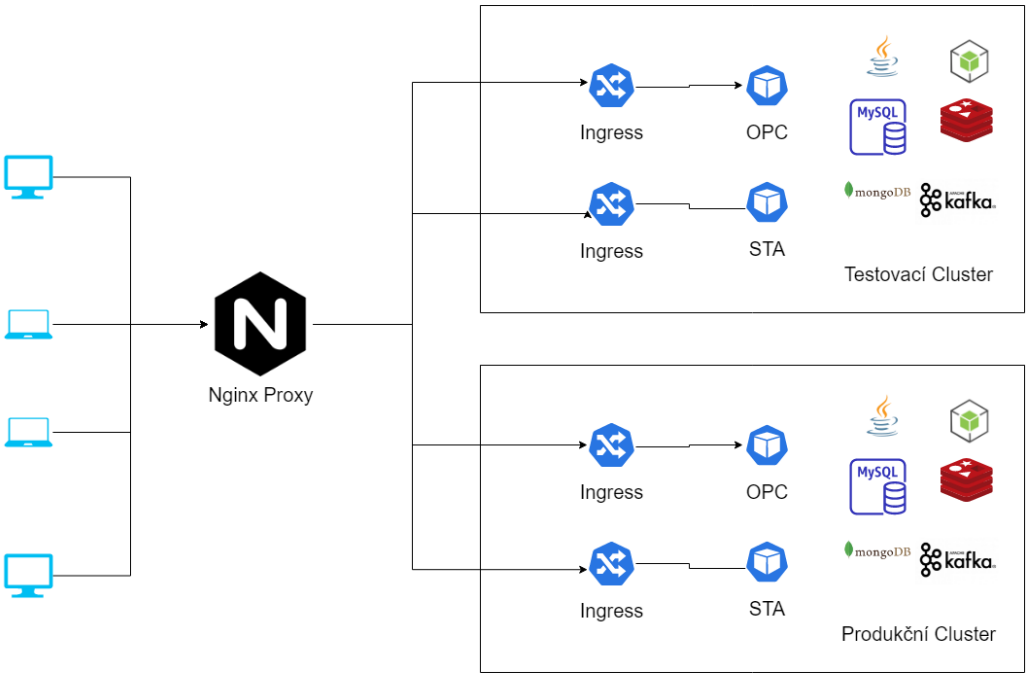
5.2 Architektura systémů

Informační systémy jsou v Unicornu vyvíjené pomocí interního frameworku UAF (Unicorn Application Framework). Integrální součástí frameworku je jednotná

metodika spolu s návody, nástroji a dokumentací. Realizačním týmům Unicornu je v rámci UAF k dispozici i podpůrný tým specialistů. Nedílnou součástí UAF je referenční implementace aplikační architektury tzv. **uuApp Framework**. Hlavní myšlenka celého konceptu je tzv. mobile-first. Aplikace jsou vyvíjeny s ohledem na mobilní zařízení a tomu je přizpůsobováno grafické rozhraní. Celý framework je psán jako REST API s klientem v podobě webové SPA aplikace.

Finální produkt je složen z tzv. mikroslužeb, které spolu komunikují přes HTTPS spojení nebo pomocí messagingu. Aktuálně se pro tento typ komunikace používá Apache Kafka. Díky REST API je celý framework platformě nezávislý. Stačí dodržet rozhraní každé aplikace a může vzájemně komunikovat služba běžící v Javě se službou běžící v NodeJS. Preferovanými programovacími jazyky jsou JavaScript společně s běhovým prostředím NodeJS ve verzi 14 a programovací jazyk Java běžící v OpenJDK ve verzi 8 nebo 11. Vše ale záleží na konkrétních potřebách daného zákazníka. Největším sektorem firmy Unicorn je energetika. Tuto oblast řeší právě i zmiňované aplikace OPC a STA.

Při vývoji OPC a STA bylo používán zejména jazyk Java. Mikroslužba v jazyce Java používá framework Spring rozšířený o implementaci průřezových problémů jako je například: logování, autentizace, autorizace, auditové logování, validace vstupů a další. Pokud je použit NodeJS, tak je mikroslužba dostupná jako webový server poskytující REST API. Obrázek níže ilustruje, jak probíhá komunikace mezi klientem a službami v Kubernetes clusteru. Vedle sebe jsou instalovány 2 clustery. Jeden je pro testovací účely a slouží k otestování nových funkcionalit či opravených chyb zákazníkem. Také se zde testují integrace do dalších systémů. Po úspěšném dokončení testů se verze produktu instaluje na produkční prostředí. V rámci clusterů je Proxy, která na základě URL adres posílá požadavky do daného clusteru. V rámci testovacího prostředí se používá sufix „acce“ (Acceptance Environment). Na základě tohoto suffixu se rozlišují požadavky na produkční/testovací cluster. Každá mikroslužba obsahuje Service a Ingress objekt Kubernetesu. Tímto je dostupná i z vnější sítě clusteru pro případné dotazování uživatelů. Aplikace mezi sebou komunikují v rámci clusteru pomocí jednotného REST API anebo přes RabbitMQ nebo Kafka message brokery.



Obrázek 10 Komunikace aplikací, zdroj: vlastní zpracování

5.3 Složení týmů

Na vývoji a následné správě/provozu projektu se podílí 2 týmy. První „DEV“ tým začal celý projekt vytvářet. Byl zodpovědný za první dodávky k zákazníkovi. Na začátku UAT (User Acceptance Tests) byl do celého procesu integrován i provozní tým. Tento tým se ze začátku staral hlavně o provoz daných aplikací, instalace do externích prostředí u zákazníka a v mezičase přebíral know-how k celému businessu a následným opravám/rozšíření obou aplikací. Tento proces je ve firmě Unicorn standardní.

5.4 Docker obraz

Aby aplikace mohly být nasazeny do Kubernetes clusteru, je zapotřebí vytvořit Docker obraz pro danou aplikaci. Jak bylo řečeno používají se Java a JavaScirpt/NodeJS programovací jazyky. V rámci OPCSTA projektu jsou vytvořeny tzv „base images“ (základní obrazy). Ty obsahují předpis pro tvorbu Docker obrazu a poté jsou používány v rámci dalších aplikací.

Je tedy nutné vytvořit dva Docker obrazy pro aplikace, na základě kterých jsou poté vytvářeny Docker obrazy pro každou mikroslužbu.

Pro Java aplikace je při sestavení vytvořen WAR soubor se sestavenou aplikací. Tento soubor je poté použit pro přípravu Docker obrazu. Pro běh aplikace v Java se používá OpenJDK ve verzi Java 8 s minimem funkcí, aby byl Docker obraz co nejmenší.

NodeJS aplikace jsou zabaleny do zip souboru a vloženy do Docker obrazu. Pro běh aplikace je použit NodeJS ve verzi 14.

5.5 Pravidla verzování

Jak již bylo zmíněno v dřívější kapitole o architektuře sytému, firma Unicorn vytváří vlastní framework pro tvorbu informačních systémů. V rámci tohoto frameworku jsou specifikována i pravidla pro tvorbu větví v Git verzovacím systému. Jedná se o lehce modifikovanou verzi standartního GitFlow.

Větve používané v rámci vývoje jsou tyto:

- Sprint – Tato větev slouží ke slučování změn od vývojářů a používá se po celou dobu vývoje, jsou z ní vydávány nové verze produktů.
- Feature – Jakmile začne vývojář pracovat na nějakém úkolu, založí si feature větev ve tvaru feature/prijmeni-jmeno-název-úkolu.
- Release – Při konci vývoje verze je tato větev vytvořena ze sprint větve. V této větvi se poté dolaďují poslední detaily před vytvořením nové verze.
- Support – Pokud se stane chyba na starší podporované verzi produktu, je vytvořena tato větev a do ní se sloučí opravy bugů, které byly nahlášeny.
- Master – Master je hlavní větev git repozitáře, když si vývojář stáhne nový repozitář, ve výchozím nastavení se mu stáhne právě tato větev. Do této větve se slučují hotové verze produktů. Je pravidlem, že k tomu dochází vždy po instalaci verze do produkčního prostředí.

Produkty jsou poté verzovány číselně podle [38] sémantického verzování. Pro finální verze se používají verze ve tvaru major.minor.patch (1.2.3) a pro vývoje

jsou využity tzv. pre-alpha verze ve tvaru minor.major.patch-pre-release (1.2.3-1). Tyto verze slouží pouze k internímu otestování a nedostávají se mimo interní použití. Toto je pak využito v novém CI procesu.

5.6 Unit testy

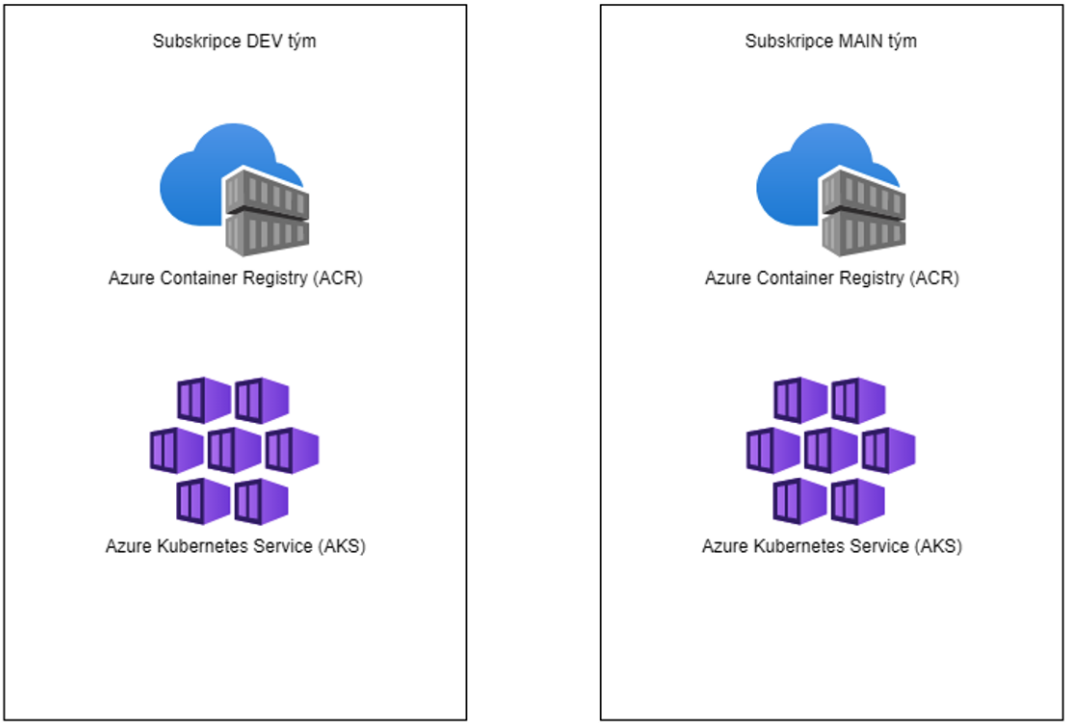
Jednotkové testování je proces při vývoji softwaru, kdy jsou nejmenší testovatelné části aplikace nazývané jednotky, individuálně a nezávisle kontrolovány, zda správně fungují. Tuto metodologii testování provádějí během procesu vývoje vývojáři softwaru a někdy testeři. Hlavním cílem tohoto testování je izolovat napsaný kód k testování a určit, zda funguje, jak bylo zamýšleno. [18] Na projektu OPC a STA došlo v minulosti k drastické proměně architektury obou aplikací a jednotkové testy se tím staly prakticky nepoužitelné. Jejich stav neodpovídal tomu, jak kód fungoval a jak bylo zamýšleno, aby fungoval. V současné době při sepisování této diplomové práce tým pracuje na opravách jednotkových testů. Jednotkové testy jsou prozatím vyjmuté ze všech CI procesů. Jedinou metrikou, jak zjistit, zda daný software funguje, jsou manuální testy.

5.7 Vývojové prostředí

Obě aplikace a všechny potřebné podpůrné technologie jsou provozovány v Kubernetes clusteru. Kubernetes byl zvolen, jelikož je dnes již defacto standardem při provozování mikroslužeb v moderních cloudových prostředích. Každý tým (vývojový i provozní) disponuje vlastní subskripcí v Azure. Azure subskripce (subscription) je základní objekt Azure, na který se dále navazují další služby a prostředky Azure [39]. Subskripce autorizuje uživatele, aby mohli používat dané prostředky zakoupené v Azure [39]. Členům týmů je poté přidělen přístup do dané subskripce a mohou pracovat s objednanými službami. Jak bylo řečeno dříve, každý tým disponuje dvěma službami (**Azure Kubernetes Service** a **Azure Container Registry**). Využitím prostředků a služeb je vyřešena celá problematika správy on-premise instalace Kubernetes clusteru. Tým se může soustředit na dodávání systému a čistě provozní problémy jejich aplikací.

Tým má svou vlastní subskripci, aby neubíral místo v ACR druhého týmu. Paralelně probíhá vývoj nových verzí a opravy případných chyb ze starých verzí.

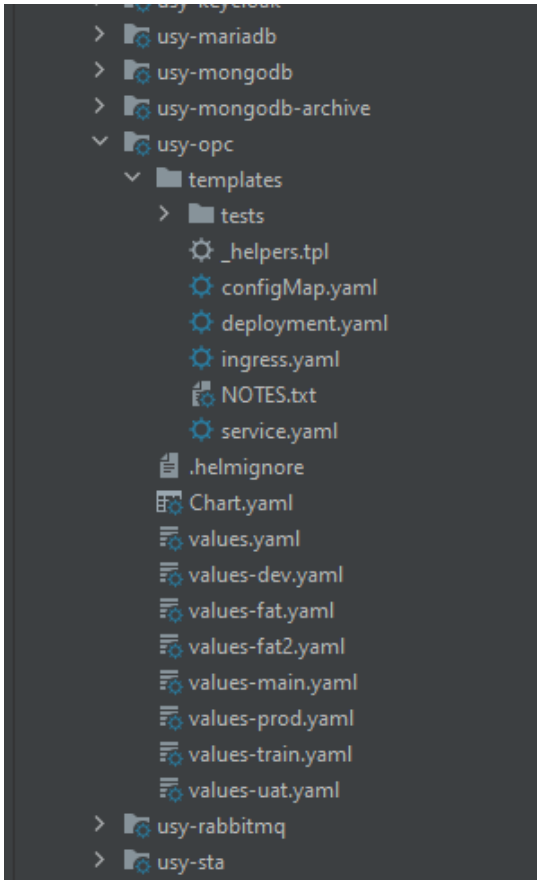
Vývojový **DEV** tým se stará hlavně o vývoj nových funkcionalit. Zatímco provozní/servisní **MAIN** tým se stará o opravy chyb z již vydaných verzí. Obrázek níže ilustruje, jak vypadají subskripce pro oba týmy.



Obrázek 11 Subskripce Azure, zdroj: vlastní zpracování

5.8 Uložení balíčků k nasazení

Definiční soubory pro Kubernetes lze ukládat v repozitáři pro následné instalování do clusteru. Nevýhodou tohoto řešení je, že ve většině případů je aplikace nasazena na několika prostředích najednou. Kvůli tomu bychom museli mít soubor duplikovaný pro každé prostředí, jelikož se někdy každé prostředí liší například počtem a velikostí hardwarových zdrojů, názvem Docker registrů atd. Již dříve byl představen nástroj Helm určený ke správě a šablonování Kubernetes souborů. Aplikace mají své Helm adresáře a soubory pro nasazování uložené ve speciálním repozitáři společně s ostatními aplikacemi a podpůrnými technologiemi. Toto je jeden ze způsobů uložení Helm chartů. Jde o tzv. „chart museum“. [40]



Obrázek 12 Adresářová struktura balíčků Helm, zdroj: vlastní zpracování

Takto jsou uloženy balíčky pro aplikace na jednom místě. Tento způsob uložení byl vybrán z důvodu podobnosti jednotlivých aplikací. Jiným přístupem může být uložení Helm balíčků jako součást repozitáře se zdrojovým kódem. Výhodou je lepší intregrovatelnost do CI procesu. Není nutno stahovat další repozitář a prodlužovat tak dobu sestavení aplikace. Nevýhodou však je, že v případě změny, která je nutná ve všech aplikacích, například přidání nové proměnné pro běh aplikace, je potřeba upravit tuto věc v každém repozitáři zvlášť. Tyto balíčky jsou uložené ve vývojovém repozitáři. Před instalací u zákazníka na externím prostředí jsou nahrány do Gitlab a rozděleny do samostatných repozitářů podle aplikací.

5.9 Continuous Integration a delivery na projektu OPCSTA

Jak bylo řečeno v teoretické části, CI server je většinou automatizační server umožňující automatizovat CI a CD procesy. Jako CI/CD server byl zvolen Jenkins. Hlavními požadavky byla možnost vlastního hostingu/nasazení na interních

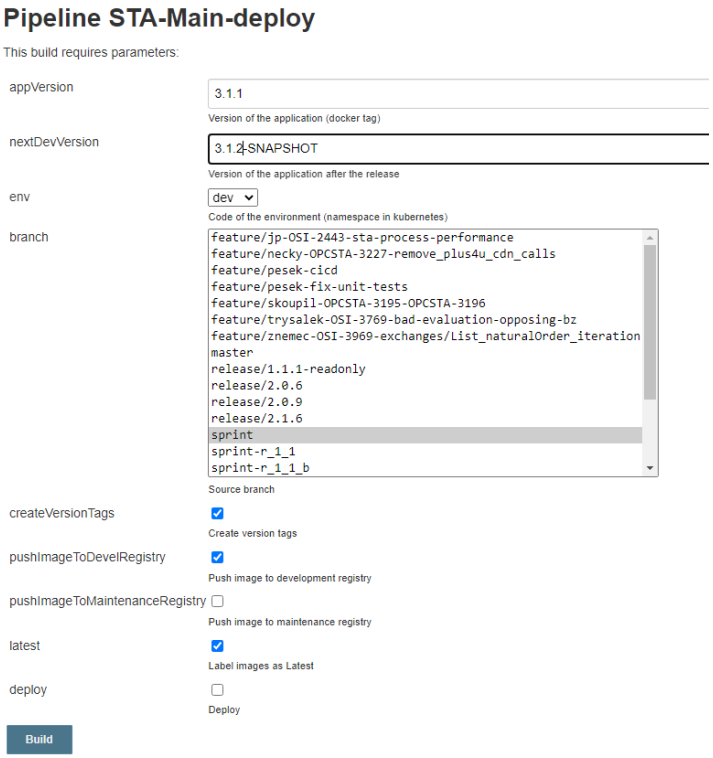
Unicorn serverech. Dalším požadavkem bylo, aby mohla služba běžet v Docker kontejneru. Vlastní hosting byl potřebný z důvodu nedostupnosti prostředí zákazníka z internetu a nutnosti nahrávat Docker obrazy k zákazníkovi. Tím byla většina cloudových řešení zamítnuta. Ke zvolení Jeknins přispěla i zkušenost vývojářů s konfigurací tohoto nástroje plus velikost jeho komunity a oblíbenost ve světě. K této technologii existuje velké množství pluginů, které umožňují integrace s dalšími technologiemi jako je Docker nebo Slack. V Jenkins serveru se odehrávají veškeré sestavení aplikací, spouštění Unit testů a přesouvání Docker obrazů mezi prostředími a k zákazníkovi. Bez tohoto řešení by veškerá práce s tvorbou verzí byla o dost náročnější. Verze Jenkins serveru byla před začátkem zpracování této diplomové práce 2.164.1 z roku 2019.

Proces CI je na projektu OPCSTA realizován několika manuálními procesy v Jenkins CI. Při vydání verze se musejí spustit tyto procesy, které budou níže popsány.

1. Sestavení aplikace a vytvoření Docker obrazu
2. Nahrání Docker obrazu k zákazníkovi
3. Nahrání Helm Chartů k zákazníkovi

5.9.1 Sestavení aplikace a vytvoření Docker obrazu

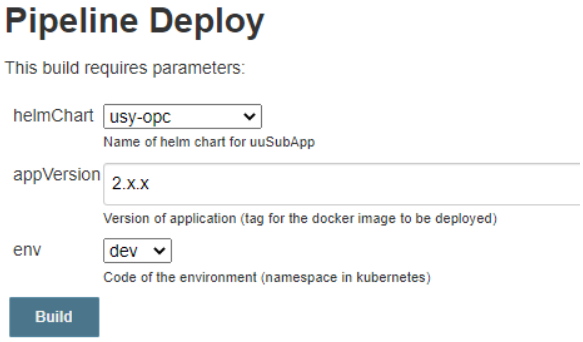
Před každým vydáním verze je nutné aplikaci sestavit. Jak bylo zmíněno v kapitole 5.6, jednotkové testy jsou v aplikacích nefunkční a jsou tudíž vyloučeny ze všech CI procesů. Tento proces tedy pouze sestaví aplikaci a tím ověří, že je aplikace spustitelná. Pokud tento krok skončí úspěšně, tak je vytvořen Docker obraz. Na obrázku níže je vidět vstupní formulář pro spuštění tohoto CI procesu v Jenkins. Uživatel vybere, jakou verzi chce vydat a jaká bude další verze pro vývoj. Poté specifikuje větev v Git repozitáři a případně na jaké prostředí chce nově vytvořenou verzi nasadit.



Obrázek 13 Vydání verze v Jenkins CI, zdroj: vlastní zpracování

5.9.2 Nasazení na vývojové prostředí

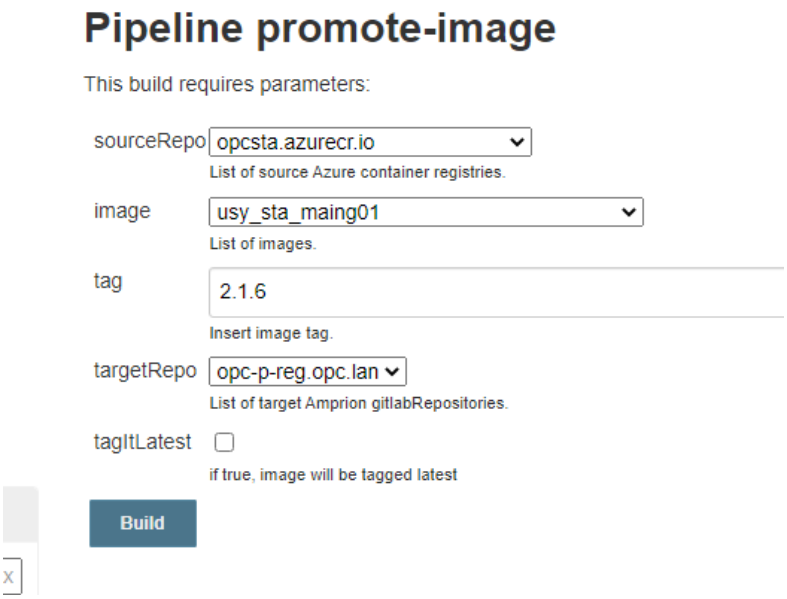
Po sestavení aplikace a vytvoření spustitelného Docker obrazu je dobré aplikaci nasadit na vývojové prostředí k otestování testery. Obrázek níže ukazuje formulář pro spuštění procesu v Jenkins, který stáhne repozitář, kde je uložený Helm Chart pro aplikaci a poté ji skrze ***kubectl*** program nainstaluje do požadovaného prostředí v AKS v MS Azure. Jako parametr stačí vybrat název aplikace, její verzi a prostředí, na které chceme nasadit.



Obrázek 14 Nasazení aplikace na vývojové prostředí, zdroj: vlastní zpracování

5.9.3 Nahrání Docker obrazu k zákazníkovi

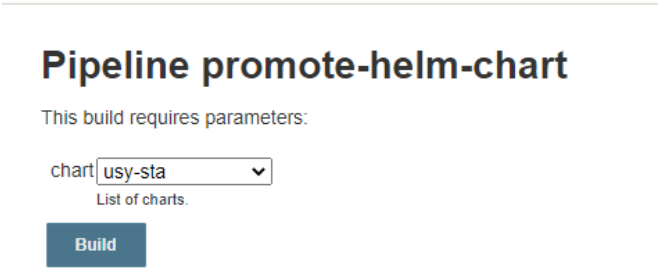
Aby aplikace mohla být nasazena u zákazníka, musí být k zákazníkovi nahrán Docker obraz s aplikací. Níže je zobrazen vstupní formulář pro spuštění procesu v Jenkins. Uživatel vybere zdrojový a cílový registr, aplikaci a verzi aplikace.



Obrázek 15 Vstupní formulář k nahrání Docker obrazu, zdroj: vlastní zpracování

5.9.4 Nahrání Helm Chartů k zákazníkovi

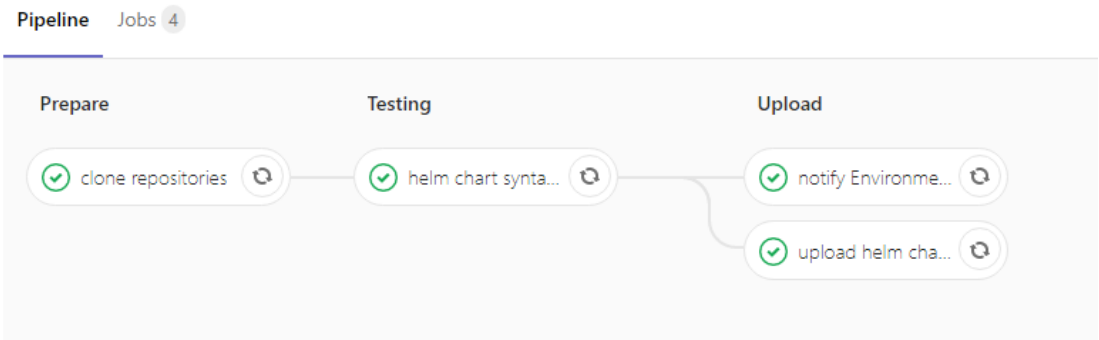
Jak již bylo řečeno, k instalacím se používají Helm Charty. Tyto soubory je potřeba, stejně jako Docker obraz, nahrát k zákazníkovi pro následnou instalaci. K tomu byl vytvořen další proces v Jenkins CI serveru. Tento proces si stáhne Git repozitář, ve kterém jsou uloženy Helm Charty pro aplikace. V repozitáři si najde hlavní (master) větev. Stáhne si příslušný repozitář pro aplikaci v GitLabu u zákazníka a tam nahraje změněné Helm Charty.



Obrázek 16 Vstupní parametry nahrání Helm Chartu, zdroj: vlastní zpracování

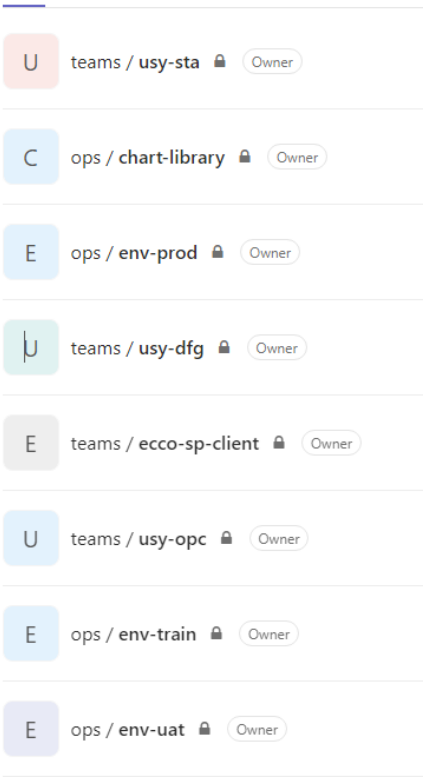
5.10 Nasazení k zákazníkovi

V kapitole 5.1 byl představen obrázek stávající CI/CD pipeline. Bylo zmíněno, že při každém vydání verze je třeba nahrát Helm Charty do Gitlabu u zákazníka. Z bezpečnostních důvodů je totiž prostředí u zákazníka, kde se nacházejí externí testovací a produkční prostředí, odstřiženo od internetu a všechny aplikace jsou dostupné jen ze Site-to-Site VPN. Obrázek níže ilustruje průběh pipeline, která kontroluje Helm Charty v případě dodání od vývojového týmů. Po nahrání Helm Chartu aplikace do Gitlabu je vytvořen Merge Request. Merge Request je žádost vývojáře o začlenění svých změn do vybrané větve. Většinou se jedná o hlavní (master) větev nebo společnou vývojovou větev (sprint, development). [29] V případě tohoto MR je vybraná větev hlavní (master). Pipeline rovněž nahraje Helm Charty do společného repozitáře. Význam chart-library (společného repozitáře pro Helm Charty) je vysvětlen níže společně s ostatními repozitáři.



Obrázek 17 GitLab pipeline, nahrání Helm Chartů

Na obrázku níže je zobrazen seznam repoziářů v GitLab s dodatečným popisem.



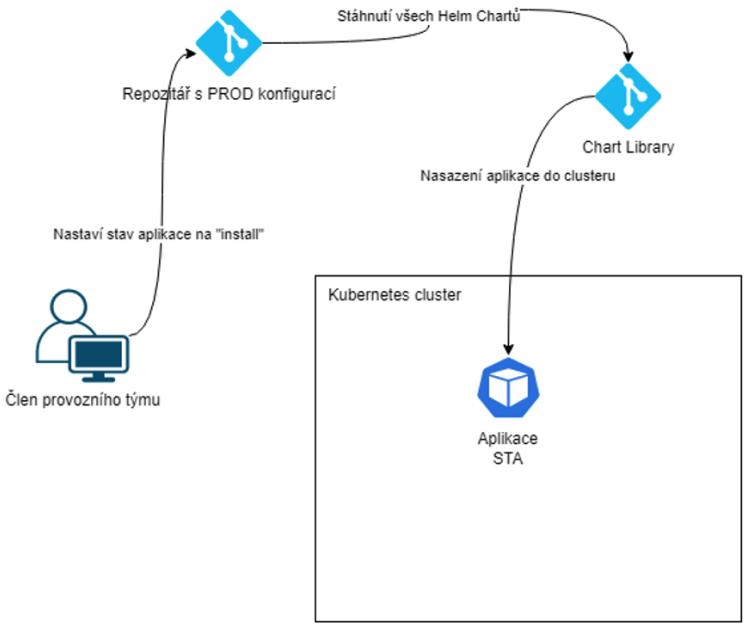
Obrázek 18 Gitlab repozitáře, zdroj: vlastní zpracování

- usy-sta
  - Tento repozitář je určen k uložení Helm Chartů jen pro aplikaci STA
- chart-library
  - Repozitář obsahuje všechny HelmCharty všech aplikací. Jsou zde složky s aplikacemi a jejich nejnovějšími HelmCharty. Při každé změně je vytvořen balíček ***.tgz*** s verzí podle verze HelmChartu. Tím je poté tým schopen nasazovat i starší verze aplikací.
- env-prod
  - Každé prostředí má svůj vlastní repozitář, ve kterém je uchována konfigurace v JSON souboru. Při změně tohoto souboru je zahájeno volání Gitlab CI pipeline, která projde soubor a podle nastavení akcí začne instalovat nebo odebírat

aplikace z patřičného clusteru. Po úspěšné instalaci do souboru zpět uloží hodnotu říkající úspěšný stav instalace/odebrání aplikace. Níže jsou uvedeny tyto stavy

- Install = installed
- Delete = deleted
- Upgrade = upgraded

Repozitář ***chart-library*** je poté používán k nasazování aplikací. Aby nemusely být stahovány všechny repozitáře najednou v momentě, kdy by došlo k potřebě nasadit více aplikací najednou. Obrázek níže ilustruje fungování CD pipeline při nasazení k zákazníkovi.



Obrázek 19 Pipeline pro instalalci u zákazníka, zdroj: vlastní zpracování

Pro instalaci aplikace je nutné v env-<prostředí kam chceme nasazovat aplikaci>.json nastavit patřičný stav aplikace. Podporované stavy jsou:

- install – nainstalování aplikace
- delete – odinstalování aplikace

- upgrade – odinstalování staré aplikace a nasazení zpět s novou verzí

```
{
    "team": "usy-sta",
    "name": "usy-sta",
    "version": "2.3.2",
    "variables": {},
    "status": "installed"
  }
```

Ukázka kódu 1 Aplikace v env-prod.json s atributy pro instalaci

Při tvorbě aplikací byl ve fázi testů v UAT (User acceptance tests) integrován do procesu instalace k zákazníkovi provozní/servisní tým. U zákazníka byl zařízen on premise Gitlab, ve kterém si provozní tým verzuje Helm charty pro nasazování do Kubernetes clusteru. Vývojový tým má svůj git repozitář, kde si v jednotlivých složkách ukládá své Helm charty. Při tvorbě verze jsou pomocí Jenkins serveru změny v Helm Chartech přeneseny i do Gitlabu a je založen Merge request. Provozní tým vyřeší vytvořený Merge request a podle pokynů v Release Notes nainstaluje pomocí Gitlab CI verzi do příslušného clusteru u zákazníka. Jak již bylo uvedeno existují 2 clustery, jeden pro testovací prostředí a druhý pro produkční.

5.11 Aktuální nedostatky CI/CD

Následující kapitola popisuje nedostatky ve CI/CD při vývoji aplikací OPC a STA ve firmě Unicorn. Pro každý bod bude v implementační části představeno navržené řešení daného problému.

5.11.1 Notifikace z Jenkins do Slack

V aktuální implementaci se jednotlivé kroky vytvoření verze spouští manuálně. Vývojář tedy spouští několik úkolů (pipelines) v Jenkins CI aplikaci. V novém řešení by měl být tento proces částečně automatizovaný. Díky tomu by vývojář nemusel vůbec chodit do Jenkins uživatelského rozhraní, ale v aplikaci Slack by mu přišla zpráva o úspešném/neúspěšném provedení dané úlohy v Jenkins. Pro integraci se Slack je nutné provést update Jenkins z verze 2.1.64.1 na 2.289.1 (aktuální v době psaní diplomové práce a průběhu aktualizace Jenkins CI).

5.11.2 Řazení spouštění aplikací

Ve světě mikroslužeb je systém tvořen několika aplikacemi, které spolu komunikují pomocí REST API a message brokerů. Aplikace OPC a STA sdílejí některé komponenty (například konfigurační mikroslužbu, ve které se ukládají konfigurace aplikací a nastavení businessových procesů). Po startu hlavní mikroslužby se aplikace připojí na konfigurační mikroslužbu a požádá ji přes HTTPS spojení o potřebná data. Pokud tato aplikace není nasazena nebo je v nedostupném stavu, nemůže hlavní aplikace nastartovat. Díky těmto závislostem může docházet k nasazení, které je ukončeno chybou. Je tedy nutno pak analyzovat logy a hledat, kde přesně nastal problém při startu aplikace.

5.11.3 Nezabezpečená hesla v Git repozitáři

Pro správu balíčků pro nasazení se používá nástroj Helm. Pro uložení balíčků existuje samostatný repozitář. Každá aplikace obsahuje persistentní uložení dat do MongoDB nebo MySQL databází. Při nasazení aplikace je nutno předat aplikaci data potřebná k připojení k dané databázi. V rámci těchto dat se předává uživatel a heslo, pod kterým se aplikace přihlašuje k databázi. V databázi je pak nastavené, na jaké databáze má daný uživatel práva a k jakým objektům může přistupovat. Tato data, zejména pro produkční prostředí, by měla být chráněna. Helm má uložená jednotlivá data v YAML souboru a tato data jsou poté vkládána do šablon a Helm z nich při instalaci do clusteru vytvoří Kubernetes manifest soubory. Je tedy zapotřebí tyto data mít šifrovaná, aby vývojáři neměli přístup k heslům do databází.

5.11.4 Manuální spouštění CI pipeline

Aktuální implementace CI je realizována pomocí pipeline v Jenkins, kde si vývojář specifikuje větev a verzi aplikace, kterou chce vydat. Jenkins server si poté stáhne repozitář s aplikací, provede sestavení aplikace a vytvoření Docker obrazu s aplikací. Tento obraz nahraje do vývojového Azure Docker Registry. Na konci označí místo v git historii pomocí git tag příkazu ve tvaru X.Y.Z (například 1.2.3). Git tag se pak používá, pokud chce vývojář řešit zadaný bug z externího prostředí a potřebuje si načíst místo v historii git repozitáře, kdy byla daná verze vydána. CI

proces by měl být automatizovaný tak, aby nemusel vývojář „ztrácet“ čas v Jenkins CI aplikaci, ale mohl se věnovat tvorbě kódu.

5.11.5 Manuální přesun verze k zákazníkovi

Jak bylo zmíněno v předchozím bodě stávající proces vydávání softwaru není automatizovaný. Proces sestavení aplikace skončí u nahrání Docker obrazu do Azure Docker Registry. Jelikož jsou externí prostředí odstřižena od internetu, tak musí být veškeré Docker obrazy umístěné v Docker registrech umístěných v místě, kde jsou provozovány aplikace. Nyní je řešeno toto pomocí manuální pipeline v Jenkins, ve které se specifikuje odkud se má stáhnout obraz, jaké aplikace a kam se má nahrát (existují totiž 2 registry pro každé prostředí zvlášť). Tento krok by šel minimálně automatizovat, jelikož při vydání finální verze ve tvaru X.Y.Z je jasné, že tato verze se bude instalovat i na externí prostředí a bude dodávána zákazníkům.

5.11.6 Aplikace se po pádu znovu nerestartuje

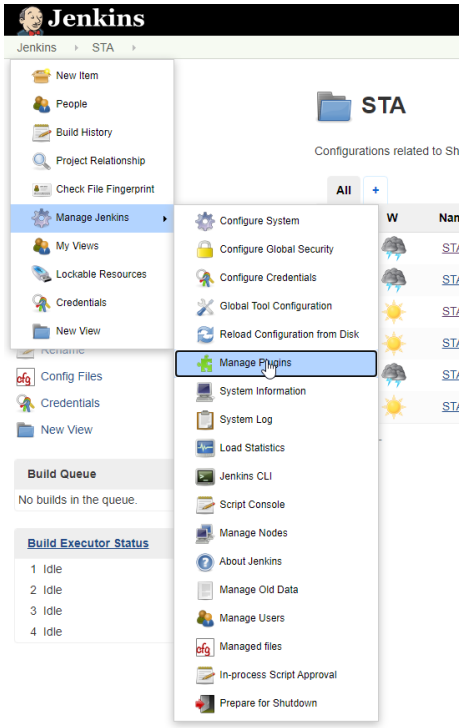
Při běhu aplikace se může stát, že dojde k pádu aplikace, po kterém se už aplikace nevzpamatuje a zůstane zastavená. Toto může být poté nahlášeno dohledovým systémem a administrátor se poté připojí na cluster a aplikaci opět uvede do provozuschopného stavu. V rámci Kubernetes technologie existují některé funkcionality, pomocí kterých je možné toto automatizovat/nakonfigurovat a předejít nedostupnosti systému. Tímto chováním lze v budoucnu zabezpečit, že se aplikace dokáže sama „opravit“ a uvést do provozuschopného stavu, aniž by ji musel někdo z provozního týmu ručně smazat a nasadit znovu do Kubernetes clusteru. Tento nedostatek se netýká přímo fungování CI a CD, ale jedná se spíše o provozní záležitost běhu aplikací.

6 Implementace změn v CI/CD

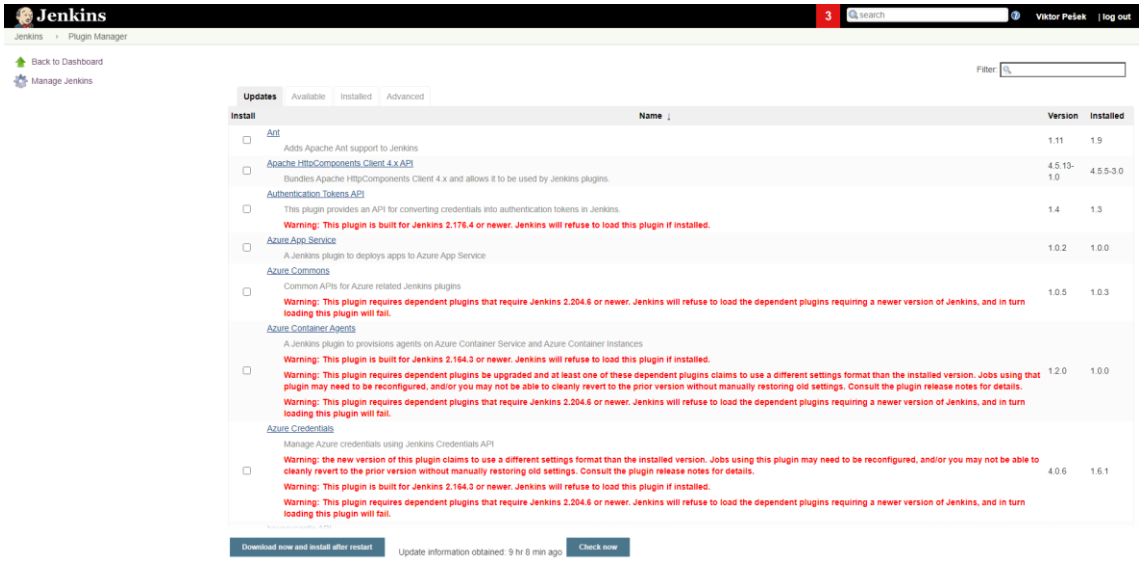
V nadcházející kapitole budou popsány implementace jednotlivých nedostatků CI a CD popsaných v kapitole 5.11.

6.1 Aktualizace verze Jenkins CI

Na projektu OPCSTA se jako CI server používá Jenkins server. Tým má k dispozici virtuální stroj, na kterém běží Docker. V Dockeru běží poté Jenkins server a má připojené 2 disky (volumes). Jeden oddíl je určený pro data Jenkins serveru a druhý je určený pro předání složky docker.sock (socket kam si Docker ukládá data). V době sepisování této diplomové práce byl nainstalovaný Jenkins ve verzi 2.164.1. Tato verze byla vydána v roce 2019 a je již zastaralá. Jak bylo dříve zmíněno tato verze nepodporuje integraci s platformou Slack. Níže je obrázek ukazující stránku s pluginy pro Jenkins. Většina pluginů je také již v zastaralé verzi a bylo potřeba je aktualizovat. S tím bylo nutné aktualizovat i celý Jenkins, jelikož nové pluginy použité v pipelině, která vznikla v rámci diplomové práce využívala pluginy, které na starší verzi nešly nainstalovat kvůli jiným pluginům, na kterých byly implementačně závislé. Do správce pluginů se lze dostat přes hlavní menu -> *Manage Jenkins -> Manage Plugins.*



Obrázek 20 Jenkins Plugin Management, zdroj: vlastní zpracování



Obrázek 21 Jenkins Plugins, zdroj: vlastní zpracování

6.1.1 Použití Ansible pro nasazení Jenkins CI

V rámci infrastruktury projektu OPCSTA existují virtuální stroje, na kterých běží Jenkins v Docker kontejneru. Při každé aktualizaci je nutné se připojit ke každému stroji a tam ručně spustit příkaz ***docker run*** s příslušnými parametry. Také je potřeba do Docker kontejneru nainstalovat nástroje používané v rámci

vývoje a vydání verze. Při vydání verzi se používá nástroj ***kubectl*** a ***helm***. Kubectl slouží ke komunikaci s Kubernetes clusterem a nasazení do testovacího prostředí a pomocí příkazu ***helm*** se vkládají do šablon hodnoty, které se používají pro generování kubernetes souborů při instalacích. Provádět tyto kroky ručně při každém startu Docker kontejneru je časově náročné a je mnohem lepší celou tuto problematiku automatizovat a použít „custom“ Docker obraz. Docker umožňuje upravit si Docker obraz podle vlastních potřeb pomocí vytvoření nového Dockerfile, ve kterém se doinstalují potřebné technologie. Dockerfile byl představen v kapitole 5.1.1.1. Pomocí tohoto souboru lze zautomatizovat vytvoření Docker obrazu zavoláním posloupnosti příkazů [33]. Níže je ukázka Dockerfile pro vytvoření modifikovaného Docker obrazu, který vyhovuje potřebám týmu.

```
FROM bitnami/kubectl:1.20.9 as kubectl
FROM jenkins/jenkins:2.289.2
MAINTAINER Viktor Pesek <viktor.pesek@unicorn.com>
USER root
RUN curl -sSL https://get.docker.com/ | sh
RUN apt-get install sudo -y
RUN echo 'jenkins ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers
COPY --from=kubectl /opt/bitnami/kubectl/bin/kubectl
/usr/local/bin/
RUN curl https://baltocdn.com/helm/signing.asc | sudo apt-key
add - \
    && sudo apt-get install apt-transport-https --yes \
    && echo "deb https://baltocdn.com/helm/stable/debian/ all
main" | sudo tee /etc/apt/sources.list.d/helm-stable-
debian.list \
    && sudo apt-get update \
    && sudo apt-get install helm
USER jenkins
```

Ukázka kódu 2 Dockerfile pro Jenkins CI, zdroj: vlastní zpracování

Začátek tvorby obrazu je prováděn s právy root uživatele. Je nainstalován Docker a program sudo, aby uživatel jenkins, pod kterým se spouští kontejner mohl

provolávat ***docker*** příkaz a vytvářet Docker obrazy pro vyvíjené aplikace. Z bitnami Docker obrazu se zkopíruje kubectl do složky /usr/local/bin a nainstaluje se helm v poslední verzi a provede aktualizaci systému. Nakonec předává pomocí USER direktivy řízení jenkins uživateli. Takto upravený Docker obraz lze nasadit na Jenkins CI server. Toho lze docílit puštěním příkazu ***docker run*** s příslušnými parametry. Tento krok vyžaduje shh připojení na server a spuštění daného příkazu. Pro spouštění Jenkins serveru bude využito programu Ansible. Takto může update udělat kdokoliv, kdo zná heslo k Jenkins CI serveru.

Ansible je open source nástroj, který automatizuje správu konfigurace, nasazování aplikací, orchestraci a mnoho dalších IT procesů. [41] Je ve světě hojně využíván a obsahuje množství pluginů například i pro práci s Docker platformou. Ansible je dostupný pouze na unixové platformě. Vyžaduje instalaci pouze na straně klienta. Pro komunikaci se serverem využívá ssh připojení. Ssh připojení musí být povolené na straně serveru. Další konfigurace není vyžadována. Ansible se skládá z tzv. playbooků (hrací kniha). Jednotkou práce v rámci playbooku je task (úkol). Pomocí playbooků lze pouštět úkoly v daném pořadí. Pro spuštění Jenkins CI serveru byl vytvořen playbook s jedním úkolem, a to spustit Docker kontejner. Každý úkol, v tomto případě **docker_container,** může být parametrizován. Pro spuštění Jenkinse musí vědět, kde najde svou domovskou složku. To je obsaženo v rámci volumes parametru. Jenkins jako takový bude přistupovat k Docker kontejneru na samotném serveru zevnitř Dockeru. Proto je zapotřebí přidat mapování k Docker socketu. Dostupný bude na portu 8080. A pro komunikaci s Gitlabem u zákazníka je také nutno přidat záznam do /etc/hosts v kontejneru. Politika restartování bude unless-stopped. To znamená, že kontejner se restartuje jedině, když mu to je zvenku přikázáno.

```
---
- name: Run Jenkins in Docker
  hosts: all
  vars:
    container_name: "jenkins"
```

```
    container_image: "viktorpesekunicorn/jenkins-
opcsta:latest"
    my_etc_hosts:
      {
        "gitlab_url ": "IP",
      }
  tasks:
    - name: Create Jenkins docker container
      docker_container:
        name: "{{ container_name }}"
        image: "{{ container_image }}"
        etc_hosts: "{{ my_etc_hosts | default({}) }}"
        volumes:
          - "/opt/jenkins:/var/jenkins_home"
          - "/var/run/docker.sock:/var/run/docker.sock"
        ports:
          - "8080:8080"
        restart_policy: unless-stopped
```

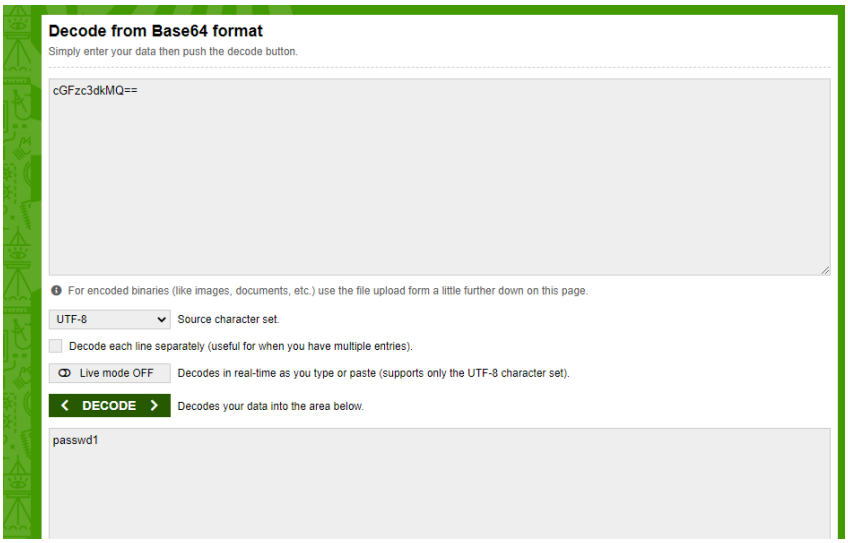
Ukázka kódu 3 Ansible playbook pro Jenkins CI, zdroj: vlastní zpracování

Takto připravený playbook stačí pouze spustit. Před spuštěním je zapotřebí definovat, na kterém serveru se má daný playbook pustit. Ansiblu se vytvoří soubor s IP adresou Jenkins CI serveru a ten je poté přidán jako parametr příkazu ansible-playbook. Příkaz pro spuštění ansible playbooku je: ***ansible-playbook -i hosts/jenkins-test main-playbook.yaml --user root -k.*** Parametrem -i je specifikován inventář s adresami, parametrem ***–user*** říkáme, pod kterým uživatelem se chceme připojit ke vzdálenému serveru, a ***-k*** slouží pro vyzvání ansiblem k zadání hesla. Určitě není bezpečné zadávat heslo přímo do příkazového řádku. Na sdíleném PC by poté bylo možné dohledat heslo přes historii vykonaných příkazů.

6.2 Zabezpečení hesel aplikací

Stejně jako při vývoji softwaru i při vývoji Kubernetes manifestů a celé infrastruktury aplikace je správný postup ji verzovat a mít ji uloženou v git

repozitáři. Tento postup lze aplikovat na všechny Kubernetes objekty mimo objekty typu Secret. Secret je objekt, který obsahuje malé množství citlivých dat jako je heslo, token nebo klíč. Takové informace by jinak mohly být vloženy do specifikace podu nebo kontejneru. Použití tohoto objektu znamená, že do kódu aplikace nemusí být zahrnuta důvěrná data [36]. Také není nutno při změně hesla např. do databáze vytvořit nový Docker obraz. V případě, že by heslo bylo v kódu, bylo by toto nutností při každé změně. Secret ukládá tento citlivý obsah zakódovaný v base64. Jak už napovídá název „zakódovaný“ nejedná se o šifrování. Tím pádem není zaručena bezpečnost a kdokoliv s přístupem k uloženému souboru se Secretou je schopen ji dekódovat a případně zneužít. Proto se výrazně nedoporučuje takto nechráněné objekty ukládat do Gitu, do kterého mají přístup i lidé, kteří by například hesla k produkční databázi znát neměli. Na obrázku níže je vidět, jak jednoduše lze z Base64 dostat citlivé údaje v plain textu.



Obrázek 22 Dekódování Base64, zdroj: https://www.base64decode.org/

6.2.1 SealedSecrets

Sealed Secrets umožňuje zašifrovat citlivá data, aby mohla být uložena ve sdíleném git repozitáři a vývojáři se nedostali například k heslům do produkčních databází. Sealed Secrtes plně využívá vysokou škálovatelnost Kubernetes. Zavádí nový objekt SealedSecret a ten vytváří pomocí CRD. Ukládá zašifrovaný obsah do

rozšířeného SealedSecret objektu. Tento objekt může dešifrovat pouze controller běžící na cílovém clusteru. Pro celou dešifrovací a šifrovací proceduru se používá asynchronní kryptografie s použitím soukromého a veřejného klíče. [42]

Pro fungování SealedSecrets je potřeba nainstalovat si lokálně kubeseal příkaz, pomocí kterého se zašifruje objekt Secret. V Kubernetes clusteru se musí nainstalovat SealedSecret controller který se stará o šifrování a dešifrování (*pro šifrování můžeme vyextrahovat soukromý klíč v .pem formátu a použít ho jako parametr příkazu, pozn. autora).* Při instalaci/smazání používáme již jen SealedSecret objekt. Při vytvoření SealedSecret poslouchá SealedSecret controller. Ten dešifruje informace a vytvoří standartní Secret objekt. Ukázky kódu níže ilustrují průběh vytvoření zabezpečeného Secret objektu z obyčejného Secret objektu a jeho zpětné dešifrování po nahrání do clusteru. V poslední ukázce je vidět, že při každém dešifrování si jej SealedSecrets controller zaznamená ve svém logu.

```
wget https://github.com/bitnami-labs/sealed-
secrets/releases/download/v0.16.0/kubeseal-linux-amd64 -O
kubeseal
sudo install -m 755 kubeseal /usr/local/bin/kubeseal
```

Ukázka kódu 4 Instalace kubeseal, zdroj: vlastní zpracování

```
kubectl apply -f https://github.com/bitnami-labs/sealed-
secrets/releases/download/v0.16.0/controller.yaml
```

Ukázka kódu 5 Instalace SealedSecret kontroleru, zdroj: vlastní zpracování

```
apiVersion: v1
data:
  password: cGFzc3dkMQ==
  user: cm9vdA==
kind: Secret
metadata:
  name: my-secret
  namespace: test
```

Ukázka kódu 6 Soubor se Secret objektem, zdroj: vlastní zpracování

```
kubeseal < secret.yaml -o yaml > sealed-secret.yaml
```

Ukázka kódu 7 Příkaz na vytvoření SealedSecret, zdroj: vlastní zpracování

```
apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  creationTimestamp: null
  name: my-secret
  namespace: test
spec:
  encryptedData:
    password:
AgAm5XED2hkQTfg4g2aam7yl55TNO6n67uKfpdHRzT3HI7zC2kwqbuZP+0CcKe
Om/enOs6T8uRQ8cr87HohhuKD4w9wKD+9M2OpNvetD2QzYPMxR0BvnpvG5EIdF
vGXRBtpirnVO+y7DyGlM5vSCEQyb6XgBW2Y4ITMsropSkD1wspwOxfUTtqhlU2
be7D5viv2RyXWGICpzxDjCIbZ7KietUyECRrxIqDlR53K0NL4V8zOGCouL9xuc
rQH+Bqj7tRJqhcTx6vdlikB1JMJXfpI9jV8vubJpkkWv5ybsGu4Xo91+gw1UOn
mDmtlTDrnlgJ7xqn+WwMS42HVm1+hZAtHGJ2ZBauE/KxZ1aPQdyhMJ7CmnJTB+
7W7BWVN/pfB4QdXRRP4VXtft0ZdwRs2hka13vPeJTI2s5oz8HzBeR891lLSgDa
LGvyGlDCV2jADjXpnlaEZY2EEgAoO4QhdGJ1fZoZqKBMjk9B4JaJZAvyvLf5EZ
nuTDHhhj8SK6+OKiZePGoyFUW9LZzdVkz7J2VaQhXPWKzmCwBZAI2yZ9V7hAp7
Hdx0JwCPQZI7+40B3MO5guUuynlBhI7Dpfw6kV3dpQhTKgM1rkf3Paxj7kFQJy
tO4VtbNvT8PB49WKP/Wl9bD4mr6LdaGBGSBQD0AFCs1UutP7v2xBTIp2sHKg5X
pwIKxJhZuqLRrczxC6uOgM/DOTHAFfgLTL
    user:
AgBD9m+wNyQTgqRjwjLJnqyLQTK8vh4w0QgiAFjTYfijQB53Viq/cH+vakvxoh
S2j6YPQ0UVYFAwYDn+VwUGRkheCSdBI795HraaL+kS/OgC4yISH9Fku9wSgGEO
ip4WoaJ2qlwusVB7WlDwOqXclg8j4zNQ/cvP1tD5jkKZvdjI33WswGCCJrMN0F
j+0KiYq0bv48dX4kCY09HRtwl0C00Q85kibWcNfIa7khJYYHXfSqopd1XKvVZR
S0gymqyX4XiDFQ4lh0aXWGTzLH9xHd8rjSr2u1jxjGnd7cMFr9b7IA0BQDRm3Z
+qd783pbJTZ3SAEQ527wYGMRtegjrMF3/46LzCvLIvZ420mOJlXNYdn9SV8qCE
L131Va96jkMhqscU7RlW2E/NOPrFWna0HGVl/IsGFr3nf2v3VVjwkzEHu3/ey2
ZSqveR8i3ehNph0KyXc1YQN/Cmp9xtk2XShiGtmqLYkbpe0BZpk0W9ItC2s5rd
yyoIdDqbJtcdh+WTXVid/gQZjstJO/FQCiyCMSoFaipukxUTHjfHT5Ew1unLJO
```

```
1RaTkaBiNJl+pzPaMgagm6bajiR/WVHESBGnZCdHjcTHLY0xdymLFTNOJquYVa
pO6GhdA7tYlyHPyj4PskbCl8xgqgi9qoAZq5s+0ZjayLMwVNaKNSoff6MfhWJC
fTaTZ372CtPwu1y7OAz6hl7diPMOCz
  template:
    data: null
    metadata:
      creationTimestamp: null
      name: my-secret
      namespace: test
```

Ukázka kódu 8 Soubor se SealedSecret, zdroj: vlastní zpracování

```
k logs -n kube-system sealed-secrets-controller-7bdbc75d47-
4mjwm

2021/11/17 10:27:41 Updating test/my-secret
2021/11/17 10:27:41
Event(v1.ObjectReference{Kind:"SealedSecret",
Namespace:"test", Name:"my-secret", UID:"5430e542-b714-48fe-
bba8-a0970a1973e8", APIVersion:"bitnami.com/v1alpha1",
ResourceVersion:"1619", FieldPath:""}): type: 'Normal' reason:
'Unsealed' SealedSecret unsealed successfully
```

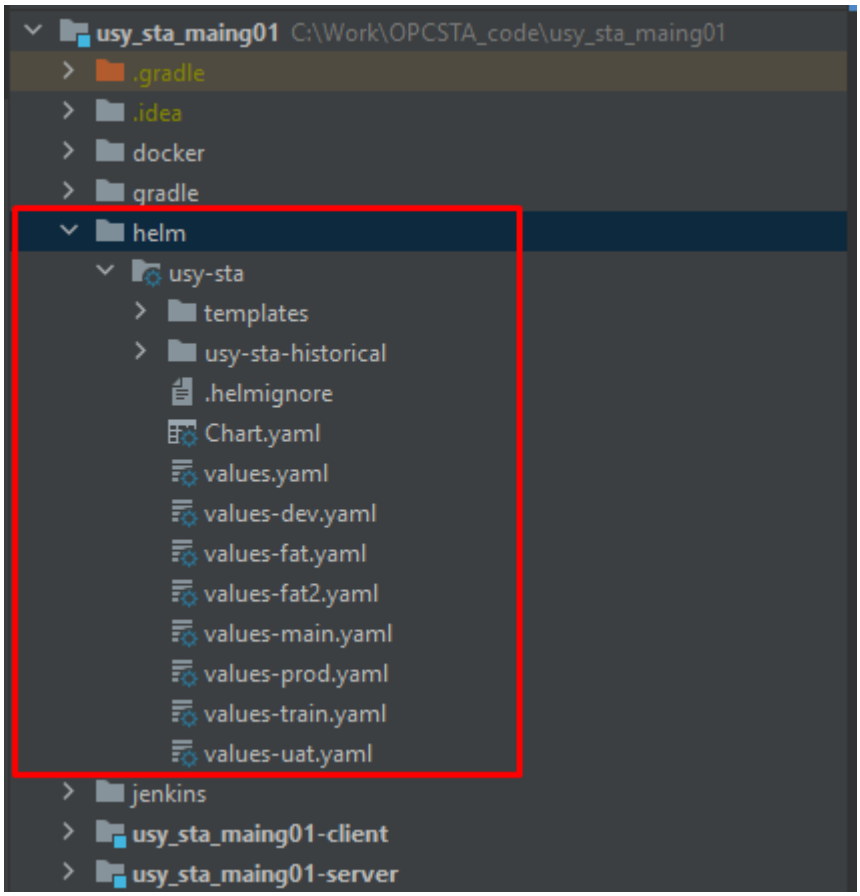
Ukázka kódu 9 Ověření dešifrování, zdroj: vlastní zpracování

Takto hotovou SealedSecret může každý bez obav o bezpečnost nahrát do veřejně přístupného git repozitáře. Důležité je zmínit, že při procesu šifrování hraje roli i název původního objektu Secret s citlivými daty a namespace (jmenného prostoru, pozn. autora), do kterého se bude Secret objekt instalovat. Pokud bychom se pokusili Secretu nahrát do jiného namespace, tak by nám controller danou Secretu odmítl dešifrovat. V rámci OPCSTA se používá šablonovací nástroj Helm. SealedSecrety budou proto poté pomocí šablony vloženy podle toho, na jaké prostředí budou aplikace v danou chvíli instalovány.

6.3 Jenkins CI pipeline

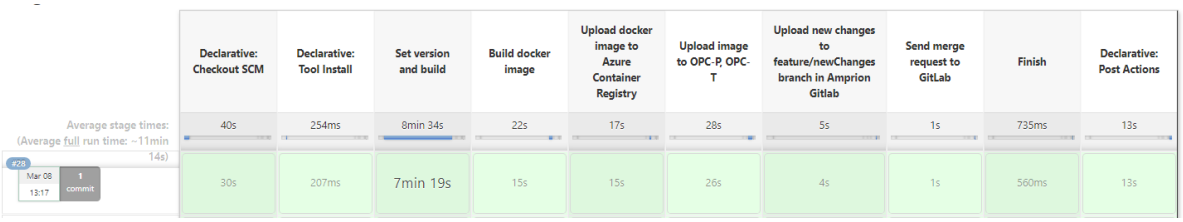
Původní použití CI se skládalo z několika po sobě jdoucích pipeline, které se pouštěly manuálně. Toto bylo představeno v předchozí kapitole. Jak bylo také

představeno v kapitole 5.8 všechny složky s Helm Charty pro aplikace jsou uloženy v samostatném repozitáři. V novém řešení bude v repozitáři s kódem uložen i Helm Chart pro danou aplikaci. Tím se standardizuje CI proces a půjde jednodušeji nahrávat změny v HC k zákazníkovi. Níže je obrázek nové struktury složek v git repozitáři. Nově přidaná je složka ***helm***.



Obrázek 23 Nová struktura git repozitáře, zdroj: vlastní zpracování

Níže bude představena nová pipeline, která automatizovaně provádí build, tvorbu Docker obrazů a v případě vydání finálního buildu i označení pro použití v testovacím a produkčním clusteru u zákazníka. Jednotlivé kroky nové pipeline budou níže rozepsány. Na obrázku níže je ilustrace nové CI pipeline. Obrázek je převzat z Jenkins CI serveru.



Obrázek 24 Detail nove CI pipeline, zdroj: vlastní zpracování

Set version and build

Vývojář nastaví, jakou verzi chce vydávat. V případě DEV fáze, kdy chceme nejdříve novou verzi interně testovat, nastaví X.Y.Z-SNAPSHOT. Toto je commitnuto do git repozitáře do speciálního souboru. Z tohoto souboru pak čte API a vrací nám přes REST rozhraní na endpoint /getHealth stav kontejneru a verzi nasazené aplikace. Při vydávání verzí v rámci DEV fáze se používají Docker tagy X.Y.Z-SNAPSHOT a X.Y.Z-SNAPHSOT-${BUILD_NUMBER}. Build number je převzato z čísla sestavení v rámci Jenkins CI serveru. V případě zadaného bugu na danou verzi se developer podívá do detailu sestavení s číslem v git tagu a podle změn v gitu si dohledá potřebný commit hash, který mu následně slouží k určení toho, které změny jsou v Docker obrazu a které ne.

Build Docker image

V tomto kroku se vytvoří Docker obraz a ten se označí podle daných pravidel a pošle do příslušných registrů. Pravidla tvorby Docker obrazů jsou tato:

- Pokud je větev /sprint/ nebo /release/ pošli Docker obraz do vývojového Azure Container Registry.
- Pokud je větev /support/ nebo /hotfix/ pošli Docker obraz do servisního Aure Container Registry.
- Pokud je verze aplikace ve tvaru X.Y.Z. pošli ho do obou interních a také externího JFrog Container Registry k zákazníkovi (používá se jen v případě vydání final buildu) pro UAT a PROD prostředí.

Upload Docker image to Azure Container Registry

Tento krok slouží k nahrání do Azure Container Registry. Jak bylo nastíněno v kapitole 5.11.6, není žádoucí, aby v obou registrech byly všechny verze, které týmy

vytvoří. Výjimky však tvoří finální verze aplikací. Ty by měl mít dostupné každý tým. Proto při vydání finální verze budu aplikace s verzí ve tvaru X.Y.Z (například 2.1.5) uloženy do obou ACR.

Upload Docker image to OPC-P, OPC-T

Zde se opět nahrávají Docker obrazy, ale do externích JFrog Container Registry u zákazníka pro následné nasazování do offline prostředí bez přístupu k internetu.

Upload new changes to branch in Gitlab

Stejně jako Docker obraz jsou nahrány i změněné Helm Charty k zákazníkovi do jeho GitLabu. Ze složky ***helm*** jsou nahrány do příslušného repozitáře v GitLabu u zákazníka.

Send merge request to GitLab

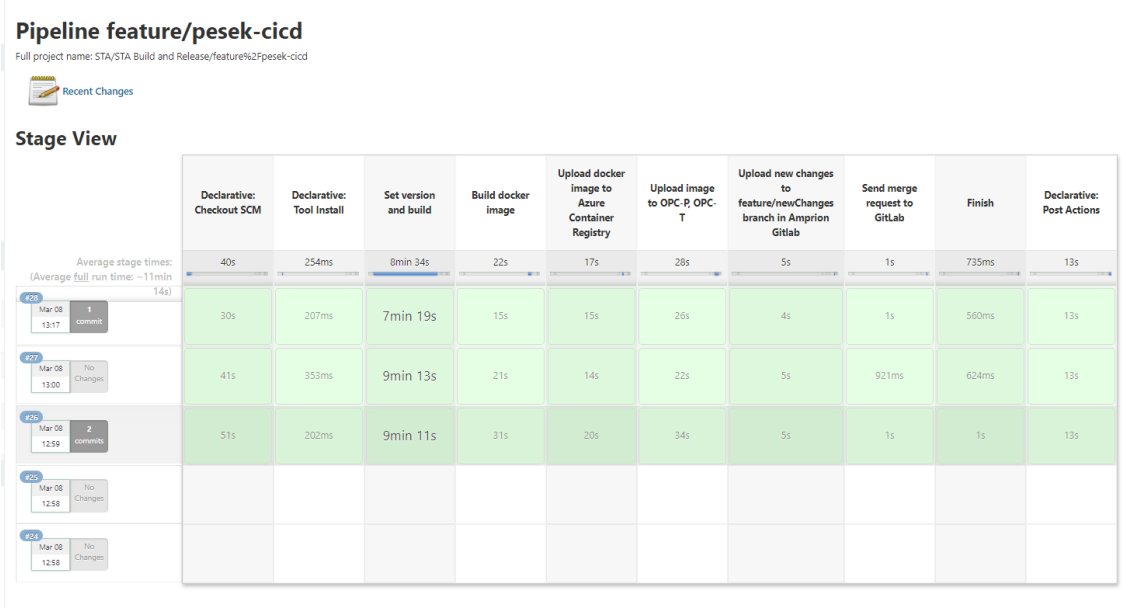
Na úplném konci je založen požadavek na sloučení (merge request) větve s novou verzí aplikace do hlavní (master) větve.

Finish a Declarative Post Actions

V rámci těchto dvou posledních kroků se smažou všechny Docker obrazy vytvořené v průběhu běhu sestavení, aby nebylo plýtváno místem na disku virtuálního stroje, kde je Jenkins CI server nainstalovaný.

Pro další ulehčení a zkrácení času jsou node_modules pro javascriptové aplikace a ReactJs frontend cachovány. Tímto se zrychlí čas instalace balíčků a sestavení aplikace. Jenkins CI server disponuje pouze jednou MASTER nodou, takže všechny sestavení probíhají na jednom virtuálním stroji. Jenkins podporuje více agentů i agenty běžící v Dockeru. Dalším krokem, který již není zahrnut v pipeline, je nasazení aplikace do jednoho z testovacích clusterů pro interní otestování. Toto nebylo možné zautomatizovat, jelikož ani vývojář neví, na jaké

prostředí zrovna bude chtít tester aplikaci naistalovat. Tudíž tato část zůstala jako manuální krok.

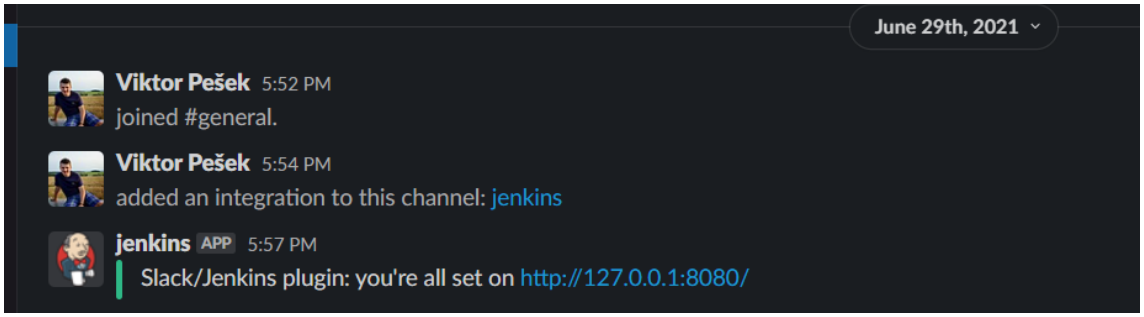


Obrázek 25 Jenkins Pipeline, zdroj: vlastní zpracování

6.4 Integrace Slack notifikací do Jenkins

Jedním z nedostatků definovaných v předchozí kapitole bylo postrádání notifikací o průběhu CI procesů. Jenkins i ostatní nástroje pro CI/CD umožňují integraci s komunikačním nástrojem Slack. Na stránce s pluginy[5] pro Jenkins je dostupný i plugin pro integraci se Slack nástrojem. Na stránce je dostupný návod, jak přidat Slack agenta v Jenkins pro posílání zpráv do Slack. Po nastavení integrace lze posílat zprávy pomocí zavolání metody ***slackSend*** v Jenkins pipeline s parametry ***channel*** a ***message***. První parametr specifikuje, do kterého kanálu/skupiny ve Slack se má zpráva poslat, a druhý parametr obsahuje tělo zprávy, která se do Slacku posílá. Níže je na obrázku ukázka notifikace z Jenkins do Slack.

[5] https://plugins.jenkins.io/slack/



Obrázek 26 Ukázka notifikace z Jenkins do Slack, zdroj: vlastní zpracování

6.5 Pořadí nasazování aplikací

Jak již bylo zmíněno v přechozí kapitole, aplikace jsou na sobě závislé. Při nasazení a startu aplikace je běžné, že si daná aplikace stahuje svou konfiguraci a další informace od jiné aplikace (existuje konfigurační mikroslužba, která toto zaštiťuje). Při nedostupnosti této aplikace první aplikace nemůže nastartovat a Kubernetes plánovač se ji bude v pravidelných intervalech pokoušet spustit dokud se mu to nepodaří nebo dokud aplikaci neuzná za chybnou a zůstane ve stavu Failed [36].

Jedním z možných řešení tohoto problému může být ošetření na úrovni nasazení. Vývojář může mít v nasazovacích skriptech definované pořadí, ve kterém lze aplikace nasazovat. Jinou možností je podle [43] použití tzv. Init Containers (inicializačních kontejnerů). Přímo v Deployment objektu Kuberenetes lze specifikovat mimo kontejnerů aplikací i inicializační kontejnery. Tyto kontejnery se spouští před samotnou aplikací. Pokud těchto kontejnerů má aplikace přiřazeno více, spouštějí se sekvenčně podle toho, v jakém pořadí jsou definovány v Deployment objektu [36]. Hlavní úkolem inicializačních kontejnerů je, jak už název napovídá, připravit Pod pro běh kontejneru aplikace. Jedním z možných použití je například stažení certifikátu, čekání na vytvoření databáze a inicializace testovacími daty a mnoho dalších [44]. Na obrázku níže je ukázka jednoduchého inicializačního kontejneru. Tento kontejner má za úkol vytvořit novou databázi pro aplikaci. Po úspěšném vytvoření nastartuje kontejner uvedený ve spec.containers s názvem app.

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  containers:
  - name: app
    image: registry.ng.bluemix.net/services/app:v1.0.1
    ports:
    - containerPort: 8080
  initContainers:
  - name: deploy
    image: appropriate/curl
    command: [ "sh" ]
    args: [ "-c", "curl -X PUT $URL/dbname" ]
```

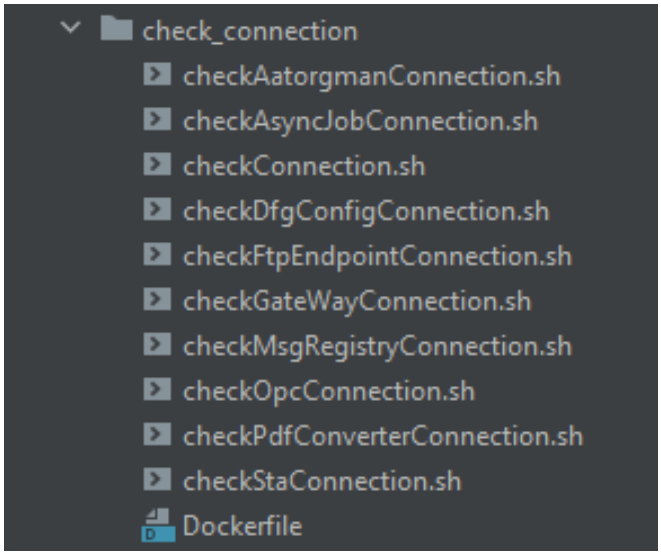
Obrázek 27 Ukázka inicializačního kontejneru, zdroj: [43]

Pro použití v této diplomové práci bude zapotřebí složitější logiky, využito bude inicializačních kontejnerů pro řešení závislostí mezi aplikacemi. Na začátku se vytvoří Ganttův diagram závislostí v čase a tento diagram bude převeden do sady skriptů psaných v Linux BASH. Skripty budou provolávat jednotlivé aplikace a shromažďovat data o dostupnosti/nedostupnosti aplikace. Pokud všechny aplikace definované v závislostech budou dostupné, inicializační kontejner bude ukončen a startovat začne hlavní kontejner aplikace. Celá tato sada skriptů bude zapouzdřena do Docker obrazu, který bude následně použit v Helm balíčcích.

	A	B	C	D	E
1	**Mirkoslužba, Pořadí nasazení**	1	2	3	4
2	MongoDB				
3	Config mikroslužba				
4	Kafka mikroslužba				
5	STA mikroslužba				
6	OPC mikroslužba				

Obrázek 28 Ganttův diagaram posloupnosti nasazení, zdroj: vlastní zpracování

Na obrázku výše je zobrazena posloupnost nasazení jednotlivých mikroslužeb, aby byly dodrženy závislosti jednotlivých komponent. Dále bude vytvořena sada BASH skriptů, které budou technicky realizovat toto schéma aplikací. Jak je vidět v obrázku, MongoDB databáze a konfigurační mikroslužba jsou společné závislosti pro obě aplikace OPC i STA. Sada skriptů se skládá z hlavního ***checkConnection.sh*** skriptu a pak ze skriptů pro jednotlivé mikroslužby. V hlavním skriptu jsou řešeny společné závislosti všech aplikací a poté jsou v switch programové konstrukci volány ostatní skripty podle toho, jakou aplikaci zrovna je potřeba kontrolovat. Společné závislosti jsou zejména databáze a Kafka message brokeři. Na obrázku níže je vidět zmíněná sada skriptů kontrolující dostupnost dalších služeb.



Obrázek 29 Skripty kontrolující dostupnost mikroslužeb, zdroj: vlastní zpracování

V ukázce níže je jeden ze specifických skriptů pro STA mikroslužbu. Ke kontrole mikroslužby se používá sys/getHealth REST volání, které je implementováno v UAF frameworku a je vytvořen přesně k tomuto účelu. Odpovědí je stav aplikace a její verze. Pomocí curl příkazu je toto API zavoláno a dokud nedostane zpět odpověď s http odezvou s kódem 200, 401 nebo 403, opakuje volání. Kód 200 znamená úspěšné zavolání a vykonání požadavku, kódy 401 a 403 poté znamenají, že uživatel volající službu nemá oprávnění k danému volání. Proto, aby bylo jasné, že služba je dostupná stačí, že volání skončí na chybě s právy. Skript je také přizpůsoben, aby zaznamenával do logu informace o aplikaci, kterou právě kontroluje a případně se kterou má nějaké problémy. V Kubernetes inicializačních kontejnerech jsou příkazy, které vedou na STDOUT, automaticky přesměrovány do logů inicializačního kontejneru a dostupné přes **kubectl** program, který slouží ke komunikaci a manipulaci s Kubernetes objekty.

```
#!/bin/bash
ENV=$1
COMMAND="sys/getHealth"
URLs=("config-svc/config/$COMMAND")

for url in "${URLs[@]}"; do
  while true; do
    STATUS=$(curl -s -o /dev/null -w "%{http_code}"
"http://$ENV-$url")
    if [ "$STATUS" -eq 200 ] || [ "$STATUS" -eq 401 ] || [
"$STATUS" -eq 403 ]; then
      echo "$(date +%FT%T) Successful connection to
http://$ENV-$url"
      break
    else
      echo "$(date +%FT%T) Waiting for http://$ENV-$url
response, getting $STATUS http code"
      sleep 2
```

```
    fi
  done
done
```

Ukázka kódu 10 Skript pro kontrolu dostupnosti aplikce, zdroj: vlastní zpracování

Jak již bylo zmíněno, celá struktura skriptů bude zabalena do Docker obrazu a poté tento Docker obraz bude sloužit jako zdroj pro inicializační kontejner při startu aplikace. V ukázce níže je uveden Dockerfile pro tuto sadu skriptů. Jednotlivé skripty jsou vloženy do /scv_scripts složky umístěné v / kořenové cestě Docker kontejneru. Jako zdrojový Docker obraz je použit alpine v poslední dostupné verzi a je nainstalována knihovna curl potřebná k fungování skriptů. Tato knihovna není dostupná v alpine Docker obraze a je nutno ji instalovat ručně. To samé je nutné provést pro Bash shell.

```
FROM alpine:latest
RUN apk update && apk add bash
RUN apk add --update curl && rm -rf /var/cache/apk/*
COPY ./*.sh ./svc_scripts/
```

Ukázka kódu 11 Dockerfile pro init kontejnery, zdroj: vlastní zpracování

Na ukázce níže je zobrazeno, jak vypadá logování v kontejneru v Podu v Kubernetes clusteru po nasazení aplikace.

```
uat -mongodb (10.43.113.249:27017) open
uat -kafka (10.43.233.46:9092) open
2022-01-17T12:29:40 Waiting for http://uat-config-svc/config/sys/getHealth response, getting 000 http code
2022-01-17T12:29:41 Successful connection to http://uat-config-svc/config/sys/getHealth
```

Ukázka kódu 12 Logování inicializačního kontejneru, zdroj: vlastní zpracování

6.6 Verzování Helm Chartů

Jak už bylo dříve představeno, aplikace je dobré verzovat podle SemVer verzovacího schématu. Podle tohoto [38] schématu, by se při každé změně v kódu

měla zvednout verze aplikace. To se týká i HelmChartů. V rámci hlavního souboru Chart.yaml jsou k dispozici atributy version a appVersion. Atribut version říká, jakou verzi má daný HelmChart. Tato verze musí být SemVer validní. Atribut appVersion říká, v jaké verzi instalujeme aplikaci uvnitř chartu. Ta může a nemusí být použita třeba jako defaultní hodnota pro tag Docker obrazu. V rámci Continuous Delivery pipeline se totiž pomocí verzí HelmChartu nasazují aplikace. Každý HelmChart je uložen v .tgz balíčku a tyto balíčky jsou verzovány. Je to z důvodu rollbacku na dřívější verzi. V případě chyby v nové verzi a neverzování helm chartů může dojít k situaci, kdy můžeme jen dopředu, ale ne zpět. V případě potřeby nasazení staré verze, je nutné reverzovat změny v HelmChartech a poté znovu spustit instalaci. Celý tento proces stojí další úsilí, kterého by nebylo třeba dodržením několika pravidel verzování. Je dobré dodržovat best practices pro vydávání softwaru a jeho verzování.

```
apiVersion: v1
appVersion: "2.0"
description: A Helm chart for Kubernetes
name: usy-sta
version: 0.3.0
```

Ukázka kódu 13 Chart.yaml, zdroj: vlastní

6.7 Přesun Docker obrazů k zákazníkovi

Existuje několik možností, jak ukládat Docker obrazy. Nejjednodušší je nahrávat je do veřejného nebo soukromého registry na hub.docker.com. Registr je hostován u společnosti Docker a tudíž odpadá nutnost řešení vlastní infrastruktury hostování.

Další možností je zaplatit si Docker Artifact Registry v MS Azure. Tým OPCSTA má v MS Azure interní Kubernetes clustery a k nim Docker registry. Do nich se poté nahrávají nové verze aplikací. K dispozici tak má každé registry cca 100 GB volného prostoru pro nahrávání Docker obrazů. Při zaplnění lze pomocí azure console registry promazat od starých Docker obrazů.

Externí prostředí jsou však z důvodu bezpečnosti odstřihnuta od internetu, nelze tudíž mít Docker obrazy v Azure, jelikož by toto řešení vyžadovalo síťový dosah do internetu. Každý cluster má tedy své interní JFrog Artifact Registry, ve

kterém má uložené Docker obrazy. Jak již bylo zmíněno v analýze stávajícího řešení, přesunutí Docker obrazů bylo děláno ručně. Přitom při dodržování sémantického verzování víme, že každá finální verze bude doručena k zákazníkovi. Proč to tedy neudělat rovnou po vytvoření Docker obrazu? V rámci vylepšení pipeline pro tvorbu verze je součástí vytvoření Docker obrazu i jeho označení (otagování) pro použití na testovacím a produkčním clusteru u zákazníka a odeslání do obou externích registrů. Tímto se docílí integrity Docker obrazů mezi prostředími. Někdy i pouhým znovu sestavením Docker obrazu se mohou obrazy lišit ve verzích závislostech.

6.8 Automatický restart aplikace po pádu

Jak bylo řečeno v kapitole 5.1.17, jedním z problémů je, že při pádu aplikace Kubernetes danou aplikaci automaticky nezkusí nastartovat. Podle [36] dokumentace ke Kubernetes lze toto řešit zavedením tzv. **Liveness Probe**. Kubelet agent ji poté používá, aby určoval, zda je daný kontejner v pořádku nebo je třeba jej restartovat. Kubelet je hlavní agent Kubernetes a běží na každém stroji (nodě). Existují tři typy Liveness Probe

1. **Kontrola příkazem** – při instalaci se specifikuje příkaz, který je každých X vteřin proveden uvnitř kontejneru a tím se zjišťuje, zda je aplikace dostupná.
2. **Použití HTTP** – u tohoto typu se využívá http GET požadavku.
3. **Použití TCP** – tento typ specifikuje TCP port, který je poté kontrolován a je vyhodnocována jeho dostupnost ze sítě.

Pro použití na OPCSTA projektu byl vybrán třetí typ, a to kontrola TCP portu. Každá aplikace má definovaný v HC port, na kterém je dostupná vnějšímu světu.

```
livenessProbe:
  tcpSocket:
  port: {{ .Values.service.internalPort }}
  initialDelaySeconds: {{ .Values.probe.initalDelaySeconds }}
  periodSeconds: {{ .Values.probe.periodSeconds }}
```

Ukázka kódu 14 Konfigurace Liveness Probe, zdroj: vlastní zpracování

Po vložení hodnot ze šablony je vygenerován soubor, který se poté nahraje do Kubernetes clusteru. Pro ukázku jsou opět vybrány jen pole, která jsou nutná pro konfiguraci Liveness Probe.

```
livenessProbe:
  tcpSocket:
    port: 8080
    initialDelaySeconds: 30
    periodSeconds: 10
```

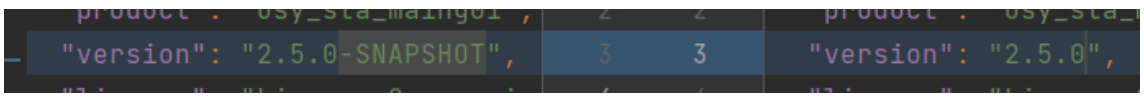
Na obrázku níže jsou vidět události pro pod aplikace OPC po nasazení do Kubernetes clusteru. Pro test bylo záměrně změněno číslo portu na 8888, které aplikace nepoužívá. Po tomto incidentu byl kontejner restartován, jak je vidět v dalších událostech.

```
Message
-------
Successfully assigned dev/dev-usy-opc-846f7759d5-j7qn5 to aks-agentpool-34442549-0
Pulling image "opcstadockerregistry.azurecr.io/dependency_check:1.0.0"
Successfully pulled image "opcstadockerregistry.azurecr.io/dependency_check:1.0.0" in 272.295806ms
Created container dependency-check
Started container dependency-check
Successfully pulled image "opcstadockerregistry.azurecr.io/usyopcstag01/usy_opc_maing01:2.3.10" in 246.611414ms
Liveness probe failed: dial tcp 10.244.1.20:8888: connect: connection refused
Container usy-opc failed liveness probe, will be restarted
Pulling image "opcstadockerregistry.azurecr.io/usyopcstag01/usy_opc_maing01:2.3.10"
Created container usy-opc
Successfully pulled image "opcstadockerregistry.azurecr.io/usyopcstag01/usy_opc_maing01:2.3.10" in 399.518072ms
Started container usy-opc
```

Obrázek 30 Ukázka události aplikace, zdroj: vlastní zpracování

6.9 Demonstrace změn v CI/CD

V následující kapitole bude představen scénář vydání nové verze servisním týmem. Bude vytvořena větev support/2.5.0 kde vývojář nastaví verzi aplikace na 2.5.0 a poté se už o nic nestará a veškerou práci s vydáním verze za něj udělá Jenkins CI server.

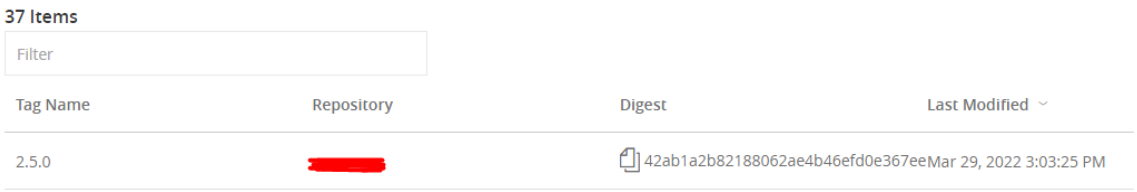


Obrázek 31 Nastavení verze aplikace na 2.5.0, zdroj: vlastní zpracování

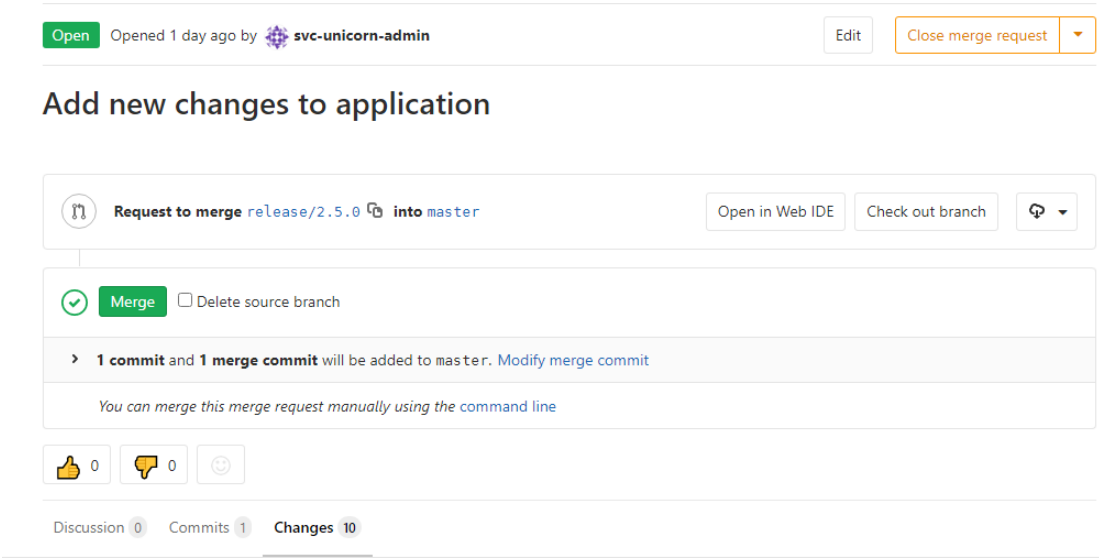
Na obrázku níže je zobrazena pipeline pro sestavení a vydání verze STA s úspěšným sestavením aplikace ve větvi support/2.5.0.

Obrázek 32 Vydání verze v Jenkins, zdroj: vlastní zpracování

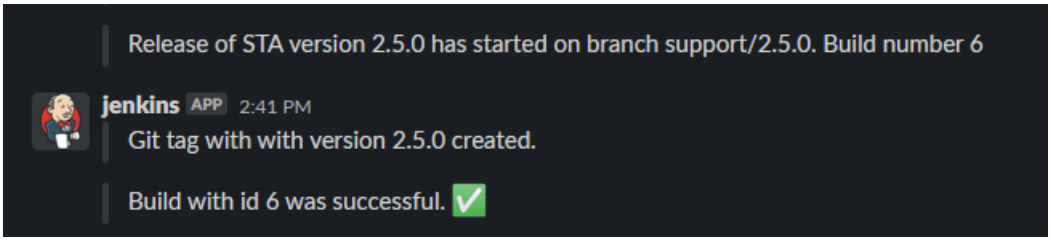
V rámci pipeline se do Docker registru u zákazníka nahraje nový Docker obraz a do GitLabu u zákazníka nahrají změny v Helm Chartu. Obrázky níže ukazují nově vytvořený Docker obraz a automaticky vytvořený požadavek o začlenění do hlavní větve. Na konci celého CI procesu je poslána notifikace do Slack.



Obrázek 33 Docker obraz u zákazníka, zdroj: vlastní zpracování



Obrázek 34 Merge request v GitLab s verzí 2.5.0, zdroj: vlastní zpracování



Obrázek 35 Slack notifikace, zdroj: vlastní zpracování

7 Závěr

Cílem diplomové práce bylo uvést čtenáře do problematiky Continuous Integration a Continuous Delivery, provést analýzu stávajícího stavu CI a CD při vývoji aplikací OPC a STA ve firmě Unicorn. Po uvedení teoretických aspektů této diplomové práce byla provedena detailní analýza procesu vývoje softwaru ve společnosti. Díky detailní analýze byly nalezeny nedostatky stávajícího řešení. Tyto nedostatky byly poté odstraněny v praktické části této diplomové práce.

Hlavním přínosem této diplomové práce bylo zavedení automatizace do procesu vývoje. Nový proces vývoje není již založen na manuálním spouštění různých dílčích procesů v Jenkins CI, jak tomu bylo u předchozího řešení. Místo toho je využito nových technologií a postupů, jak automatizovat proces integrace a dodávání nových verzí aplikací OPC a STA. V rámci změn byla představena i integrace Jenkins do Slack komunikačního nástroje, kam jsou posílány notifikace o průběhu CI procesu v Jenkins serveru. Vývojáři tedy už nemusí trávit čas spouštěním a kontrolou průběhu procesů v Jenkins, ale automaticky jsou informování o začátku, průběhu a úspěšném či neúspěšném dokončení CI procesu v Jenkins.

Díky popsaným změnám se mohou vývojáři soustředit na vytváření nových funkcionalit a celý proces CI a CD je řešen automatizovaně pouze s manuálními kroky při instalaci do vývojových prostředí v Azure a externích prostředí u zákazníka. Výsledek diplomové práce má praktické uplatnění, které bylo oceněno hlavně na daném projektu, kde přineslo výrazné ulehčení a časové úspory při dodávání nových verzí aplikací.

8 Seznam použité literatury

[1] What Is the Waterfall Model? *Project Manager* [online]. 2020 [cit. 2021-06-11]. Dostupné z: https://www.projectmanager.com/waterfall-methodology

[2] Waterfall vs. Agile: Which is the Right Development Methodology for Your Project? *Segue Technologies* [online]. 2018 [cit. 2021-06-11]. Dostupné z: https://www.seguetech.com/waterfall-vs-agile-methodology/

[3] What is a Gantt Chart? *Gantt.com* [online]. [cit. 2021-06-11]. Dostupné z: https://www.gantt.com/

[4] What is Agile? What is Scrum? *cprime* [online]. [cit. 2021-06-28]. Dostupné z: https://www.cprime.com/resources/what-is-agile-what-is-scrum/

[5] *What is Scrum?* [online]. [cit. 2022-01-13]. Dostupné z: https://www.digite.com/agile/scrum-methodology/

[6] RADIGAN, Dan. *What is kanban?* [online]. Dostupné z: https://www.atlassian.com/agile/kanban

[7] *Kanban Explained for Beginners* [online]. [cit. 2022-01-13]. Dostupné z: https://kanbanize.com/kanban-resources/getting-started/what-is-kanban#what_is_kanban

[8] What is SDLC. *Stackify* [online]. [cit. 2021-06-28]. Dostupné z: https://stackify.com/what-is-sdlc/

[9] *SDLC (Software Development Life Cycle) Phases, Process, Models* [online]. 5. leden 2022 [cit. 2022-01-13]. Dostupné z: https://www.softwaretestinghelp.com/software-development-life-cycle-sdlc/

[10] *System Development Life Cycle Guide* [online]. 21. leden 2021 [cit. 2022-01-13]. Dostupné z: https://www.clouddefense.ai/blog/system-development-life-cycle

[11] SDLC - Waterfall Model. *Tutorialspoint* [online]. [cit. 2022-01-17]. Dostupné z: https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm

[12] What is Agile SDLC and how should you use it in 2021? *Monday* [online]. 30. září 2020 [cit. 2022-01-17]. Dostupné z: https://monday.com/blog/rnd/agile-sdlc/

[13] TURNKETT, Oliver. *SDLC Methodologies: From Waterfall to Agile* [online]. 27. srpen 2020 [cit. 2022-01-14]. Dostupné z: https://www.virtasant.com/blog/sdlc-methodologies

[14] Agile Testing vs. Waterfall Testing. *EuroSTAR Huddle* [online]. 25. březen 2019 [cit. 2022-03-09]. Dostupné z: https://huddle.eurostarsoftwaretesting.com/agile-testing-vs-waterfall-testing/

[15] SHARMA, Sanjeev. *The DevOps adoption playbook: a guide to adopting DevOps in a multi-speed IT enterprise*. Indianapolis, IN: Wiley, 2017. ISBN 978-1-119-30874-4.

[16] *What is Continuous Integration?* [online]. [cit. 2022-01-19]. Dostupné z: https://aws.amazon.com/devops/continuous-integration/

[17] RITI, Pierluigi. *Pro DevOps with Google Cloud Platform: with Docker, Jenkins, and Kubernetes*. 2018. ISBN 978-1-4842-3896-7.

[18] Unit testing. *TechTarget* [online]. 1. srpen 2019 [cit. 2022-03-08]. Dostupné z: https://searchsoftwarequality.techtarget.com/definition/unit-testing

[19] ONOFRIO, Thaise a Marilene LOURENÇO. Demystifying the software engineering test pyramid. *LeadDev* [online]. 20. duben 2021 [cit. 2021-03-08]. Dostupné z: https://leaddev.com/agile-other-ways-working/demystifying-software-engineering-test-pyramid

[20] VOCKE, Ham. The Practical Test Pyramid. *MartinFowler* [online]. 26. únor 2018 [cit. 2022-03-08]. Dostupné z: https://martinfowler.com/articles/practical-test-pyramid.html

[21] MAYNARD, Claire. *Software testing for continuous delivery* [online]. [cit. 2022-03-09]. Dostupné z: https://www.atlassian.com/continuous-delivery/software-testing

[22] JUNG, June. *How to test software, part I: mocking, stubbing, and contract testing* [online]. 4. duben 2019 [cit. 2022-03-09]. Dostupné z: https://circleci.com/blog/how-to-test-software-part-i-mocking-stubbing-and-contract-testing/

[23] *What is Continuous Delivery?* [online]. [cit. 2022-03-09]. Dostupné z: https://aws.amazon.com/devops/continuous-delivery/

[24] PINNET, Sten. *Continuous integration vs. continuous delivery vs. continuous deployment* [online]. [cit. 2022-03-09]. Dostupné z: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

[25] *What is a continuous integration server?* [online]. [cit. 2022-03-19]. Dostupné z: https://about.gitlab.com/topics/ci-cd/continuous-integration-server/

[26] *Jenkins documentation* [online]. 2022 [cit. 2022-03-07]. Dostupné z: https://www.jenkins.io/

[27] *TeamCity documentation* [online]. [cit. 2022-03-19]. Dostupné z: https://www.jetbrains.com/teamcity/

[28] *CircleCi documentation* [online]. [cit. 2022-03-19]. Dostupné z: https://circleci.com/docs/

[29] *GitLab documentation* [online]. [cit. 2022-03-09]. Dostupné z: https://docs.gitlab.com/

[30] SCHULTS, Carlos. What Is Infrastructure as Code? How It Works, Best Practices, Tutorials. *Stackify* [online]. 2021 [cit. 2021-08-03]. Dostupné z: https://stackify.com/what-is-infrastructure-as-code-how-it-works-best-practices-tutorials/

[31] What is Infrastructure as Code (IaC)? *RedHat* [online]. 1. prosinec 2020 [cit. 2022-02-02]. Dostupné z: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac

[32] What is Infrastructure as Code? *Azure DevOps* [online]. 29. červen 2021 [cit. 2022-02-02]. Dostupné z: https://docs.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code

[33] Docker. *Docker Documentation* [online]. [cit. 2021-10-12]. Dostupné z: https://docs.docker.com/engine/reference/builder/

[34] *Azure Kubernetes Service* [online]. 2022 [cit. 2022-03-07]. Dostupné z: https://azure.microsoft.com/en-us/services/kubernetes-service/

[35] *Azure Container Registry* [online]. 2022 [cit. 2022-03-07]. Dostupné z: https://azure.microsoft.com/en-us/services/container-registry/#overview

[36] *Kubernetes Documentation* [online]. [cit. 2021-11-17]. Dostupné z: https://kubernetes.io/

[37] *Helm documentation* [online]. 2022 [cit. 2022-03-07]. Dostupné z: https://helm.sh/

[38] PRESTON-WERNER, Tom. *Semantic Versioning 2.0.0* [online]. [cit. 2021-11-17]. Dostupné z: https://semver.org/

[39] BONUCECLLI, Giorgio. *Easy Guide to Your Azure Subscription* [online]. 2. září 2021 [cit. 2022-03-07]. Dostupné z: https://www.parallels.com/blogs/ras/azure-subscription/

[40] CARTER, Merlin. *What's the best way to manage Helm charts?* [online]. 29. květen 2020. Dostupné z: https://insights.project-a.com/whats-the-best-way-to-manage-helm-charts-1cbf2614ec40

[41] *Learning Ansible basics* [online]. 2020 [cit. 2021-10-21]. Dostupné z: https://www.redhat.com/en/topics/automation/learning-ansible-tutorial

[42] *Several ways of kubernetes Secrets Encryption* [online]. [cit. 2021-11-17]. Dostupné z: https://developpaper.com/several-ways-of-kubernetes-secrets-encryption-idcf/

[43] LEHOTA, Ondrej. Automate the deployment of pod dependencies in Kubernetes. *IBM Developer* [online]. 2. červenec 2019 [cit. 2022-02-07]. Dostupné z: https://developer.ibm.com/articles/automating-deployment-pod-dependencies-in-kubernetes/

[44] AHMED, Mohamed. Kubernetes Patterns : The Init Container Pattern. *Magalix* [online]. 28. říjen 2019 [cit. 2022-02-07]. Dostupné z: https://www.magalix.com/blog/kubernetes-patterns-the-init-container-pattern

9 Přílohy

- Ansible-jenkins – playbook Ansible pro spuštění Jenkins CI serveru v docker kontejneru
- Dependency-check – zdrojové kódy pro inicializační kontejnery
- Helm – ukázkový Helm Chart obsahující změny inicializačních kontejnerů, SealedSecrets, livenessProbe
- Jenkins – Dockerfile s upravenou verzí Jenkins pro potřeby projektu včetně skriptů na sestavení a nahrání nového docker obrazu
- Jenkinsfile.groovy – Jenkins pipeline skript napsaný v programovacím jazyce Groovy

Oskenované zadání práce

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2019/2020

Studijní program: Aplikovaná informatika
Forma studia: Kombinovaná
Obor/kombinace: Aplikovaná informatika (ai2-k)

Podklad pro zadání DIPLOMOVÉ práce studenta

Jméno a příjmení:	**Bc. Viktor Pešek**
Osobní číslo:	**I1900548**
Adresa:	Prokopova 324, Česká Třebová – Parník, 56002 Česká Třebová 2, Česká republika
Téma práce:	Využití nástrojů CI/CD při vývoji software
Téma práce anglicky:	Using CI / CD tools in software development
Vedoucí práce:	prof. RNDr. PhDr. Antonín Slabý, CSc. Katedra informatiky a kvantitativních metod

Zásady pro vypracování:

Cíl práce:Cílem diplomové práce je navrhnout efektivní řešení CI/CD. Následně toto řešení otestovat úspěšným zavedením do procesu vývoje reálného a komplexního informačního systému.Body diplomové práce: Zavedení automatizace do procesu vývoje od implementace přes testování až po nasazení do produkčního prostředí, se zaměřením na velké projekty. Dopady na projektové řízení a metodiku vývoje. Nástroje pro automatizaci CI/CD.

Seznam doporučené literatury:

1. B. Atkinson a D. Edwards, Generic Pipelines Using Docker The DevOps Guide to Building Reusable, Platform Agnostic CI/CD Frameworks. Berkeley, CA: Apress : Imprint: Apress, 2018.
2. S. Picozzi, DevOps with OpenShift: cloud deployments made easy, First edition. Beijing: O'Reilly, 2017.
3. V. Kantsev, Implementing DevOps on AWS: bring the best out of DevOps and build, deploy, and maintain applications on AWS. Birmingham, UK: Packt Publishing, 2017.
4. P. Riti, Pro DevOps with Google Cloud Platform: with Docker, Jenkins, and Kubernetes. 2018.
5. J. Vehent, Securing DevOps: security in the Cloud. Shelter Island, New York: Manning Publications Co, 2018.
6. S. Sharma, The DevOps adoption playbook: a guide to adopting DevOps in a multi-speed IT enterprise. Indianapolis, IN: Wiley, 2017.
7. G. Kim, P. Debois, J. Willis, J. Humble, a J. Allspaw, The DevOps handbook: how to create world-class agility, reliability, & security in technology organizations, First edition. Portland, OR: IT Revolution Press, LLC, 2016.

Podpis studenta: Datum:

Podpis vedoucího práce: Datum: