ČESKÉ VYSOKÉ
UČENÍ TECHNICKÉ
V PRAZE

F3

Fakulta elektrotechnická
Katedra počítačů

Bakalářská práce

GitOps – Implementace CI/CD pipeline pro dokumentaci produktu při agilním vývoji

Richard Havel

Leden 2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Havel** Jméno: **Richard** Osobní číslo: **465852**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

GitOps – Implementace CI/CD pipeline pro dokumentaci produktu při agilním vývoji

Název bakalářské práce anglicky:

GitOps – CI/CD pipeline implementation for product documentation in the agile process

Pokyny pro vypracování:

Nastudujte metodiky GitOps a DevOps.
• Nastudujte technologie Docusaurus, GitLab CI, Docker, Kubernetes.
• Proveďte rešerši nástrojů používaných k implementaci metodiky GitOps.
• Navrhněte workflow pro tvorbu technické dokumentace k platformě CodeNOW v rámci metodiky GitOps.
• Navrhněte a implementujte pipeline průběžné integrace a nasazení metodikou GitOps.
• Celou práci realizujte agilním způsobem. Správnost dílčích řešení ověřujte v praxi a postupně je vylepšujte.

Seznam doporučené literatury:

Literatura:
Karslioglu, M. Kubernetes - A Complete DevOps Cookbook: Build and manage your applications, orchestrate containers, and deploy cloud-native services, , 2020, Packt Publishing, ISBN 9781838820336

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Komárek, kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **23.09.2020** Termín odevzdání bakalářské práce: **05.01.2021**

Platnost zadání bakalářské práce: **30.09.2022**

Ing. Martin Komárek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

Poděkování / Prohlášení

Tímto bych rád poděkoval vedoucímu své bakalářské práce Ing. Martinovi Komárkovi, za cenné rady a vstřícný přístup při psaní této práce. Také děkuji mé rodině a přátelům za podporu během celého studia.

Prohlašuji, že jsem bakalářskou práci GitOps – Implementace CI/CD pipeline pro dokumentaci produktu při agilním vývoji vypracoval samostatně a to pod vedením Ing. Martina Komárka. Dále v ní řádně cituji všechny použité prameny a literaturu.

V Praze dne 5. 1. 2021

.....

Abstrakt / Abstract

Tato bakalářská práce řeší tvorbu a nasazení dokumentace metodikou dodávky softwaru GitOps. Cílem bylo nastudovat potřebné technologie (Kubernetes, Docker a Docusaurus) a navrhnout workflow práce vícečlenného týmu pro dokumentaci agilně vyvíjené platformy, do které se zapojí pipeline kontinuální integrace a kontinuálního nasazení. Součástí práce je přehled nástrojů pro implementaci metodiky GitOps. Výsledkem práce je systém pro tvorbu a zveřejňování verzované softwarové dokumentace použitelný i pro jiné projekty. Implementace CI/CD pipeline využívá technologie GitLab CI a Argo CD.

Klíčová slova: GitOps; IaC; Infrastructure as Code; Kubernetes; Git; DevOps; CI/CD; Argo CD; Dokumentace; Kontinuální integrace; Kontinuální nasazení

This bachelor's thesis deals with the creation and deployment of documentation using the GitOps methodology of software delivery. The aim was to study the necessary technologies (Kubernetes, Docker and Docusaurus) and to design the workflow of the documentation process of a platform being developed using agile software development. The pipeline of continuous integration and continuous deployment will be involved in the workflow. The thesis contains an overview of tools for implementing the GitOps methodology. The result of the thesis is a system for creating and publishing versioned software documentation reusable for other projects. The CI/CD pipeline is implemented using GitLab CI and Argo CD.

Keywords: GitOps; IaC; Infrastructure as Code; Kubernetes; Git; DevOps; CI/CD; Argo CD; Documentation; Continuous Integration; Continuous deployment

Obsah /

Tabulky / Obrázky

Kapitola 1
Úvod

Téměr každý softwarový projekt obsahuje nějakou formu dokumentace, na které se podílí více členů týmu. Dokumentace se sdílí mezi zainteresované osoby, ať už jde o textový dokument, technickou dokumentaci automaticky generovanou z komentářů v kódu, nebo uživatelskou dokumentaci zveřejněnou na webové stránce. Všechny formy je potřeba udržovat aktuální, většinou pomoci cloudového úložiště nebo verzovacího systému Git. Práci více lidi nad Git repozitářem je potřeba korigovat a webové stránky pravidelně aktualizovat. Tady již přichází myšlenka k tvorbě dokumentace přistupovat jako k vývoji softwaru. Práce v režimu DevOps[1] je v agilním vývoji standardem. GitOps[2] je jedním ze způsobů, jak se vypořádat s DevOps v Kubernetes.

GitOps je relativně novou metodikou zpopularizovanou firmou Weaveworks. Důvodů pro výběr GitOps místo klasických DevOps je více, většina z nich vychází z toho, že se z Git stává středobodem vytváření, aktualizace a vytváření infrastruktury pro nasazení softwaru. V praxi to znamená správu například vývojářského, testovacího nebo produkčního prostředí pomocí konfiguračních souborů v Git repozitáři, kdy se na daných prostředích automaticky projeví každá změna těchto souborů. Správa infrastruktury pomocí Git umožňuje snadnou kontrolu nad změnami a jednoduchý návrat do dříve využívaného funkčního stavu. Dalšími důvody jsou dle Karslioglu[3] opakovatelnost, spolehlivost, efektivita a transparentnost.

V této bakalářské práci zapojím metodiku GitOps do workflow tvorby dokumentace platformy CodeNow[1] firmy Stratox. CodeNow je softwarová továrna[4] pro tvorbu a údržbu cloud-native aplikací. Při zapojení GitOps do tvorby dokumentace CodeNow získám vhled do základní funkcionality Kubernetes a dalších technologií použíných pro nasazení aplikací.

Zanalyzuji a upravím existující workflow tak, aby zapojení CI/CD pipeline bylo takové, aby nenastala situace, kdy ji správci dokumentace zastavují, obcházejí, nebo spouští ručně. Tedy, aby se automaticky spouštěla právě tehdy, když je potřeba a zefektivnila se tím práce na dokumentaci projektu. Dále implementuji CI/CD pipeline pro automaticé nasazení verzované dokumentace pomocí mnou vybraných technologií.

[1] `https://www.codenow.com/`

Kapitola 2
Workflow dokumentačního procesu

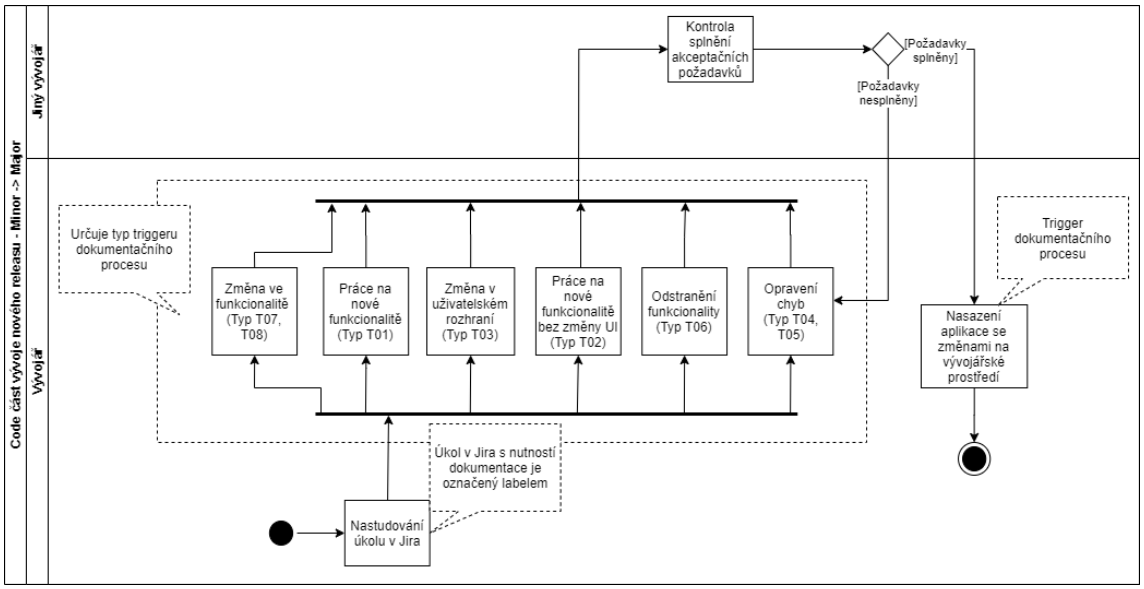
V této kapitole formuluji workflow dokumentačního procesu vycházející z existujicí workflow vývoje a dokumentace platformy CodeNow a analyzuji místa pro zapojení CI/CD pipeline. Pro potřeby návrhu workflow musím brát v potaz proces vývoje dokumentovaného kódu a naleznout změny v tomto kódu, které vyžadují úpravu dokumentace. V jednotlivých částech nového dokumentačního procesu popíšu reakci na možné změny k dokumentaci v dokumentovaném kódu od vytvoření úkolu v plánovacím systému, přes kontrolu správnosti dokumentace, po zapojení změny v dokumentaci do plánovaného vydání následující verze. Nakonec nadefinuji model větvení repositáře a zapojení CI/CD pipeline.

2.1 Workflow vývoje aplikace

Pomocí diagramů aktivit velmi zjednodušeně popisuji vývoj nové verze softwaru metodikou DevOps s použitím sémantického verzování [5]. V rámci vývoje CodeNow je v naprosté většině případů na začátku vývoje nové verze softwaru známo, zda nová verze bude major, minor, nebo patch. Z toho důvodu jsem workflow vývoje rozdělil na tři druhy dle typu příštího releasu tak, abych rozpoznal možné změny vycházející z metodiky sémantického verzování. Při čtení diagramů workflow vývoje je nutné přihlédnout k tomu, že k vlastnímu spuštění procesu tvorby dokumentace dochází při označení vývojového úkolu v Jira značkou „Document“. K tomuto označení může dojít kdykoli při plnění daného úkolu.

2.1.1 Workflow vývoje nové major verze

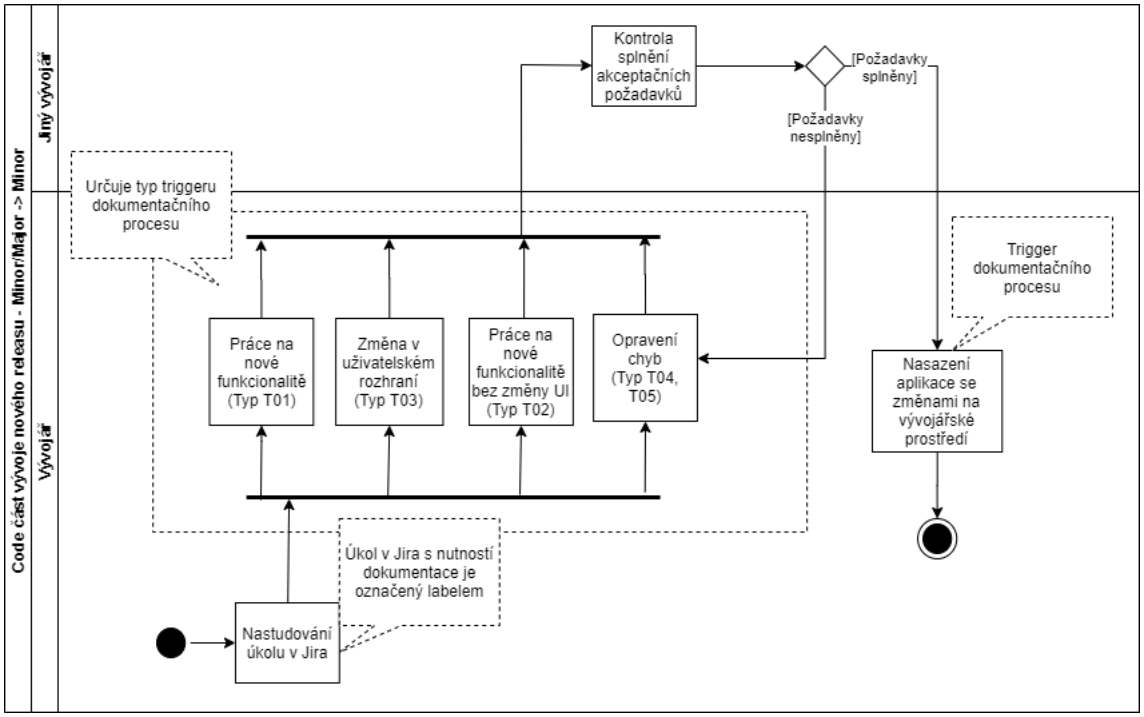
Major release může obsahovat zpětně nekompatibilní změny aplikace.



Obrázek 2.1. Zjednodušený diagram vývoje nové major verze

2.1.2 Workflow vývoje nové minor verze

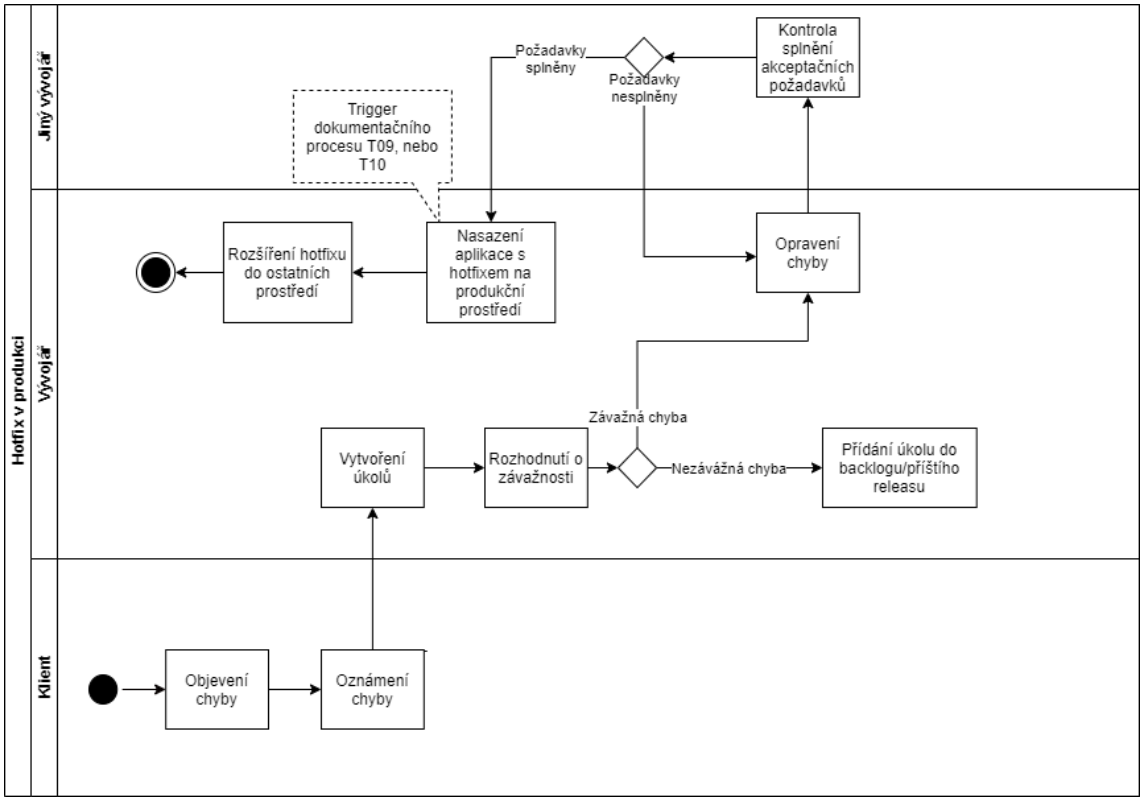
Diagram přechodu mezi minor verzemi a přechodu z major verze k nové minor verzi jsem zanesl do jednoho diagramu, protože jediným rozdílem mezi nimi může být větší množství oprav v přechodu od major verze k minor verzi.



Obrázek 2.2. Zjednodušený diagram vývoje nové minor verze

2.1.3 Workflow vývoje nové patch verze

Inkrementace patch verze obsahuje pouze opravy nevyžádaného chování aplikace.[5] To může znamenat buď plánovanou údržbu, která může být vykonávaná se všemi náležitostmi uvedenými v kapitolách 2.1.1 nebo 2.1.2, nebo může označovat neodkladné opravení chyby aplikace již nasazené v prostředí klienta, neboli hotfix. Hotfix workflow popisuje diagram níže.



Obrázek 2.3. Zjednodušný diagram vývoje hot fixu

2.1.4 Spouštěče dokumentačního procesu

Triggery dokumentačního procesu vychází z předchozích diagramů vývoje aplikace. Jediným triggerem nevycházejícím z předchozí analýzy je T11 - Bug fix v dokumentaci. Ten je očekávanou součástí života dokumentace, jsou tím myšleny objevené překlepy, nefunční odkazy, rozbité prostředí pro nasazení, atd.

Id	Název typu triggeru
T01	nová feature se změnou UI
T02	nová feature bez změny UI
T03	jenom změna UI
T04	bug fix se změnou UI
T05	bug fix bez změny UI
T06	odstranění feature
T07	změna ve feature se změnou UI
T08	změna ve feature bez změny UI
T09	hot fix se změnou UI
T10	hot fix bez změny UI
T11	bug fix v dokumentaci

Tabulka 2.1. Spouštěče dokumentačního procesu v průběhu vývoje aplikace

2.2 Návrh dokumentačního workflow

V této podkapitole prezentuji návrhy reakcí dokumentačního týmu na triggery zadefinováné v 2.1.4. Počet diagramů aktivit reakcí, přesněji diagramů workflow tvorby

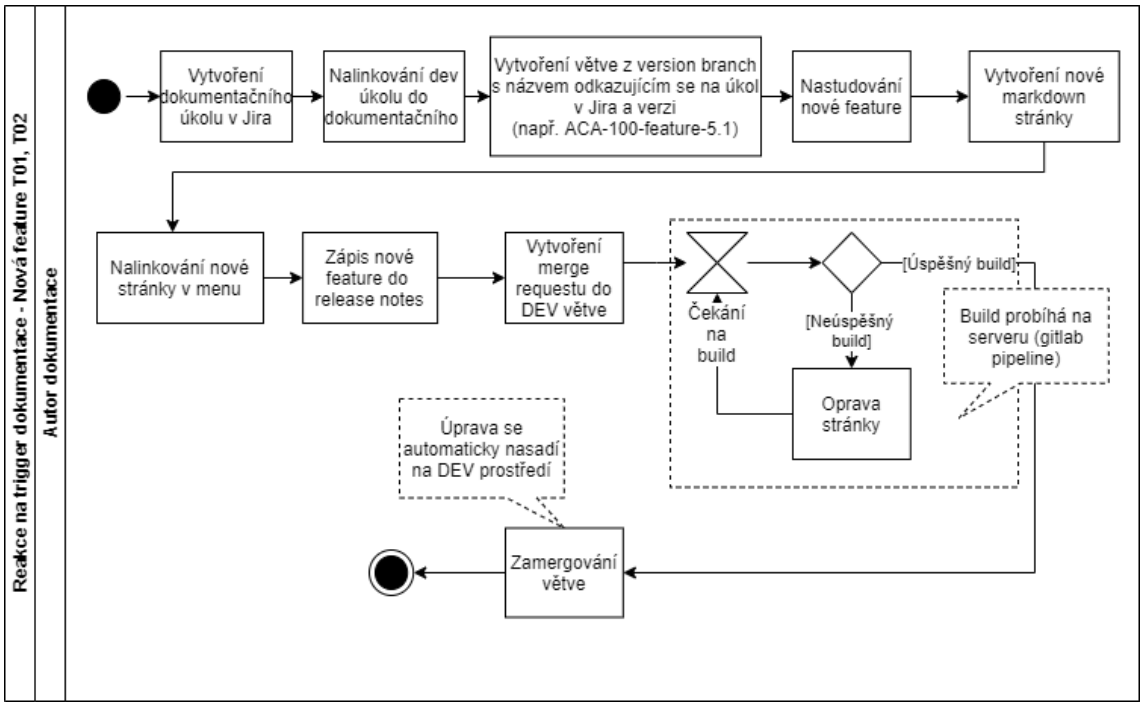
dokumentace, jsem zredukoval podle typu změn nad dokumentací. Dále popisuji procesy kontroly jednotlivých úkolů a kontrolu celých nových verzí ukončeným nasazením na prostředí pro uživatelské akceptační testování. Zároveň z návrhu workflow lze vyčíst momenty, kdy bude zapojena pipeline průběžné integrace jak samostatně, tak i s průběžným nasazením jako celá CI/CD pipeline.

2.2.1 Proces tvorby dokumentace

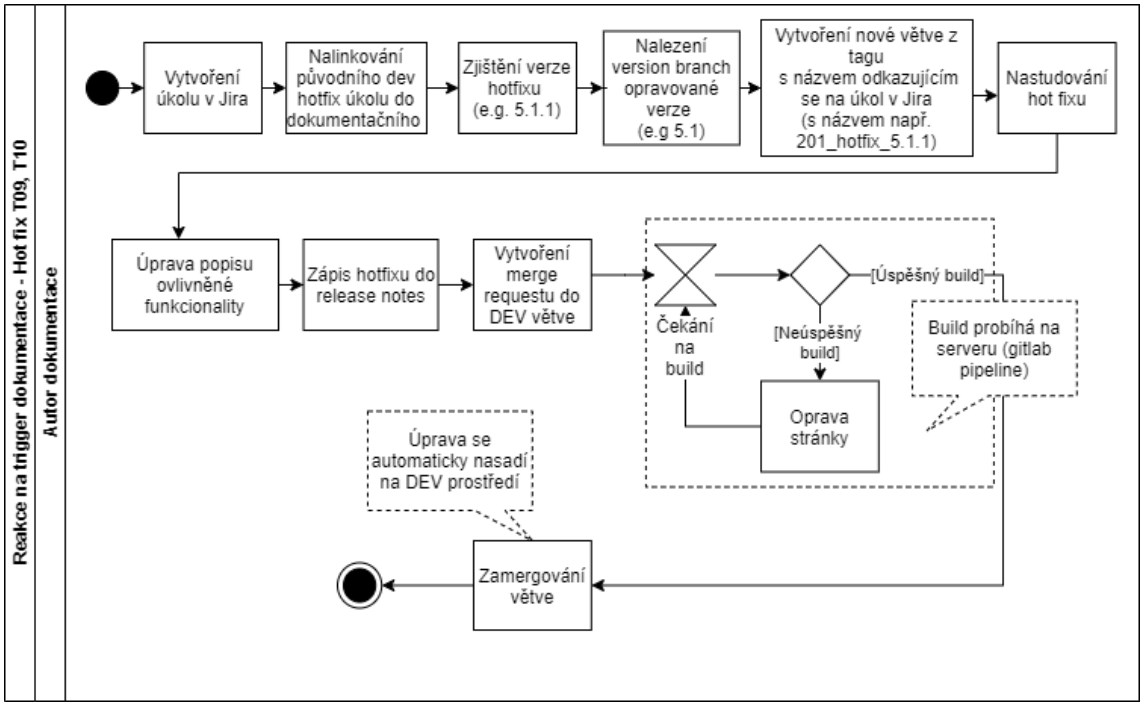
Workflow tvorby dokumentace podrobněji popsané na navržených diagramech lze shrnout těmito body:

- Vytvoření úkolu v Jira
- Vytvoření větve dodržující konvenci pojmenovávání ve verzovacím systému
- Vlastní práce na dokumentačním úkolu
- Založení merge requestu
- Nasazení na vývojové prostředí (viz. 4.1.2)

Do textu této práce vložil pouze diagramy dokumentace nové funkcionality a hot fixu, které lze vidět níže. Záměrně vynechávám návrhy pro workflow dokumentace změny funkcionality (spouštěče T03, T04, T05, T07, T08), dokumentace odstranění funkcionality (T06) a opravení chyby v dokumentaci (T11). Tyto diagramy jsou totiž až na vlastní aktivitu plnění úkolu v rámci změny v dokumentaci shodné s diagramem pro popis workflow dokumentace nové funkcionality.



Obrázek 2.4. Diagram aktivit workflow dokumentace nové funkcionality



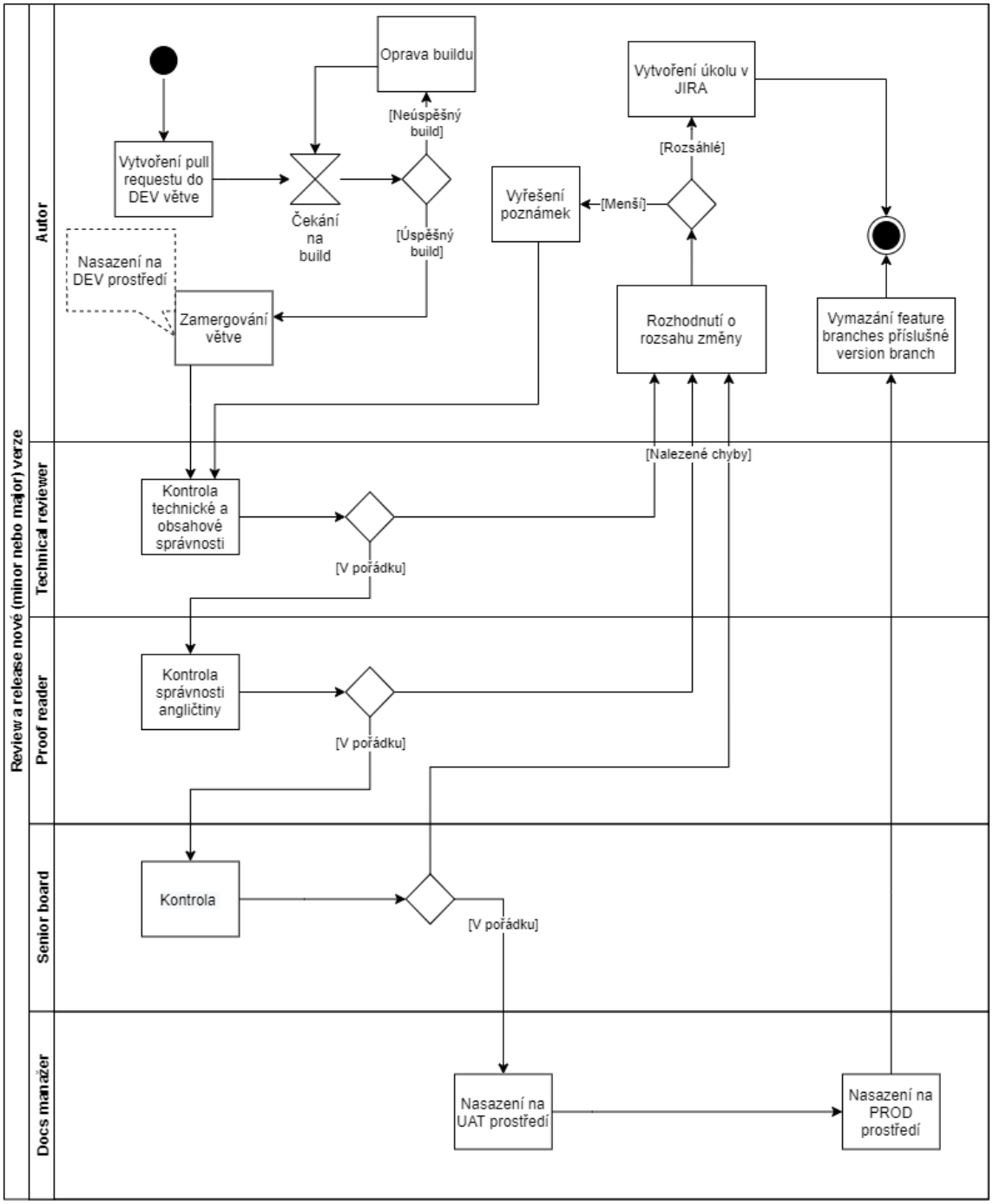
Obrázek 2.5. Diagram aktivit workflow dokumentace hot fixu

2.2.2 Proces kontroly a vydání nové verze dokumentace

Kontrolní proces změn v dokumentaci a zvěřejňování nových verzí vychází z metodiky správy dokumentace DocOps[1]. Navržený proces má až čtyři stupně kontroly podle rozsahu změn.

1. Průchod automatickým sestavením dokumentace pomocí CI pipeline (viz. 4.1.1)
2. Technické review jiným správcem dokumentace
3. Proof reading
4. Senior board assesment

Stějně jako u procesu tvorby dokumentace jsem zredukoval prezentované diagramy. Diagram níže popisující kontrolu a zveřejnení nové verze dokumentace využívá všechny čtyři stupně kontroly a tak je dobrým příkladem workflow kontroly změn v dokumentaci.



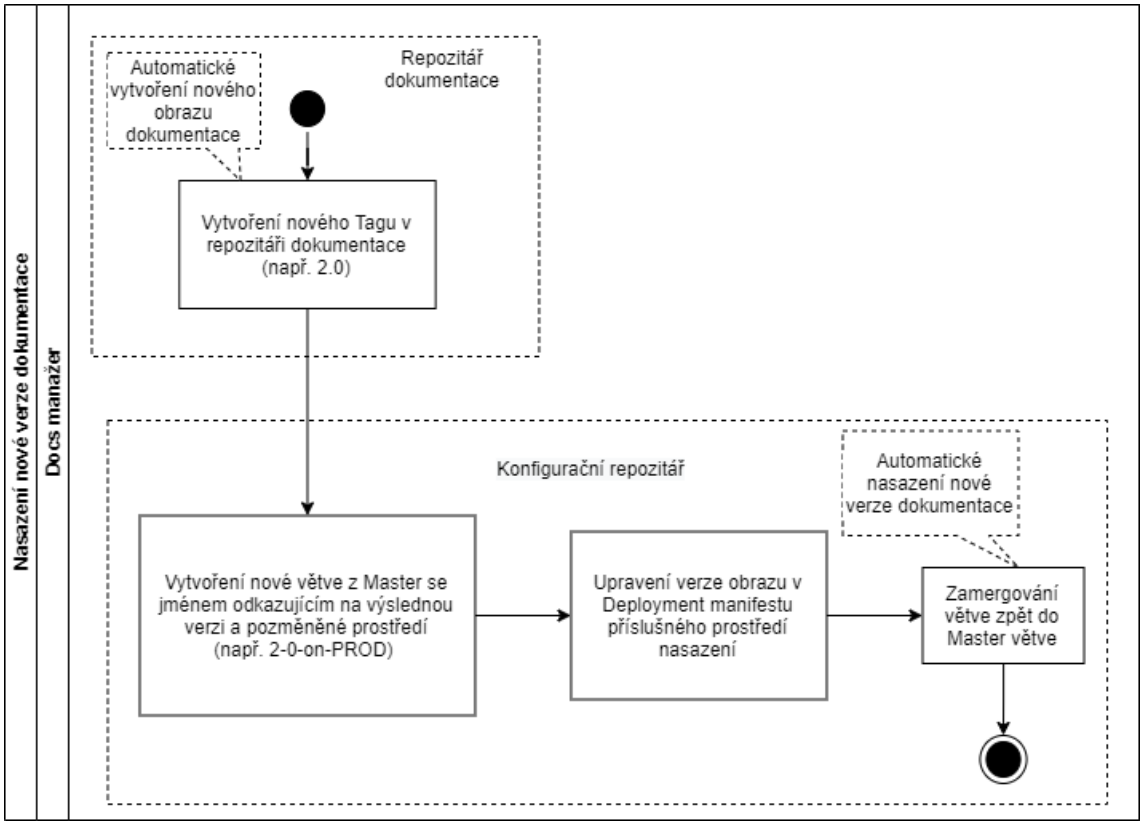
Obrázek 2.6. Diagram aktivit popisující kontrolu a zveřejnení nové verze dokumentace

Aktivity diagramy reakcí na spouštěče jsem vytvářel pro jednotlivé spouštěče změn v dokumentaci podle potřebného stupně kontroly popsaného v tabulce 2.2. Všechny změny z daných reakcí se propagují do `master` větve a tím automaticky na DEV prostředí.

Poslední částí dokumentačního cyklu je nasazení nové verze dokumentace popsané v diagramu 2.7. Nejdříve na testovací prostředí a po vydání nové verze dokumentované aplikace i na prostředí produkční.

Spouštěče změny v dokumentaci	Stupně kontroly změny
Dokumentace nové funkcionality (T01, T02)	1. až 3.
Změna funkcionality (T07, T08)	1. až 3.
Malé změny v dokumentaci (T03, T04, T05, T06, T11)	1. a 2.
Dokumentace hot fixu (T09, T10)	1. a 2.

Tabulka 2.2. Stupně kontroly změn dokumentace



Obrázek 2.7. Diagram aktivit popisující nasazení nové verze dokumentace

2.3 Model větvení Git repositáře

Branching model[6–8] jsem vytvářel s ohledem na podmínky vycházející z existujícího stavu repozitáře dokumentace:

- Kontrola změn dokumentace probíhá nad dokumentací běžící na DEV prostředí
- Branching model by měl obsahovat větve pro jednotlivé verze
- Po vydání nové verze se mažou `feature` větve jejichž změny nová verze obsahuje
- Mohou probíhat práce na více verzích současně

2.3.1 Existující modely

Existují tři hlavní modely větvení Git repositáře:

- GitHub Flow
- GitLab Flow
- GitFlow

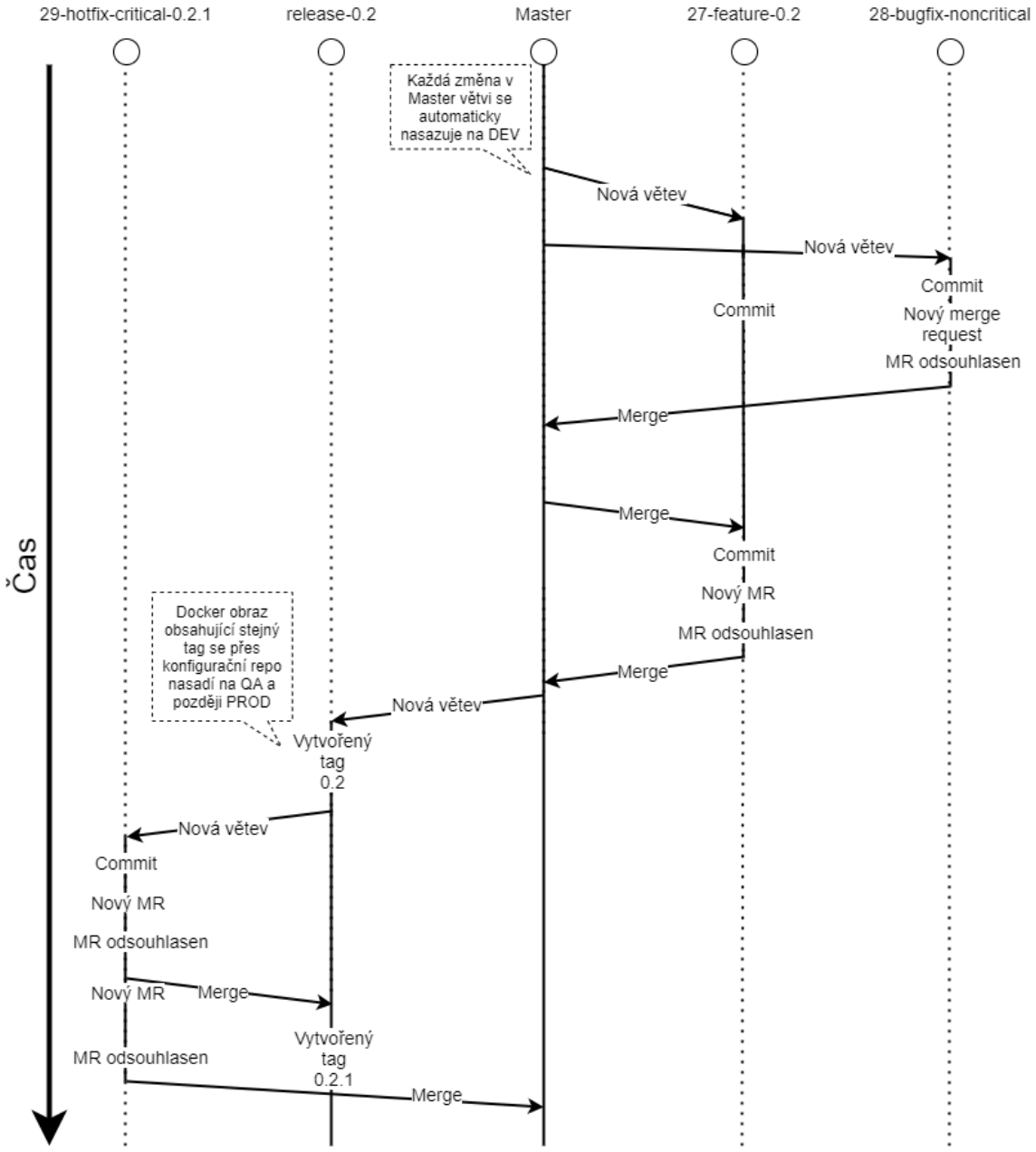
GitHub Flow je nejjednodušší z výše uvedených modelů, je jedním z nejvíce používaných. Počítá se v něm s jednou verzí aplikace, nachází se v `master` větvi. Hlavní větev by vždy měla být nasaditelná do produkce a tak si vývojář musí dávat pozor, co do ní vkládá. Podle Hogbin Westby [7] se v současnosti v GitHub Flow nebezpečí „rozbití“ produkce řeší tak, že se po schválení Merge request daná změna nasadí na testovací prostředí předtím, než se zpropaguje do `master` větve.

GitLab Flow zavádí použití větví `staging` a `production`. Do `staging` se provádí merge z `master` pro testovací účely před nasazením na produkční prostředí. Kniha Git for Teams [7] popisuje možnost využití větví pro jednotlivé verze místo jedné `production`. Zároveň Sijbrandij [9] při použití modelu s `release` větvemi nahrazuje pro nasazení na testovací prostředí větev `staging` větví `master`. Větve `release` by se měly vytvářet z `master` tehdy, kdy jsou dokončené všechny plánované změny pro plánované vydání nové verze. Do `release` se přidávají už jen bug fixy a větev tak je připravená na případné nasazení do produkce.

GitFlow model byl jeden z prvních návrhů, jak využít Git větve a Zaal [6] zmiňuje, že je vhodný pro použití ve workflow s plánovaným zveřejňováním nových verzí produktu. Kromě `feature` větví pro vývoj nové funkcionality, které obsahují všechny známé větvící modely, a `release` větví zmiňovaných výše, u GitLab Flow zavádí GitFlow větve `develop` a hotfix. Na rozdíl od GitLab Flow se na produkční prostředí nasazuje `master` po vložení změn z `release`. V `master` větvi jsou jednotlivé verze označené tagem. Centrální větev, ze které se vytvářejí všechny ostatní, je v tomto modelu `develop`. Hot fix, jakožto opravu chyby nalezené v produkčním prostředí, popisuji v 2.1.3.

2.3.2 Navržený model

Navržený model větvení Git repozitáře dokumentace vychází z GitLab Flow verzi s `release` větvemi. Odchýleními od GitLab Flow jsou přidané `hotfix` větvě a DEV prostředí. Zároveň přesun testovacího prostředí na `release` větve s tím, že nasazení nové verze na UAT a PROD prostředí probíhá v rámci repozitáře konfigurace nasazení.



Obrázek 2.8. Příklad použití navrženého modelu větvení

Kapitola 3
Výběr technologií

3.1 Git platforma

Git je hlavním zdrojem pravdy v rámci GitOps a tím se stává středobodem většiny workflow této práce. V rámci zdarma poskytovaných služeb platforem ze kterých jsem vybíral, plní požadavek na alespoň dva privátní repozitáře zdarma pro více než jednoho člověka všichni známí Git poskytovatelé. Hlavním parametrem pro výběr platformy se proto stává podpora CI pipeline pro Docker image a možnost připojení se na Kubernetes cluster v rámci CD části pipeline. Vedlejšími požadavky je možnost REST API pro připojení se do jiného repozitáře v rámci stejné platformy, kde se buď změní nějaký soubor, nebo se lépe spustí pipeline. Požadavek na REST API vychází z plánované budoucí funkcionality pipeline, viz. 5.1.

Vybíral jsem mezi Bitbucket, GitLab a GitHub, avšak záhy jsem zjistil, že mé požadavky plní všechny tři CI/CD nástroje daných poskytovatelů. Proto jsem za určující ukazatel vzal spolehlivost pro velké firmy. Tím myslím určitou vyspělost CI/CD nástrojů a možnost tyto nástroje nasadit na vlastní prostředí. Vyspělost považuji za stav, kdy je delší dobu dokončena a zdokumentována většina funkcionality. Nasazením nástroje, který spouští CI/CD pipeline, na vlastní prostředí můžeme škálováním dosáhnout rychlejšího běhu pipeline a v ideálním případě také větší spolehlivosti.

3.1.1 GitLab CI

GitLab CI se umisťuje na nejvyšších pozicích v žebříčcích CI platforem. Začátek vývoje GitLab CI byl v roce 2012 jako separátní aplikace, od roku 2015 je nástroj integrovaný do GitLab. Umožňuje běh GitLab Runner na vlastním stroji.

3.1.2 GitHub Actions

Actions jsou se svým Marketplace velmi podobné Azure DevOps, obě platformy jsou vyvíjené firmou Microsoft. Z Marketplace lze získat předvytvořené akce pro jednotlivé kroky pipeline, které se pak volají jako funkce s dosazenými vlastními parametry. Stále však narozdíl od Azure DevOps podporuje jednoduché volání vlastních příkazů. Azure DevOps umí však se svou editací pipeline primárně pomocí UI v kombinaci s Marketplace být více intuitivní. Bylo vydáno pro veřejnost na konci roku 2019 a plán do budoucna počítá s velkým množstvím změn. GitHub Actions umožňuje běh na vlastním stroji.

3.1.3 Bitbucket pipelines

S Bitbucket mám pracovní zkušenosti a v kombinaci s JIRA je výbornou platformou. Obsahuje na první pohled nejintuitivnější API pro práci s jiným repozitářem, ale jen 50 minut zdarma měsíčně na běh pipeline je však pro potřeby tohoto projektu nedostatečné. Také neumožnuje nasazení na vlastním stroji. Vydáno v roce 2016.

3.2 Kubernetes hosting

Službu na které by běžel Kubernetes cluster pro nasazení dokumentace jsem hledal z důvodu toho, abych mohl vedoucímu práce předvést demo CI/CD pipeline a abych se vyhnul prezentování interních informací firmy Stratox při obhajobě této práce. Hosting jsem vybíral takový, abych za něj ideálně po dobu mé práce na tomto projektu nemusel platit. Z tabulky níže lze vyčíst, že mé potřeby splňuje Google Cloud Platform.

Název služby	Počet měsíců pro běh projektu zdarma
Google Cloud Platform	3
Microsoft Azure	2,5
Digital Ocean	2
IBM Cloud	1
Amazon Web Services	0

Tabulka 3.1. Přehled Kubernetes hostingu zdarma

3.3 GitOps operátory

Opravdovým středobodem této práce je však GitOps operátor, používaný k dosažení stavu prostředí nasazení popsaného v Git repositáři. Porovnávám možnosti operátorů v rámci kontinuálního nasazení a požaduji co nejlepší podporu Kubernetes, pokračující podporu dané technologie jejími vývojáři a dostatečnou velikost uživatelské základny. Nástroje níže tyto požadavky splňují. Dále jsem v průběhu používaní vybraného operátoru Argo CD ocenil přehledné grafické uživatelské rozhraní. Uživatelské rozhraní považuji za dobré pro rychlou kontrolu správného běhu pipeline jakýmkoli členem týmu.

3.3.1 Argo CD

Argo CD[1] je operátorem specifickým pro Kubernetes, též formát konfiguračních souborů vycházejí z Kubernetes manifestů. Podporuje ovládání více clusterů zároveň a podporuje běh v režimu vysoké dostupnosti (viz. 3.4.1). Má jednoduché a přehledné grafické uživatelské rozhraní s funkcionalitou ekvivalentní tomu v příkazovém řádku.

3.3.2 Jenkins X

Jenkins X[2] je komplexní CI/CD platformou pro Kubernetes, zaštitující velkou část softwarového vývoje bez nutnosti zapojení dalších nástrojů. Umožňuje vše, od správy Git repozitáře, přes testování a možnosti nasazení aplikace pro náhled při merge request, po samotné nasazení na existující prostředí. Dle Yuen [10] není potřeba rozumět Kubernetes pro používání Jenkins X.

3.3.3 Flux v2

Flux v2[3] obsahuje velké množství zajímavých funkcionalit[4] přidaných k funkcím Flux v1. Bude velmi silným GitOps operátorem pro nasazení sémanticky verzovaných aplikací[5], nyní je však ve vývoji a tato funkce je v alpha stádiu. Flux nemá uživatelské rozhraní, což dává smysl, každý cluster, o který se stará, má vlastní Flux instanci.

[1] `https://argoproj.github.io/argo-cd/`
[2] `https://jenkins-x.io/`
[3] `https://toolkit.fluxcd.io/`
[4] `https://toolkit.fluxcd.io/faq/`
[5] `https://toolkit.fluxcd.io/guides/image-update/`

3.3.4 WKSctl

WKSctl[1] je jednoduchý nástroj pro vytváření Kubernetes clusterů s podporou GitOps metodiky. Obsahuje integrace pro nástroje Sealed Secrets, Flux a Weave Net. Tento nástroj sice není určený k kontinuálnímu nasazení, ale zmiňuji ho, protože v kombinaci s Flux a Sealed Secrets jde o silný GitOps systém pro lehce replikovatelné nasazení aplikací. Weaveworks stojí za WKSctl a Flux zpopularizovala termín GitOps.

3.3.5 Keptn

Keptn[2] je platformou nejen pro kontinuální nasazení pro Kubernetes s podporou GitOps, ale je komplexním DevOps systémem s možností jednoduchého zapojení služby pro monitoring jako je Prometheus Service nebo Dynatrace a nástrojů pro automatické testování již nasazených prostředků pomocí Selenium[3] a NeoLoad[4]. Též je možná propagace notifikací do vybraných kanálů jako je Slack nebo MS Teams.

3.3.6 Werf

Werf[5] je minimalistický GitOps operátor s podporou tvorby základní CI/CD pipeline. K nasazení na Kubernetes cluster využívá buď registr obrazů, nebo werf přímo tvoří obrazy podle Dockerfile předpisu v Git repositáři. Zajímavá je funkcionalita umožňující automaticky mazat nepoužívané obrazy z registru.

3.4 Instalace technologií

Nyní ukáži instalaci některých nástrojů potřebných pro vývoj. Výsledkem instalace je lokální prostředí připravené na vývoj Docusaurus dokumentace a cluster s Argo CD připraveným na nasazení této dokumentace. Já jsem si na lokální prostředí kromě potřebného nástroje Yarn nainstaloval i Docker, na kterém jsem rozběhl lokální Kubernetes cluster. Instalace a udržování tohoto clusteru byla s mým výběrem technologií špatnou volbou, vyzkoušel jsem si na něm pouze pár základních příkazů pro použití Kubernetes a Docker. Pokud bych chtěl lokálně testovat celou CI/CD pipeline, tak bych potřeboval nakonfigurovat i lokálně běžící GitLab Runner, což by nejspíše zabralo několik dní práce. Pro porovnání mi příprava Kubernetes cluster s nainstalovaným Argo CD operátorem zabrala maximálně hodinu času.

3.4.1 Argo CD

Argo CD jsem instaloval podle dokumentace[6] tohoto nástroje. Na připraveném Kubernetes clusteru (příklad pro Google Cloud Platform viz. 4.2.1) jsem vytvořil `argocd` namespace a aplikoval jsem manifest pro instalaci Argo CD přes gcloud CLI takto:

```
kubectl create namespace argocd
kubectl apply -n argocd -f $ARGO_CD_INSTALLATION
```

Kde `ARGO_CD_INSTALLATION` je tato URL[7]. Každou část Argo CD jsem nechal po jedné běžící replice, ale pro správu většího množství projektů Argo CD dle dokumentace[8]

[1] https://github.com/weaveworks/wksctl
[2] https://keptn.sh/docs/0.7.x
[3] https://www.selenium.dev/
[4] https://www.neotys.com/
[5] https://github.com/werf/werf
[6] https://argoproj.github.io/argo-cd/getting_started/
[7] https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
[8] https://argoproj.github.io/argo-cd/operator-manual/high_availability/

umožňuje pro vysokou dostupnost služeb horizontální škálování. Pokud chceme nainstalovat Argo CD v nastavení vysoké dostupnosti, tak stačí při aplikaci instalačního manifestu nahradit v odkazu `/install.yaml` za `/ha/install.yaml`[1].

Po úspěšné instalaci běží na clusteru Argo CD, ke kterému není zvenku clusteru přístup. Je proto nutné pozměnit existující Kubernetes Service `argocdserver` na LoadBalancer. Já jsem k tomu využil uživatelské prostředí Google Cloud Platform, ale je možné i následovat dokumentaci a použít tento příkaz:

```
kubectl patch svc argocd-server -n argocd  \
    -p '{"spec": {"type": "LoadBalancer"}}'
```

Následuje výpis dané služby, kde lze v poli `EXTERNAL IP` vyčíst IP adresu, kde je Argo CD přístupné:

```
kubectl --namespace=argocd get svc argocd-server
```

Výchozím uživatelským jménem k přihlášením do Argo CD je `admin`, dále stačí jen zjistit výchozí heslo k přihlášení se do Argo CD. Výchozím heslem je název Podu, kde běží `argocdserver`. K zjištění jsem použil příkaz pro výpis všech Podů:

```
kubectl --namespace=argocd get pod
```

Po přidání alespoň jednoho manifestu do konfiguračního repositáře nasazovaného projektu můžeme vytvořit Argo CD aplikaci. Já využil intuitivního uživatelské rozhraní tohoto GitOps operátoru, kde jsem nejdříve v nastavení přidal připojení k nově vytvořenému repositáři. Nakonec jsem vytvořil vlastní aplikaci, která se odkazuje na nové připojený repozitář.

3.4.2 Docusaurus

Ke správě JavaScript knihoven jsem na doporučení vedoucího práce použil nástroj yarn. Projekt dokumentace jsem inicializoval podle návodu popsaného v Docusaurus dokumentaci[2], vygerovaný projekt obsahuje Google analytics integraci, takže není potřeba tuto integraci instalovat zvlášť. Pokud to uděláme, tak Docusaurus požaduje dva Google Analytics identifikátory. Příkazy na spuštění jsou stejně jako pro Dockerfile, to částečně vysvětluje způsob vytvoření Dockerfile a dalších CI scriptů jako zápis příkazů, které používáme na spouštění projektů na lokálním prostředí do scriptu. Příkazy pro spuštění Docusaurus dokumentace:

```
yarn install
yarn run serve --build
```

[1] https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/ha/install.yaml
[2] https://v2.docusaurus.io/docs/installation

Kapitola 4
Implementace CI/CD pipeline

4.1 GitLab část

V GitLab části pipeline se v CI pipeline vytváří obraz kontejneru pro běh dokumentace, také obsahuje CD pipeline pro nasazení na DEV prostředí. GitLab spouští příkazy pomocí GitLab Runner zadefinováné v souboru `.gitlab-ci.yml` nacházejícím se v kořenové složce repozitáře.

4.1.1 Vytvoření Docker obrazu

Obraz kontejneru vytváří program pro příkazovou řádku docker, pomocí seznamu instrukcí popsaném v souboru Dockerfile. Vytvořený obraz se poté označí tagem odpovídajícím verzi dokumentace, nebo tagem `latest`. Obraz se nakonec nahraje do Docker Hub knihovny. Nahraný verzovaný obraz je připravený na průchod workflow kontroly a případné nasazení nové verze aplikace 2.2.2. Obraz s tagem `latest` se automaticky nasadí na DEV prostředí.

Soubor Dockerfile pro generování kontejneru s běžící dokumentací Docusaurus:

```
FROM node:lts

WORKDIR /app/docs

COPY ./docs /app/docs
RUN yarn install
EXPOSE 3000:3000
ENTRYPOINT ["yarn", "run", "serve", "--build"]
```

Verzovaný obraz se CI pipeline vytváří následujícími příkazy [11]:

```
build_tagged:
  only:
    - tags
  stage: build tagged
  script:
    - docker login -u ${DOCKER_USER} -p ${DOCKER_PASSWORD}
    - docker build -t richardhavel/demo:${CI_COMMIT_TAG} .
    - docker push richardhavel/demo
```

Kde uživatelské jméno a heslo pro Docker Hub jsou zadefinováné v nastavení proměnných CI/CD pipeline GitLab repositáře. Proměnná `CI_COMMIT_TAG` vychází z tagu commitu. Tento script se automaticky spouští, pokud se v repositáři vytvořil nový tag, čehož je docíleno limitací `only`. Obraz s tagem `latest` se vytváří obdobně, jen místo odkazu na proměnnou `CI_COMMIT_TAG` je napsáno `latest`. Dalším rozdílem je limitace na změny ve větvi `master`, tedy `only: master`.

4.1.2 Nasazení dokumentace na existující Kubernetes Deployment

```
deploy:
  only:
    - master
  stage: deploy
  image: dtzar/helm-kubectl
  script:
    - kubectl config set-cluster k8s --server="${SERVER}"
    - kubectl config set clusters.k8s.certificate-authority-data ${CAD}
    - kubectl config set-credentials gitlab --token="${USER_TOKEN}"
    - kubectl config set-context default --cluster=k8s --user=gitlab
    - kubectl config use-context default
    - kubectl rollout restart deployment ${DEPLOYMENT_NAME}
```

Kde jsou všechny proměnné získané v Google Cloud CLI pomocí příkazů popsaných v 4.1.3, a uložené v nastavení GitLab CI/CD. Příkaz `kubectl rollout restart` restartuje Deployment. Při restartu se stáhne aktuální obraz kontejneru, který se nasadí podle strategie RollingUpdate[12].

4.1.3 Proměnné pro GitLab CI/CD

V GitLab CI/CD se dají používat buď předdefinované proměnné[1], proměnné prostředí uložené v nastavení GitLab repositáře a samozřejmě proměnné zadefinováné přímo ve skriptu popisujícím CI/CD pipeline. Z předdefinovaných proměnných jsem využíval proměnné popisující commit, nad kterým pipeline pracuje. `CI_COMMIT_SHA` a `CI_COMMIT_TAG` jsem využil k otagování vytvářeného obrazu kontejneru.

Do nastavení repositáře jsem si uložil proměnné týkající se připojení ke Kubernetes clusteru. IP adresu clusteru pro `SERVER` jsem získal z popisu clusteru.

Pro získání certifikátu autority a uživatelského tokenu si musíme vytvořit Kubernetes ServiceAccount. Poté stačí zjistit název vytvořeného tokenu příslušícímu danému uživateli pomocí výpisu Kubernetes secrets a tento získaný název nakonec využít k opatření položky `token` v popisu tokenu a `data: ca.crt` jako certifikát autority v YAML výpisu daného tokenu [3], tedy:

```
# výpis secrets
kubectl get secrets
# popis uživatelského tokenu
kubectl describe secret gitlab-service-account-token-b54bh
# výpis tokenu ve formátu yaml s certificate authority data
kubectl get secret gitlab-service-account-token-b54bh -o yaml
```

4.2 Argo CD část

Cílem využití GitOps operátoru Argo CD je vytvoření infrastruktury pro automatické nasazení obrazů kontejnerů do Kubernetes clusteru. Konfigurační soubory pro Argo CD se nacházejí v Git repositáři nezávislém na repositáři dokumentace. Využívám soubory buď přímo pro Argo CD, nebo pro Kubernetes. Tato konfigurace popisuje v případě této CI/CD pipeline vytvoření a nastavení Kubernetes clusteru a popis prostředí nasazení spolu s definováním proměnných prostředí.

[1] `https://docs.gitlab.com/ee/ci/variables/predefined_variables.html`

4.2.1 Nastavení clusteru

V této podkapitole nabízím způsob vytvoření a nastavení Kubernetes clusteru připraveného pro DevOps na Google Cloud Platform. Výsledkem bude možnost aplikovat Kubernetes manifesty nasazení obsahující reference na obrazy uložené v Docker hub knihovně odkudkoliv, kde je nainstalovaný Kubernetes nástroj pro příkazový řádek.

Nejdříve jsem vytvořil cluster spolu s Kubernetes node, což je v uživatelském rozhraní Google Cloud Platformy vcelku intuitivní, kromě maximálního množství Podů v rámci node jsem ponechal výchozí hodnoty.

Pro potřebu tohoto projektu jsem metodou pokus-omyl zvolil počet 25 Podů na node. Na clusteru je předinstalovaných 11 Podů pro monitoring, automatické škálování a další funkcionality. Argo CD zabere podle nároků na dostupnost 5 až 10 Podů a zbývající Pody jsou volné pro nasazení libovolných aplikací.

Nyní je možné pro zavedení GitOps nainstalovat Argo CD. Dalším krokem je vytvoření secret pro připojení se k Docker hub. K trvalému připojení do Docker hub jsem použil tento příkaz, převzatý z Hands-On Microservices with Kubernetes [11]:

```
$ kubectl create secret docker-registry private-dockerhub \
  --docker-server=docker.io \
  --docker-username=g1g1 \
  --docker-password=$DOCKER_PASSWORD \
  --docker-email=$DOCKER_EMAIL
```

Dle později nalezené literatury není však bezpečné používat příkaz výše v praxi. Dle Yuen [10] bychom neměli psát citlivá data do příkazové řádky a doporučuje několik technologií pro správu secrets. Pro jednoduchost je vhodné použití Kubeseal[1].

Nakonec jsem vytvořil servisního uživatele pro připojení z GitLabu. Ten se v CD pipeline využije k restartu prostředků Kubernetes Deployment, proto je potřeba udělit uživateli pravomoce pro modifikaci těchto prostředků. K definovaní pravomocí se využívá Role, které se přidělují pomocí RoleBinding[2].

```
$ kubectl create serviceaccount gitlab-service-account
$ kubectl create role patch-role --verb=get,list,watch,patch \
  --resource=deployment.apps,deployment.extensions
$ kubectl create rolebinding gitlab-patch-binding \
  --clusterrole=patch-role \
  --user=gitlab-service-account \
  --namespace=bp-namespace
```

Servisní uživatele s pravomocemi lze spravovat jako konfigurační soubory a můžeme je v duchu GitOps uložit do repozitáře konfigurace projektu.

4.2.2 Popis nasazení dokumentace pomocí manifestu

Pro strukturalizaci nasazení aplikace jsem využil návrhový vzor aplikace aplikací[3], ten určuje hlavní aplikaci, která se skládá pouze z dalších aplikací. Původně jsem měl všechny prostředí nasazení v jedné aplikaci, ale nebylo to dost přehledné ani v konfiguračním Git repositáři, ani v uživatelském rozhraní Argo CD. Rozdělil jsem původní aplikaci na aplikaci obsahující nastaveníclusteru společné pro všechny prostředí a aplikace pro jednotlivé nasazení. Manifesty jsem rozdělil do složek a vytvořil jsem složku pro aplikace, které se na tyto složky s manifesty odkazují.

[1] https://github.com/bitnami-labs/sealed-secrets
[2] https://kubernetes.io/docs/reference/access-authn-authz/rbac/
[3] https://argoproj.github.io/argo-cd/operator-manual/cluster-bootstrapping/

Manifesty Argo CD aplikací v nasazení dokumentace se příliš neliší od manifestu hlavní aplikace níže. Všechny odkazují na stejný repositář v nastavení `repoURL` a mají stejnou `syncPolicy`[1]. Povoluji zásady synchronizace `selfHeal`, neboli požadavek na zachování a udržování stavu jaký je popsaného v Git repositáři, a `prune` povolující Argo CD operátoru odstraňovat při synchronizaci prostředky, které byli odstraněny v Git. Tím jsem docílil deklarativnosti konfigurace požadované GitOps metodikou.

Rozdíly mezi aplikacemi jsou v metadatech a ve složce s manifesty aplikace. Každá aplikace má svůj název a každá „podaplikace“ má vlastní složku. Zároveň jsem pro „podaplikace“ vytvořil namespace `bp-namespace`.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: applications
  namespace: argocd
spec:
  destination:
    namespace: argocd
    server: https://kubernetes.default.svc
  project: default
  source:
    path: applications
    repoURL: https://gitlab.com/havelric/gitops-docusaurus-config
    targetRevision: HEAD
  syncPolicy:
    automated:
      selfHeal: true
      prune: true
```

Pro popis prostředí nasazení dokumentace jsem měl na výběr několik typů Manifestů:

- Pod
- ReplicaSet
- Deployment

Od prostředí nasazení požaduji deklarativnost, možnost vložení proměnných prostředí a vytvoření více než jednoho Podu. Yuen [10] popisuje deklarativnost mimo jiné jako požadavek na to, že se změny v manifestu propíší do všech částí, které popisuje.

Pod je nejmenší jednotka v rámci Kubernetes a pro můj příklad by byla dostačující. Nemohl bych ale vyzkoušet funkcionalitu více replik a tím zvýšení dostupnosti dokumentace, takže jsem musel zvolit jinou možnost popisu nasazení, pokud bych chtěl nasadit něco, co vyžaduje neustálou dostupnost.

ReplicaSet jak je z názvu patrné, umožňuje správu více replik Podů. Yuen [10] to však nedoporučuje z důvodu špatně proveditelných aktualizaci Podů, které ReplicaSet zaštiťuje. ReplicaSet totiž zajišťuje pouze určený počet replik Podů, pokud však v manifestu například změníme verzi obrazu, tak se změna propíše jen do jednoho Podu, není tedy deklarativní. Navíc jsem nenašel zdokumentované zapojení proměnných prostředí pomocí ReplicaSet.

Deployment jako nejlepší možnost doporučuje jak Yuen [10], tak dokumentace Kubernetes[2]. Deployment umožňuje správu více instancí ReplicaSet a tím i více Podů, umožňuje zapojení proměnných prostředí a je deklarativní.

[1] `https://argoproj.github.io/argo-cd/user-guide/auto_sync/`
[2] `https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/`

Sayfan [11] jej popisuje pole po poli, popíši tedy jen pole, která nezmiňuje. Do `imagePullSecrets` se zapisuje název Kubernetes secret pro připojení ke knihovně obrazů, v tomto případě jde o Docker Hub. Vytváření Kubernetes secrets se věnuji v podkapitole o nastavení Clusteru 4.2.1. Pole `env` se využívá k deklaraci proměnných prostředí, více v 4.2.3. Níže přikládám Deployment manifest pro DEV prostředí.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: docusaurus-bp-dev
  labels:
    app: docs-dev
  namespace: bp-namespace
spec:
  replicas: 1
  selector:
    matchLabels:
      app: docs-dev
  template:
    metadata:
      labels:
        app: docs-dev
    spec:
  revisionHistoryLimit: 0
      imagePullSecrets:
      - name: regcred
      containers:
      - name: docs-dev
        image: richardhavel/demo:latest
        ports:
          - containerPort: 3000
        env:
          - name: COLOR_MODE
            valueFrom:
              configMapKeyRef:
                name: docs-configmap-dev
                key: colormode
          - name: NO_INDEX
            valueFrom:
              configMapKeyRef:
                name: docs-configmap-dev
                key: noindex
          - name: GTAG_ID
            valueFrom:
              configMapKeyRef:
                name: docs-configmap-dev
                key: gtag
```

Častou chybou při psaní manifestu Deployment, kterou jsem také udělal, je předpoklad, že tag `latest` u obrazu nebo příznak `imagePullPolicy: always` u kontejneru zajistí aktualizaci obrazu v Kubernetes Podu pokaždé, když se objeví nová verze daného obrazu s uvedeným tagem. Tento příznak přitom zajišťuje opětovné stažení obrazu při každém restartu Podu a je proto nutné tento restart při požadavku na aktualizaci zajistit. Restartu Podu lze docílit manuálně v uživatelském rozhraní Argo CD nebo Go-

ogle Cloud Platform. Lepším řešením je možné použití příkazu `rollout restart` (viz. 4.1.2). Na DEV prostředí se počítá s častými změnami nasazených obrazů, po každém novém nasazení zůstává v clusteru prázdný ReplicaSet. Proto jsem přidal parametr `revisionHistoryLimit`, jeho výchozí hodnota je deset a určuje maximální počet těchto prázdných prostředků

Pro exponování portů aplikace se používají Kubernetes Service. Sayfan [11] popisuje několik typů těchto Service. Mezi ty, které odhalují porty mimo Cluster, patří LoadBalancer a NodePort. NodePort jen odhaluje porty. Já jsem vybral LoadBalancer, který se zároveň stará o vyvažování zátěže v rámci replik aplikace. LoadBalancer můžeme vytvořit například tímto příkazem:

```
$ kubectl expose deployment docs-dev --port=80 --target-port=3000 \
      --name=docs-svc-dev --type=LoadBalancer  \
  --output="yaml" --namespace="bp-namespace"
```

Přepínač `target port` očekává port na Podu, který se exponuje a `port` značí port, který bude veřejný. Výsledný soubor není delší než příkaz, který ho vytvořil a vypadá takto:

```
apiVersion: v1
kind: Service
metadata:
  name: docs-svc-dev
  labels:
    run: docs-svc-dev
  namespace: bp-namespace
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 3000
      protocol: TCP
  selector:
    app: docs-dev
```

4.2.3 Proměnné prostředí Kubernetes

Proměnné prostředí pro nasazení dokumentace potřebuje pro předaní Google Analytics identifikátorů a HTML metaznačky `noindex` JavaScript konfiguračnímu souboru `docusaurus.config.js`. Pro ukládání proměnných prostředí se v Kubernetes používá ConfigMap. Práce s ConfigMap je popsána v Hands-On Microservices with Kubernetes [11], a to i pro složitější případy. Výsledkem je, že díky předání různých dat pro DEV a PROD prostředí docílím přesunu předávání hodnot proměnných prostředí do CD části pipeline a ušetřím tak vytváření nových obrazů pro každou změnu v rámci těchto dat. Též se tím umožňuje přepoužití jednoho obrazu kontejneru pro více prostředí nasazení. Pro DEV prostředí, pro které nepotřebuji běžící Google Analytics ani indexování vyhledávači, jsem ConfigMap vytvořil z příkazové řádky:

```
$ kubectl create configmap test \
  --from-literal=noindex='true' \
  --from-literal=gtag='default' -o yaml
```

Výsledný YAML soubor jsem přidal do konfiguračního repositáře. Dokumentace Kubernetes[1] popisuje pouze použití zapojení ConfigMap do Pod, ale na ConfigMap se lze dle

[1] `https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/`

Gupta[13] odkazovat i z manifestu Deployment. Já používám manifesty Deployment pro popis jednotlivých prostředí nasazení. Je možné ConfigMap využít k vygenerování celého souboru, ale já se kvůli malému množství proměnných rozhodl referencovat jednotlivé proměnné, které se dále mapují na již použitelné proměnné prostředí. Níže přikládám část manifestu, ve které je vidět, jak a kam správně vložit referenci na ConfigMap.

```
spec:
  template:
    spec:
      containers:
        env:
          # název proměnné, na který se lze odkazovat z kódu
          - name: NO_INDEX
            valueFrom:
              configMapKeyRef:
                # název souboru ConfigMap
                name: docs-configmap-dev
                # název proměnné v ConfigMap
                key: noindex
```

Využití proměnných prostředí v JavaScript je jednoduché, v tomto případě boolean `noindex` získám pomocí kódu `process.env.NO_INDEX || false`, kde `false` je výchozí hodnotou, pokud by referencovaná proměnná nebyla definována.

Kapitola 5
Závěr

Cílem této bakalářské práce bylo s pomocí metodiky GitOps navrhnout workflow a implementovat CI/CD pipeline pro tvorbu a hlavně nasazení dokumentace. Pro účely této práce jsem nastudoval potřebné technologie Docusaurus, GitLab CI, Docker a Kubernetes. Dále jsem navrhl workflow pro tvorbu, kontrolu a nasazení dokumentace. V přípravné části jsem vyhledal aktuálně podporované GitOps nástroje, tyto nástroje jsem porovnal a vybral Argo CD.

Výsledkem práce je snadno přepoužitelný systém, umožňující spolehlivé automatické nasazovaní, nejen softwarové verzované dokumentace na různá prostředí metodikou GitOps. V průběhu práce jsem se naučil používat Kubernetes a Docker, rád bych na to navázal dalším studiem microservice architektury tak, abych zvládl v praxi verzované microservice nasazovat a škálovat do obou směrů.

Nepodařilo se mi aplikovat systém v praxi, protože mnou řešený problém byl vyřešen firemními pracovníky dříve než jsem mohl aplikovat má řešení. Místo toho jsem svou práci testoval na nasazení textu této bakalářské práce. Čtyřikrát týdně jsem konzultoval výsledky své práce a inkrementálně jsem stavěl CI/CD pipeline od klasického DevOps nasazení na Kubernetes cluster přes nasazení pomocí GitOps metodiky až po zapojení sémantického verzování do pipeline. Též jsem takto řešil problém restartu Kubernetes Pod z CI/CD pipeline nejdříve upozorněním pověřené osoby pomocí Slack zprávy a později reálným restartem z dané pipeline. Výsledky mé práce lze nalézt na GitLab repozitáři s konfiguračními soubory[1] a repozitáři s textem této bakalářské práce[2].

5.1 Možná vylepšení

V rámci CI/CD pipeline této práce mě v průběhu implementace napadlo několik vylepšení, které jsem nestihl implementovat do termínu odevzdání práce. Do budoucna bych do CI pipeline zapojil statickou analýzu Markdown syntaxe. Také bych bezpečněji generoval a spravoval Kubernetes Secrets a rád bych do CD pipeline přidal konfiguraci pro více jak jeden cluster. Dále bych rád zjednodušil tvůrcům vydání nové verze dokumentace o jeden krok. Nyní je potřeba při vydání nové verze tuto změnu ručně propsat do konfiguračního repositáře dokumentace, což mě přivedlo na zajímavý problém komunikace mezi různými repozitáři jednoho projektu. Přesněji komunikaci mezi pipelines těchto repositářů například pomocí REST API, díky které bych spustil pipeline, která by automaticky pozměnila nasazenou verzi dokumentace.

[1] `https://gitlab.com/havelric/gitops-docusaurus-config/`
[2] `https://gitlab.com/havelric/gitops-docusaurus/`

Literatura

[1] Mike Loukides. *What is DevOps?* 2012.
http://radar.oreilly.com/2012/06/what-is-devops.html.

[2] Alexis Richardson. ■*GitOps - Operations by Pull Request*. 2017.
https://www.weave.works/blog/gitops-operations-by-pull-request.

[3] Murat Karslioglu. *Kubernetes - A Complete DevOps Cookbook: Build and manage your applications, orchestrate containers, and deploy cloud-native services*. 1. vydání. Birmingham: Packt Publishing, 2020. ISBN 9781838820336.

[4] Yaniv Sayers. *Why a software factory is key to your enterprise DevOps success*. 2019.
https://techbeacon.com/devops/why-software-factory-key-your-enterprise-devops-success.

[5] Tom Preston-Werner. *Sémantické verzování 2.0.0*. 2020.
https://semver.org/lang/cs/.

[6] Sjoukje Zaal; Stefano Demiliani; Amit Malik. *Azure DevOps Explained*. 2. vydání. Birmingham: Packt Publishing, 2020. ISBN 978-1-80056-351-3.

[7] Emma Jane Hogbin Westby. *Git for Teams: A User-Centered Approach to Creating Efficient Workflows in Git*. 1. vydání. Sebastopol: O'Reilly Media, 2015. ISBN 978-1-491-91118-1.

[8] Adam O'Grady. *GitLab Quick Start Guide: Migrate to GitLab for all your repository management solutions*. 1. vydání. Birmingham: Packt Publishing, 2018. ISBN 9781789534344.

[9] Sytse Sijbrandij. *GitLab Flow*. 2014.
https://about.gitlab.com/blog/2014/09/29/gitlab-flow/.

[10] KBilly Yuen; Alexander Matyushentsev; Todd Ekenstam; Jesse Suen. *GitOps and Kubernetes*. Version 6 vydání. Shelter Island: Manning, 2020. ISBN 9781617297274.

[11] Gigi Sayfan. *Hands-On Microservices with Kubernetes: Build, deploy, and manage scalable microservices on Kubernetes*. 1. vydání. Birmingham: Packt Publishing, 2019. ISBN 978-1-78980-546-8.

[12] Etienne Tremel. *Kubernetes deployment strategies*. 2017.
https://blog.container-solutions.com/kubernetes-deployment-strategies.

[13] Abhishek Gupta. *Learn how to configure Kubernetes apps using ConfigMap*. 2019.
https://itnext.io/learn-how-to-configure-your-kubernetes-apps-using-the-configmap-object-d8f30f99abeb.

Příloha A
Zkratky

API	Application Programming Interface
CD	Continuous Deployment
CI	Continuous Integration
CLI	Command line interface
DEV	Development
IaC	Infrastracture as Code
PROD	Production
REST	Representational State Transfer
UAT	User acceptance testing
YAML	YAML Ain't Markup Language