FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Hloubkové porovnání CI/CD systémů
Student:	Bc. Mikuláš Dítě
Vedoucí:	Ing. Tomáš Vondra, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Prozkoumejte veřejně dostupná řešení pro CI/CD (continuous integration/continuous delivery).

Popište možnosti administrační a projektové konfigurace, izolaci klientů, celkové zabezpečení, otestujte možnosti aktualizace, zhodnoťte úroveň dostupnosti systému (ve smyslu high availability) jak při běžném používání, tak i při údržbě.

Prozkoumejte dostupné instalační a testovací kroky, integrace s další typickou infrastrukturou: verzovací systémy a systémy, do kterých se aplikace nasazuje (deploy).

Navrhněte a použijte vhodnou metodiku pro porovnání CI/CD na následujících projektech:
- jednoduchá aplikace bez externích závislostí
- komplexní aplikace vyžadující relační databázi, případně další závislosti jako jsou např. fronta, key-value storage, …
- aplikace distribuovaná jako kontejner

Na základě předchozího porovnání navrhněte řešení CI/CD pro webovou agenturu. Hlavním požadavkem je praktičnost realizace. Seznamte se s typickým vývojem projektů dané firmy a při návrhu ho respektujte.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 22. prosince 2018

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra softwarového inženýrství

Diplomová práce

Hloubkové porovnání CI/CD systémů

Bc. Mikuláš Dítě

Vedoucí práce: Ing. Tomáš Vondra, Ph.D.

21. dubna, 2019

Poděkování

Děkuji vedoucímu diplomové práce Ing. Tomáši Vondrovi, Ph.D. za podnětné připomínky při vymýšlení zadání a užitečné rady při vypracovávání práce. Děkuji přítelkyni a rodině za podporu během celého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

Praha dne 21. dubna, 2019 ……

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

DÍTĚ, Mikuláš. *Hloubkové porovnání CI/CD systémů.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019. Dostupný také z WWW: ⟨`https://github.com/mikulas/thesis`⟩.

Abstrakt

Tato práce obsahuje porovnání vybraných nástrojů pro průběžnou integraci (*continuous integration*, CI) a průběžné nasazování (*continuous deployment*, CD). Byla připravena metodika a podle ní zhodnoceny nástroje *GitLab, Jenkins, GoCD, Drone, Concourse, CircleCI, Travis CI* a *Semaphore CI*. Každý z těchto systémů byl použit pro nasazení tří rozdílných webových aplikací: statická, komplexní s externími závislostmi a kontejnerizovaná. V porovnání byl kladen důraz na funkcionalitu, dostupnost, bezpečnost a rozšiřitelnost. V závěru práce je zdokumentován výběr *GitLab* jako nejvhodnějšího nástroje a následné nasazení CI/CD ve firmě *manGoweb*.

Klíčová slova cloud, DevOps, porovnání CI/CD

Abstract

This work compares publicly available *continuous integration* (CI) and *continuous deployment* (CD) systems. Evaluated systems include *GitLab, Jenkins, GoCD, Drone, Concourse, CircleCI, Travis CI* and *Semaphore CI*. Each of those was used to deploy all three of those different web applications: static, complex app with external dependencies and containerized. The comparison is primarily based on functionality, security, availability and extendibility. In the second part of the work, *GitLab* is recognized as a most appropriate tool and itś deployed at *manGoweb* web studio.

Keywords cloud, DevOps, CI/CD comparison

Obsah

Úvod

Proces vývoje software, při kterém změny všech vývojářů začleňujeme do společného kódu co nejrychleji – continuous integration, CI – je jedna z rozšířených praktik extrémního programování a zapadá i do smýšlení DevOps a SRE [1]. Ačkoliv to není nezbytně nutné [2], proces začleňování změn je zpravidla podporován nějakým integračním serverem [3]. Tato práce je se věnuje důkladnému porovnání těchto CI/CD nástrojů: *GitLab, Jenkins, GoCD, Drone, Concourse, CircleCI, Travis CI* a *Semaphore CI*.

Struktura práce

Práce je rozdělena na tři kapitoly. První z nich se věnuje definici a upřesnění termínů používaných ve zbytku práce.

V druhé kapitole je definována metodika výběru porovnávaných CI/CD nástrojů a v podkapitole je každý systém zhodnocen. V závěru této druhé kapitoly je shrnutí výhod a nevýhod otestovaných nástrojů. Podkapitoly o jednotlivých systémech jsou seřazeny a v pozdějších částech se odkazuji na již otestované nástroje. Přesto by každá část měla být čitelná sama o sobě; při výběru vhodného CI/CD dokonce doporučuji nejprve nahlédnout na závěrečné shrnutí (strana 68), vybrat systémy odpovídající Vašemu použití a následně číst příslušné podkapitoly.

V poslední třetí kapitole je zdůvodněn výběr nástroje *GitLab* jako nejvhodnějšího pro firmu *manGoweb* a je zdokumentováno nasazení tohoto systému.

Kapitola 1

Definice pojmů

V této kapitole upřesňuji termíny použité ve zbytku práce. Pro nadměrně používané pojmy s nejasným významem vyberu jednu konkrétní definici (*cloud*, *continuous deployment*). Popíšu některé technologie používané porovnávanými CI nástroji (kontejnerizace, SAST/DAST). V rámci úvodu do problematiky CI/CD uvedu možnosti nahrání aplikace na web server. Uvádím také definici dostupnosti webové aplikace, na kterou se později budu odkazovat při testování CI/CD nástrojů.

Dostupnost aplikací

Fishman (Sun Microsystems Inc.) definuje dostupnost jako vztah mezi střední čas mezi výpadky (MTBI/MTBF) a dobou zotavení z výpadku (MTTR) [4]. Dostupnost se často vulgarizuje na „počet devítek“ (např. 99,999 % dostupnost odpovídá pěti devítkám, což je kolem 5 minut nedostupnosti ročně). Cílem je teoretickou dostupnost maximalizovat. B. Wilson z firmy Backblaze trefně dodává, že od několikáté devítky jsou všechny odhady čistě akademické. Aplikace s větší pravděpodobností nebude fungovat kvůli ozbrojenému konfliktu [5].

Aplikaci lze bez výpadku provozovat i v jedné instanci na jednom fyzickém serveru. Příkladem takového nasazení je server Stratus s redundantním hardwarem, který měl v jednom případě rekordní nepřetržitý provoz 24 let [6]. Celý systém běží na proprietárním operačním systému, který od let 2000 nebyl aktualizovaný. Tyto servery jsou ale fyzicky na jednom místě a při přírodní pohromě nebo delším výpadku elektřiny přestanou být dostupné.

Praktičtější a levnější řešení je aplikaci provozovat distribuovaně, v několika fyzicky vzdálených datacentrech. V případě výpadku části systému se uživatelské požadavky plynule rozdělí mezi zbylé funkční systémy. Pro uživatele je tato operace transparentní a neprojeví se zhoršením dostupnosti a v ideálním případě ani uživateli pozorované kvality. Provozovat a později aktualizovat distribuovanou aplikaci je relativně složité a více to popisuji v sekci 1.0.3. Získáme ale horizontální škálovatelnost a tím vyšší teoretickou dostupnost.

Při testování CI/CD systémů se budu zajímat o několik různých dostupností:

- Dostupnost CI/CD systémů při běžném provozu;
- Dostupnost CI/CD systémů při správě, typicky při aktualizaci na novější verzi;
- Podpora nasazení uživatelské aplikace v daném CI/CD a dopad na dostupnost dané aplikace.

Žádný z velkých poskytovatelů nenabízí SLA pro jednu konkrétní instanci (virtuální stroj). Definují pouze 99,99 % dostupnost na celý region (AWS [7], GCP [8], Azure [9]). V rámci regionu může bez vlivu na SLA vypadnout několik datacenter (*availability zones*). Při porušení SLA vrací poskytovatelé na požádání část kreditů. Neexistuje tak prakticky žádná garance dostupnosti a aplikace musí HA zajistit distribuováním mezi datacentra.

Testování aplikací

Ve vhodně zvoleném CI nástroji je možné aplikaci testovat na všech úrovních: od jednotkových (*unit*) přes integrační až po funkcionální testy. Jednotkové testy v kontextu webových aplikací se uvažují maximálně o velikosti jedné stránky [10]. V případě dostatečné dekompozice (mj. za správného použití návrhu MVC) budou testované jednotky především v modelové vrstvě (byznys logice). Takové testy budou využívat pouze zkompilovanou aplikaci a případně runtime a závislosti budou simulovat atrapami (*Mock objects*) [11]. Díky tomu jsou tyto testy jednoduše spustitelné na CI.

Integrační testy se spouští nad moduly otestovanými jednotkově [12] a ověřují, že výsledné propojení komponent je funkční [13]. Tyto testy mohou využívat externí závislosti, například databáze.

Nejucelenějším ověřením správnosti aplikace jsou funkcionální testy (také *end-to-end* testy, e2e). Při nich se aplikace spustí s externími závislostmi, simulují se uživatelské akce a monitorují se odpovědi aplikace. Ve webovém prostředí to může být volání aplikace nástrojem `curl`, dedikovaný testovací nástroj Selenium, nebo moderní knihovna Puppeteer pro prohlížeč Google Chrome. Co se týče nasazení na CI jsou funkcionální testy nejkomplikovanější, protože obsahují řadu závislostí. Cloudové služby můžou jejich nasazení zjednodušit [14].

Testování bezpečnosti aplikací zmíním níže v sekci o SAST a DAST 1.0.5.

Cloud

Na cloud se budu odkazovat v několika rovinách: jako platforma pro hostování CI, spouštění jobů a hostování samotných aplikací, což má význam pro nasazení a tedy CD.

Velkým cloudem primárně uvažuji velké organizace, které mají desítky tisíc serverů v jednom regionu [15]. Kromě hlavní trojice AWS, GCP a Azure můžeme uvažovat i DigitalOcean, IBM Cloud, Linode a celou řadu dalších globálních poskytovatelů. Kromě virtuálních serverů patří typicky do nabídky služeb managed varianty stavových aplikací jako jsou databáze, kontrolní roviny orchestrátorů, síťové disky a další. Drobnější a regionální poskytovatele cloudu, stejně tak jako on-premise cloudová řešení (OpenStack), mohou poskytovat stejně kvalitní funkce jako velký cloud. U menších cloudů ale bývá jiný model pro platby nevhodný pro CI servery a z tohoto pohledu jsou ekvivalentní jako klasické virtuální stroje nebo bare metal. Z českých hostingů nabízí „on-demand“ virtuální servery například Master Internet [16], ale jde o jiný model než velký cloud: servery se rezervují na měsíc a platí se za využitý výkon nad nějakou hranici. U jiných českých hostingových firem jsem jiné než fixní měsíční poplatky za server nenašel.

Speciálně cenou je cloud pro CI vhodný. Jelikož CI má nárazové využití a to často jenom přes pracovní hodiny, je u dedikovaných serverů špatná utilizace. V cloudu je možné platit pouze za využité zdroje a zapínat je dynamicky. Při porovnávání CI nástrojů budu zjišťovat, jestli a do jaké míry podporují nasazení v cloudu; konkrétně se zaměřím na to, jestli je možná integrace s databází spravovanou poskytovatelem a napojení na objektové úložiště.

Nasazení aplikace

V této sekci popisuji varianty nasazení nové verze aplikace s vysokou dostupností. U porovnávaných CI/CD systémů budu později zkoumat, jaké varianty nasazení podporují, případně jak obtížné je jednotlivé varianty implementovat.

Konceptuálně je nasazení webového software přímočaré: vystavěný balíček aplikace se nějakým způsobem nahraje na jeden nebo více serverů a následně se spustí. Detaily se liší podle použitého prostředí. V následujícím textu jsem se zaměřil na monolitické webové aplikace, ale podobné principy fungují i pro nasazení dílčích podpůrných služeb (*microservices*), které mimo HTTP rozhraní mohou nabízet například gRPC, Java RMI, nebo jakékoliv jiné textové nebo binární protokoly.

Komplexita nasazení drasticky narůstá, pokud vyžadujeme kontinuální dostupnost aplikace [1]. V ideálním případě by aplikace měla při nasazování změn úspěšně obsloužit všechny příchozí požadavky. Pokud neuvažujeme continuous deployment a nasazování probíhá výjimečně, může nasazení nové verze spočívat v zastavení původní verze, výpadku, a nasazení nové verze. Toto dodnes některé firmy praktikují. Například všichni tři velcí čeští telefonní operátoři měli v roce 2018 minimálně jednu několikahodinovou odstávku webových služeb (T-Mobile [17], O2 [18], Vodafone 2. května 2018 a tiskovou zprávu z webu smazal). V následujícím textu se zaměřím pouze na nasazování bez výpadku dostupnosti aplikace.

1.0.1 Hot swapping

Jedna varianta kontinuální dostupnosti je hot-swapping, při kterém se aplikace může upravit bez nutnosti ji vypnout. Toto je ale běžně dostupné pouze v prostředí s JVM a dále například v – na webu málo používaných – Lisp, Erlang, Smalltalk a Elm. Při použití hot-swapping může aplikace běžet pouze v jedné instanci a nasazení nové verze aplikace tak můžeme udělat atomicky. Tím se vyhneme celé řadě problémů popsaných níže, na druhou stranu ale výrazně zhoršujeme praktickou dostupnost: server na kterém aplikace běží musí být vždy dostupný.

1.0.2 Přeliv provozu mezi starou a novou aplikací

Alternativní řešení je spustit novou verzi aplikace souběžně s původní verzí a následně přesunout provoz ze staré instance na novou. V případě PHP-FPM při zapnuté OPcache

můžeme například přepsat soubory na disku a novou verzi aplikace nasadit smazáním OPcache (což se dělá pomocí rolling update PHP-FPM workerů). Ve světě Node.js lze využít PM2, který abstrahuje spouštění a vypínání procesů aplikace, v Python prostředí existuje `pman`. Všechny tyto systémy pracují na principu load balanceru (LB): uživatelské požadavky chodí na jeden vedoucí proces, který si ukládá informace o běžících aplikacích na pozadí a požadavky na ně přeposílá. Tento princip je implementovatelný pro libovolnou aplikaci. Pro load balancing můžeme použít celou řadu software: PFSense, HAProxy, Nginx, Varnish, v pokročilejších infrastrukturách Envoy *service mesh*, … Důležité rozhodnutí při výběru LB je balancovaná komunikační vrstva. Layer 4 (transportní vrtva OSI modelu) LB přeposílají TCP požadavky na základě cílové IP adresy a portu. Je definován i Layer 3 LB, který se rozhoduje pouze podle cílové IP, ale v literatuře se zahrnuje do L4. Layer 7 (aplikační vrstva) LB přeposílají celé HTTP požadavky. Z vyššího protokolu vyčtou víc informací, například doménu nebo jiné hlavičky, a při přeposílání požadavku tak mohou vybrat ze celé skupiny různých serverů ty, kde daná aplikace běží. Dále L7 LB musí řešit HTTPS (TLS termination) a musí tak dopředu vědět, jaké domény obsluhuje. L7 LB dále může reagovat i na získané HTTP odpovědi. Díky HTTP specifikaci idempotentních metod (GET, HEAD, PUT, DELETE [19]) lze některé požadavky zkusit znovu na jiném serveru a to bez toho, aby se původní chyba vrátila uživateli. Load balancer může u každého serveru na pozadí měřit chybovost a rozbité servery z rotace dočasně vyřadit [20].

Jak L4 tak L7 LB často podporují *PROXY protocol* [21], který požadavek obalí vlastní hlavičkou, které je mj. původní IP adresa uživatele. V případě čistého L7 load balancování lze původní IP vyčíst z problematické [22] hlavičky `X-Forwarded-For` nebo z `Forwarded` rozšíření [23]. U čistého L4 load balancování o původní IP přijdeme bez PROXY protokolu úplně.

Je nutné uvažovat konzistenci napříč několika příchozími HTTP požadavky: uživatel může načíst HTML z původní verze, ale následující HTTP požadavek na kaskádové styly už může přijít na novou verzi aplikace.

Tento postup můžeme aplikovat i na aplikace běžící ve víc než jedné replice. Spustíme novou verzi aplikace souběžně se starou verzí a atomicky aktualizujeme na load balanceru cíle na novou skupinu serverů. Stále ale nemusíme řešit většinu problémů, které by přinesl distribuovaný systém.

1.0.3 Distribuovaná aplikace

Postup z předchozí sekce vyžaduje atomickou změnu konfigurace load balanceru. Té je těžké docílit, pokud máme více než jednu instanci load balanceru[†] . Další nevýhodou je, že při nasazování nové verze aplikace se po nějakou dobu zdvojnásobí nutné výpočetní zdroje (musí běžet současně n starých a n nových instancí aplikace). Při continuous deployment to může být velmi často, což se prodraží na potřebné infrastruktuře. Praktický problém tohoto řešení je i samotná jednorázovost přelivu uživatelů: pokud je část aplikace rozbitá, je rozbitá pro 100 % uživatelů.

Vhodnější řešení je tzv. rolling update, kdy souběžně nějakou dobu provozujeme novou i starou verzi aplikace v narůstajícím poměru. Průběžně monitorujeme aplikační metriky a v případě, že se nová verze aplikace nechová podle očekávaní, můžeme nasazování pozastavit nebo úplně vrátit. Pracujeme tedy s následujícími předpoklady:

- Aplikace zároveň běží ve staré i nové verzi. Uživatelé mohou být obsluhováni novou a následně starou verzí aplikace. Můžeme na úrovni LB částečně zajistit, aby uživatelé v po sobě jdoucích požadavcích byly nasměrování na stejnou verzi aplikace (*session affinity, session persistence*), ale není to spolehlivé a 100 % řešení. Používá se buď sledovací cookie nebo IP adresa a oboje může uživatel upravit. Kromě toho chceme umožnit tzv. *rollback*, při kterém se aplikace downgraduje na předchozí verzi.
- Rozdělení požadavků mezi aplikace řeší několik load balancerů. Máme vysokou dostupnost, ale ztratili jsme možnost atomických úprav konfigurace.

Můžeme libovolně nastavit nejenom poměr aplikací, ale i jejich absolutní počty. Pokud snížíme teoretickou dostupnost, můžeme z n starých verzí aplikací vypnout $n - 1$. Jedna instance staré aplikace stále běží a obsluhuje 100 % požadavků, nenastal výpadek. Následně zapneme $n - 1$ nových instancí. Tím jsme minimalizovali počet nutných výpočetních zdrojů. Pokud ale máme k dispozici volné zdroje, můžeme nechat běžících n původních instancí a nasadit m nových. Například v *Kubernetes* je tato

[†] Pokud si pod pojmem *atomická změna* představíme, že po žádném požadavku zpracovávaného novou verzí konfigurace nesmí být žádný požadavek zpracováván starou konfigurací, je možných několik implementací. Jedna z nich je vyřazení všech kromě jednoho z load balancerů, což nemusí byt triviální a rychlé, pokud jsou load balancery první síťový prvek v naší infrastruktuře a nemůžeme je například snadno vyřadit z DNS. Další možností je změny udělat dvoukrokově: nejprve všechny kromě jednoho primárního load balanceru překonfigurovat tak, aby přeposílali všechny požadavky na tento primární load balancer. Potom lze atomicky změnit konfiguraci nejprve primárního a následně všech ostatních load balancerů.

logika abstrahována do nastavení očekávaného počtu instancí, minimálního počtu dostupných instancí a maximálního nárůstu. Při nasazování se pak postupně vypínají staré aplikace, čeká se na zapnutí nových a tak dokola, než jsou všechny instance nové aplikace. Z tohoto procesu pochází název *rolling update*.

Za těchto podmínek nemůžeme nasazovat libovolnou aplikaci. Obecně musí být aplikace zpětně nebo dopředně kompatibilní. To komplikuje databázové migrace, nebo jinou změnu formátu na sdílených (především persistentních) úložištích. Některé změny je tak nutné nasazovat dvoufázově: první nasadíme úpravu, která zajistí dopřednou kompatibilitu, a poté samotné (původně nekompatibilní) změny.

Definice CI/CD

1.0.4 Continuous integration, CI

Nejstarší zmínka o CI je projekt *Infuse* z roku 1989 [24]. Jde o návrh systému testování komplexních modulárních projektů, který podle hierarchie spouští postupně jednotkové, integrační a akceptační testy.

Fowler v článku z roku 2000 [25] definuje CI jako praktiku, při které vývojáři změny začleňují do sdíleného kódu jednou denně nebo častěji. Logická funkčnost (ale ne nutně úplnost byznysových funkcí) je pak po každé změně otestována automatickými testy. Pokud testy selžou, vývojář jehož změny zapříčinily při testování problémy je informován a očekává se, že chyby opraví.

V aktualizovaném článku z roku 2006 Fowler [3] zmiňuje užitečnost CI serveru jako volitelnou – ale praktickou – podporu této praktiky. Na vhodný výběr CI serveru se soustředí tato práce.

Pro potřeby této práce je CI server nástroj, který od vývojářů přijímá zdrojový kód, rozhodne jestli je akceptovatelný a vrátí booleovskou odpověď. Je vhodné – ale ne nutné – aby CI server byl úzce integrovaný s nástrojem na správu repozitářů, kde po testování můžou být otestované změny přímo zařazeny do společného kódu.

1.0.5 Continuous delivery vs Continuous deployment, CD

Přestože se pojmy *delivery* a *deployment* zkratce CD zaměňují, mají jeden podstatný rozdíl. Při continuous delivery je kód stále udržován v nasaditelném stavu, ale samotné spouštění nasazení je manuální [26]. Continuous deployment je rozšíření této praktiky, které nasazuje veškerý integrovaný kód automaticky [27].

Chad Wathington vymezuje continuous integration jako podmnožinu CD [28].

Kontejnerizace

V sekci 1.0.2 o nasazení aplikace v jedné instanci zmiňuji nutnost spustit novou a starou verzi aplikace souběžně. V kontextu HTTP aplikací to může být problém, protože stará verze aplikace blokuje systémový port a nová verze aplikace se kvůli tomu nemůže na stejném portu zapnout. Jedno z řešení je nastavovat aplikační port při startu, což ale dále komplikuje nastavení load balanceru. Snadněji a hlavně bez úpravy aplikace lze v moderních operačních systémech využít kontejnerizace. Za použití primitiv z LXC (*Linux Containers*) nebo abstrakce jako jsou Docker, rkt nebo containerd můžeme aplikaci spustit v izolovaném prostředí a pod známým portem. Pro jiné než Linuxové systémy existují různé alternativy, například Solaris má vlastní *Solaris Containers*.

Kontejnerizace má celou řadu dalších výhod. Díky tomu, že v obrazu (*Docker image*) jsou kromě samotné aplikace i všechny nezbytné systémové závislosti; typicky knihovny vyžadované k dynamickému linkování, nebo runtime pro interpretované jazyky (například JRE). Mezi další výhody patří zabudované automatické cachovaní kompilace vrstev Docker obrazu a snadná distribuce obrazů.

Jeden z problémů, který při stavění kontejnerů budu muset řešit, je zanořování Dockeru. První vrstva je prostředí, ve kterém se spouští jednotlivé CI joby. Kontejnerizace na této úrovni přináší lepší kontrolu závislostí a izolaci, ale není nutná. To je vesměs nezávislé na tom, jaké aplikace se budou na CI stavět. Druhá vrstva, ve které se Docker může využívat, je stavění Docker obrazů v rámci konkrétního CI jobu. Přístup k Dockeru na této úrovni nelze ničím nahradit. Jedna možnost zanořování se označuje *Docker in Docker* (DinD, diagram 1.1), při které všechny kontejnery stále sdílí jedno linuxové jádro, ale vnitřní kontejner je řízen Docker démonem spuštěným uvnitř vnějšího kontejneru.

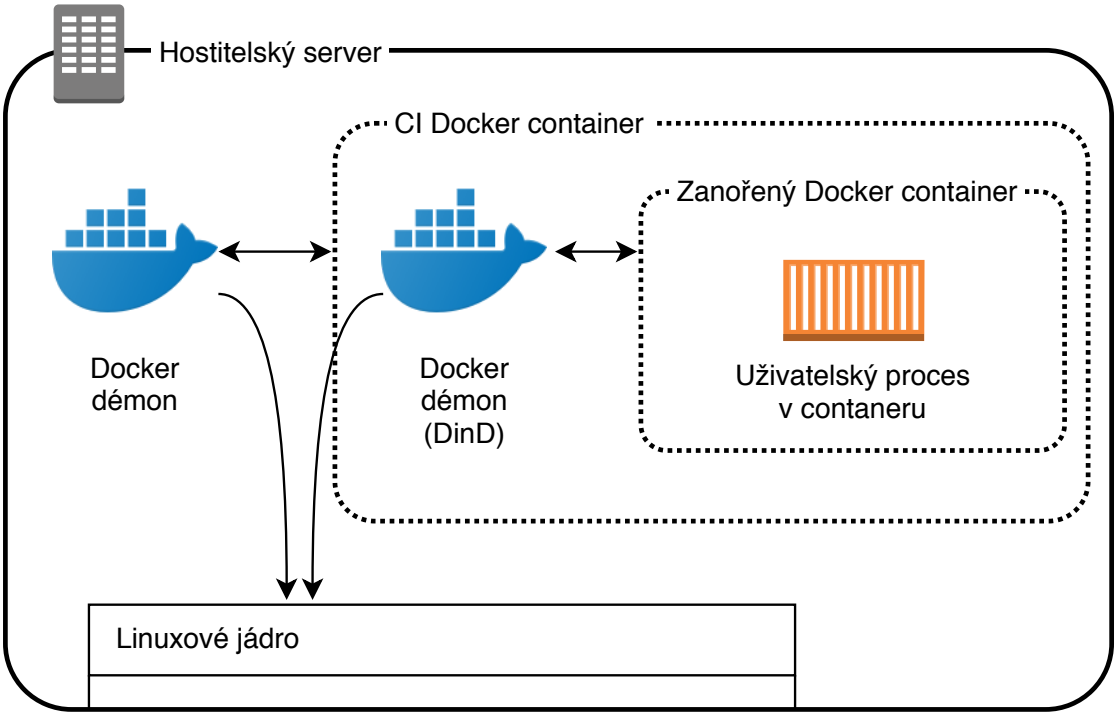


Fig. 1.1: Diagram použití Dockeru v Dockeru.

Speciálně pro testování je kontejnerizace nejlepší možné řešení. Testy je často potřeba spouštět na několika různých místech, minimálně na lokálních zařízeních všech vývojářů a na CI. S kontejnery odpadá nutnost instalovat celé testovací prostředí, které se navíc ještě víc blíží tomu produkčnímu. Díky tomu, že vývojáři a CI spouští testy (skoro) úplně stejně, odpadají z velké části problémy kde testy lokálně prochází, ale na CI ne. Kontejnery také zjednodušují začleňování nových vývojářů, kteří místo instalace všech závislostí a komponent mohou rovnou upravovat a spustit aplikaci.

Nasazování aplikace se kontejnerizací také usnadňuje. Na CI se nějakým deterministickým způsobem vytvoří Docker obraz a nahraje se do registru. Webové servery se pak informují o nové verzi, obraz stáhnou a spustí novou verzi aplikace. Odpadají tak některé problémy s atomicitou nasazení, s postupným nahrazováním a případně mazáním souborů. Další výhodou je, že vývojáři mohou lokálně spustit produkční verzi aplikace – například pro odlazení chyby.

Bezpečnost

V následující kapitole budu porovnávat různé CI nástroje a jedním z kritérií na které se zaměřím je bezpečnost. V rámci jedné organizace není žádoucí, aby všichni uživatelé viděli všechny ostatní testované projekty. Příklad z praxe jsou zaměstnanci bez podepsaného NDA nebo ve zkušební době.

Samostatná kategorie bezpečnosti CI je izolace spuštěným jobů. Lze rozlišovat následující úrovně:

- Žádná izolace. Všechny joby se navzájem ovliňují. Globální závislosti jako jsou runtime, databáze atp. jsou předinstalované.
- Kontejnerizace. Všechny joby sdílí společný hardware, ale každý job běží ve vlastním kontejneru a není nijak ovlivněn ostatními joby, ani se navzájem nevidí.
- Virtuální stroje (VM). Joby se spouští nad virtualizovaným hardware a odlišným jádrem. Není omezeno na Linux.
- Serverová/fyzická izolace. Každý job běží na jiném fyzickém serveru. Toto je prakticky velmi těžko dosažitelné, ale do jisté míry se lze tomuto ideálu přiblížit v cloudu, kde jsou stovky fyzických serverů. Přes to že hardware je sdílený, je pro potenciálního útočníka prakticky nemožné zajistit si kolokaci v krátkém čase co job běží.

Kontejnerizace je praktická, ale nedostačující izolace pro sdílená CI. Pokud se útočníkovi podaří z kontejneru utéct, má plnou kontrolu nad hostitelským systémem a i všemi joby. Přesně taková zranitelnost byla nalezena v únoru 2019 v nástroji runc, který využívají Docker, Containerd, CRI-O a další a také orchestrační technologie nad nimi postavené: Kubernetes, Docker Swarm, … (CVE-2019-5736 [29]). V rámci jedné organizace ale může být kontejnerizace dobrý kompromis mezi rychlostí, využitím zdrojů a bezpečností.

Dále u CI budu zmiňovat možnosti testování bezpečnosti aplikace. To lze rozdělit na dvě kategorie: statická analýza (SAST) a dynamická analýza (DAST). Rozdíly obou přístupů popisují následující odstavce.

SAST je statická analýza aplikačního kódu bez jeho spuštění [30]. Příkladem ve světě jazyka C je volání `printf(argv[1])`, které by mělo být nahrazeno za `printf("%s", argv[1])`. Kvalitní kompilátory mají nějakou formu SAST zabudovanou. LLVM pro ukázkový

`printf` vygeneruje *warning: format string is not a string literal (potentially insecure)*. Pro většinu programovacích jazyků a frameworků existuje externí statický analyzér (pro PHP phpstan, pro bash shellcheck, pro Dockerfile hadolint, …).

DAST je testování bezpečnosti aplikace při jejím běhu [31]. Často skenují různé body ze seznamu OWASP Top 10. Přestože žádný scanner nemůže najít všechny bezpečnostní chyby [32], jsou tyto nástroje užitečné.

Verzování závislostí

Uchovávání knihoven třetích stran (tzv. *vendoring*) je kontroverzní: někdo rozvážně nepreferuje ani jednu variantu [33], ale oba přístupy mají svoje problémy. Toto rozhodnutí částečně souvisí s CI, protože potřebujeme buď stahovat externí závislosti, nebo kontrolovat jejich aktuálnost.

Nevýhodou přidání závislostí do repozitáře je zvětšení repozitáře, což má negativní dopad na rychlost mnoha operací a podle druhu CI/CD pipeline přímý dopad na rychlost nasazování. Další problém je znepřehlednění rozdílů mezi jednotlivými verzemi v systému a při nevhodném použití verzovacího systému to může velmi komplikovat merge operace [34].

Výhody verzování externích závislostí jsou mnohé [35]. Při buildu CI/CD pipeline nemusí stahovat zdroje mimo samostatného repozitáře, což zvyšuje dostupnost při výpadku cizí služby (klasické závislosti jsou GitHub, JavaScript balíčky NPM, pro Javu Maven central repository). Aplikace se vždy kompiluje s předem známými a neměnnými verzemi závislostí, takže i při nezamknutí verze balíčku se při vydání breaking change nic nerozbije (v PHP composer.json, v NPM package.json, pro Java Maven pom.xml). Tento problém lze alternativně také řešit verzováním lockfile pro daný balíčkovací systém (composer.lock, package.lock/yarn.lock, Maven lockfile nemá); problém ale může stále nastat, pokud na cizím úložišti někdo otagovanou verzi balíčku přepíše. To se může stát buď omylem nebo třeba s nekalými úmysly a spoléhání na cizí úložiště při buildu tak lze považovat za bezpečnostní riziko. Při verzování závilostí mohou všechny zdrojové soubory podléhat kontrole. Dále verzování řeší projekty smazané [36]. Další výhodou je snadná úprava kódu závislostí bez nutnosti publikovat změny třetím stranám a čekat na integraci [37].

Kapitola 2

Porovnání nástrojů pro CI/CD

V této práci se budu soustředit na systémy pro webové aplikace. To jsou především Javascript, Java, Python a PHP [38]. Přesto že se CI většinou nelimitují na konkrétní jazyky, vyhnu se při porovnání systémům primárně pro mobilní a desktopové aplikace. Dále se nebudu věnovat úzce zaměřeným proprietárním systémům (jako jsou například *Visual Studio Team Services*, *TeamCity*, ...).

Přestože je hodně systémů prodáváno jako CI/CD, podporují většinou pouze CI část (primárně testování) a v nejlepším případě nějakou formou umožňují sledovat a spouštět nasazení aplikace. To je případ Jenkins, GitLab, Drone a dalších [39]. To souvisí s tím, že pro komplexní CD systém je nezbytná úzká integrace s hostitelskými servery – hlavní komponenty které CD konfiguruje jsou networking a potažmo DNS, HTTP web servery, můžou nastavovat load balancery případně jiné části infrastruktury jako jsou databáze atd.

2.1 Metodika porovnávání

U každého systému se nejprve seznámím s aktuální dokumentací. CI/CD systémy provozované pouze jako SaaS (*Software as a Service*) budu muset testovat v cloudu. Pokud systém nabízí self-hosted variantu, zprovozním ji na lokálním virtuálním serveru. Pro systémy s oficiálním kontejnerem budu systém spouštět v Dockeru. V případě, že systém má několik oficiálních variant instalace a nabízí jinou funkcionalitu (to je případ

GitLabu), nainstaluji a otestuji systém několikrát. Všechny instalace budou nascriptované a opakovatelné. Technickou část práce netýkající se přímo daných CI/CD systémů popisuji v příloze B.

Na instalovaných systémech pak provedu základní nezbytné nastavení: to bude pravděpodobně konfigurace veřejné URL, případně portů. Dále bude nutná konfigurace vzdálené správy repozitářů, pokud ho CI/CD nástroj nemá přímo zabudovaný. Bez dalšího nastavení zhodnotím zvlášť aplikační dostupnost za provozu a za správcovských úkonů: samostatně pro pro aktualizace CI systému a pro rekonfiguraci. Tím se dozvím, jestli se lze spolehnout na výchozí nastavení dodavatele. Dostupnost budu měřit nástrojem `siege` [40]. Jde o podobný nástroj jako je Apache `ab` [41], který navíc umí parsovat přijaté HTML a odesílat další požadavky na odkázané javascripty, kaskádové styly a další zdroje. Simuluje tak lépe chování uživatelů s webovým prohlížečem. Dostupnost budu testovat kontinuálními požadavky bez prodlev (volba `--benchmark`) s 10 uživateli (`--concurrent=10`). Výsledkem testu je počet selhaných požadavků, měřených podle HTTP kódu odpovědi. Ideální výsledek je 0 selhaných požadavků. Dostupnost za klasického provozu aplikace budu měřit po dobu 15 minut. U administrátorských úkonů budu dostupnost měřit po celou dobu úkonu, která se bude lišit.

Následně nakonfiguruji systém tak, aby měl co nejlepší možnou dostupnost a opět zopakuji testy na dostupnost.

Na třech ukázkových projektech otestuji možnosti CI, které systém nabízí. Aplikační testy budu z části simulovat jednoduchým bash scriptem. Zaměřím se na: sledování stavu testů a přehlednost (logování, reporting), možnosti debugování (vzdálené připojení do testovacího prostředí, `ssh`), uchovávání artefaktů (výsledků kompilace), konfigurace v kódu (vs konfigurace klikáním v administraci).

2.1.1 P1: Jednoduchá aplikace bez externích závislostí

Tato kategorie pokrývá výhradně statické weby. Aplikace může obsahovat dynamické části jako je například kontaktní formulář, ale samotné zpracování se děje mimo statickou aplikaci – cílem formulářů může být například Google Form [42], nebo jiné aplikace (ať už třetích stran nebo vlastní).

Pro testování CI/CD systémů jsem vytvořil jednoduchý statický projekt s nástrojem Jekyll [43]. Ten ze zdrojových souborů které se skládají hlavně z Markdown textů [44], metadat a šablon generuje statické HTML soubory. Dále umí kompilovat i styly, což

jsem využil pro jednoduchý `scss` soubor. Na závěr každé kompilace se všechny assety přemístí do složky podle aktuálního data. Tím je aplikace připravena pro nasazení současně se starší nebo novější verzí.

Při CI/CD se typicky v repozitáři uchovávají pouze zdrojová data. Pro generátory statických webů to mohou být například soubory ve formátu Markdown a specifikace HTML šablon. Z těch se pak v rámci kompilace na CI/CD pipeline vygenerují výsledné HTML soubory. Je vhodné, aby statické zdroje (JavaScripty, kaskádové styly, …) byly vygenerovány do unikátní složky, například podle verze buildu nebo podle času. Samotné nasazení na produkční server pak lze udělat bez výpadku tak, jak ilustruje diagram 2.1: nejprve je nutné nasadit nové statické zdroje. Ty mají unikátní názvy (nejlépe jsou v unikátně pojmenované složce), takže jejich nasazení nepřepíše současné soubory. Následně se přepíší HTML soubory. Uživatelé kteří ještě načetli starou verzi načtou staré zdroje. HTML soubory neexistující v nové verzi lze po uvážení z produkčního serveru odstranit. Není potřeba invalidovat cache, protože nové zdroje jsou pojmenované jinak než ty staré. Po uplynutí vhodného času lze staré zdrojové soubory ze serveru smazat. Pro vysokou dostupnost je vhodné použít víc než jeden server. V tom případě stačí nahrát nové zdroje na všechny servery a po bariéře přepsat všechna HTML.

Při použití kontejnerů nelze snadno dosáhnout toho, aby v kontejneru byly nové a současně staré zdroje. Místo toho se problém přesune na nějakou vyšší vrstvu, například ingress controller. Ten může podle HTTP kódu odpovědi rozhodnout, jestli se pokusí přeposlat požadavek na jiný kontejner. Pokud uživatel stáhne HTML z nové verze aplikace, ale request na CSS přijde na starý container a vrátí kód *404 Not Found*, může ingress controller GET požadavek přeposlat na nový kontejner, kde soubor existuje. Alternativou je použití session affinity, například pomocí cookies nebo IP adresy. Nebo lze mezi ingress controller a kontejnery umístit nějakou cachující HTTP vrstvu (HAProxy, Varnish), která prakticky v základním nastavení načte a uloží do paměti zdroje z obou verzí. Je pouze nutné ošetřit, aby HTML ze starých kontejnerů nepřepsalo HTML z kontejnerů nových, což může rozhodnout například podle hlavičky `Last-Modified-At`. Tento proces je vizualizován na diagramu 2.2.

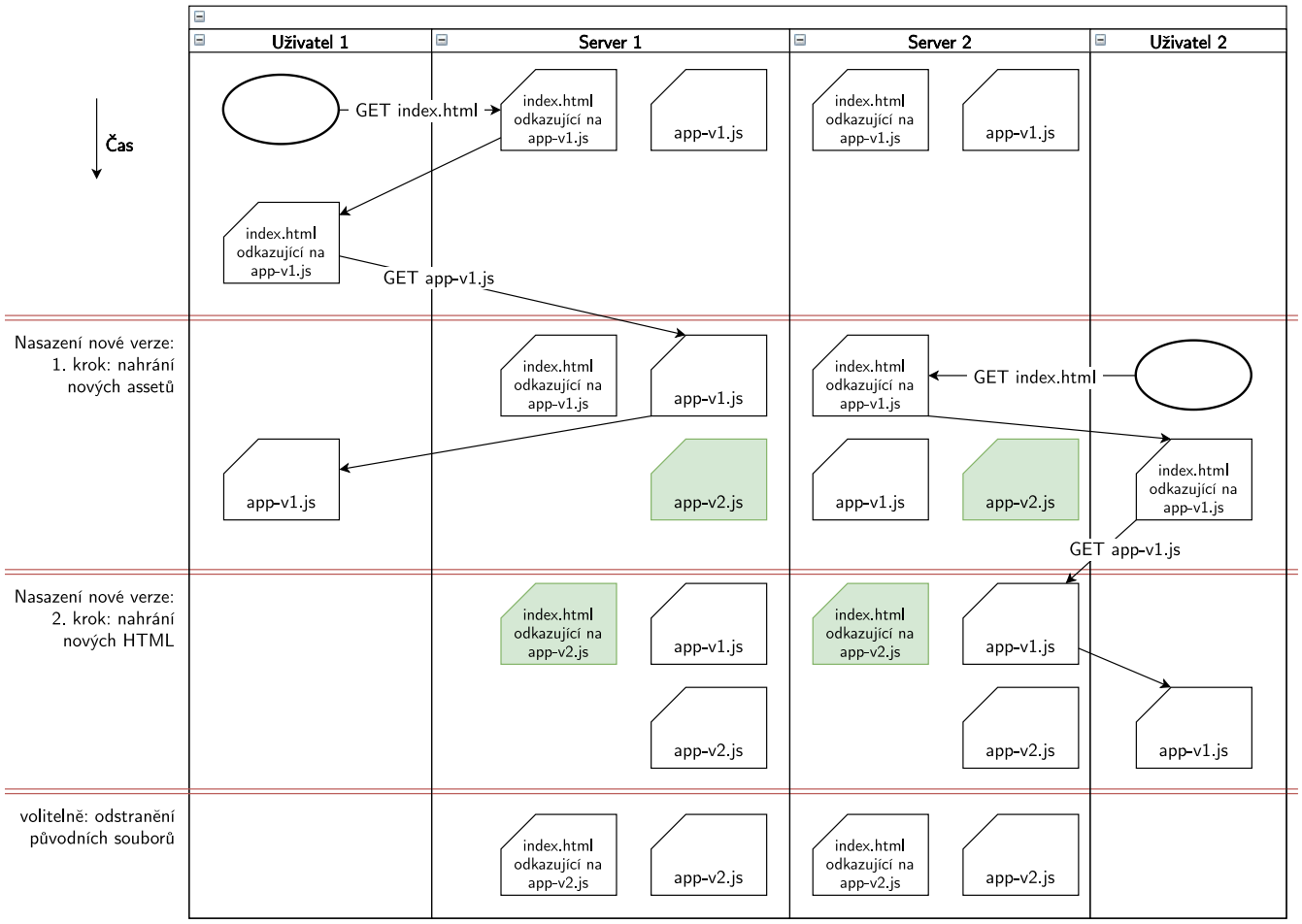


Fig. 2.1: Časový diagram znázorňující požadavky od dvou různých uživatelů ve dvou fázích nasazení nové verze statické aplikace. Uživatel 1 ilustruje nutnost nahrát nejprve na všechny servery sdílené assety. HTML musíme nahrát až v druhé fázi, aby bylo garantované, že uživatelé co načtou novou verzi HTML budou mít možnost ze všech serverů stáhnout odkazované zdroje. Uživatel 2 znázorňuje, proč je nutné uchovávat na serveru i starší verze zdrojů. Ty můžeme smazat až po nějaké delší době. Čas plyne odshora dolů, červené dvojité čáry reprezentují synchronizační bariéru.

2.1.2 P2: Komplexní aplikace

V této kategorii uvažuji komplexní aplikace vyžadující relační databázi, případně další závislosti jako můžou být fronta, key-value storage, mailer, služba na zpracování obrázků nebo generování faktur, nebo platební brána. Na rozdíl od statických webů se tyto aplikace testují obtížně. U databáze například stačí spustit pro testy nový databázový stoj a spustit migrace.

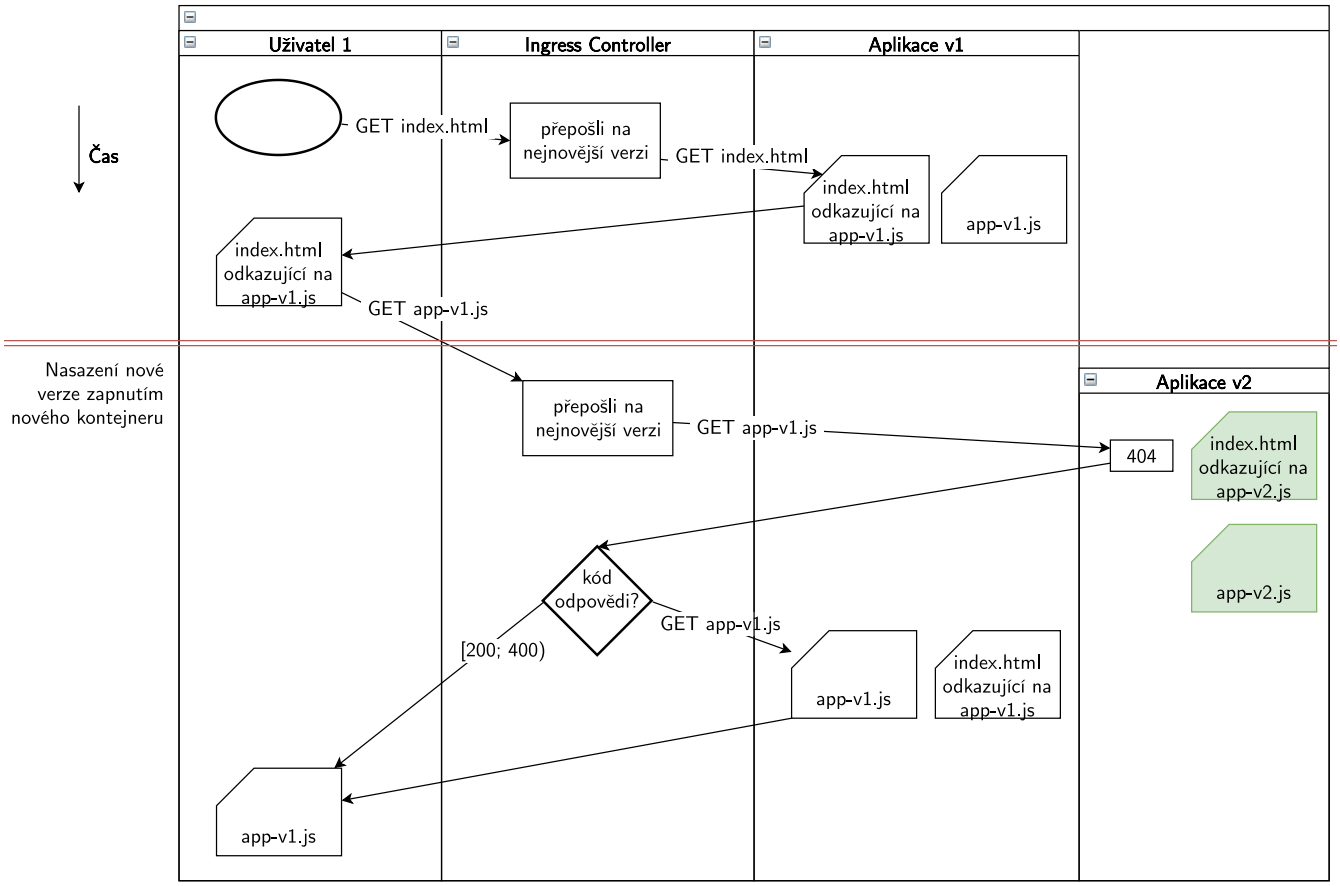


Fig. 2.2: Časový diagram pro nasazení nové verze statické aplikace za použití kontejnerů. Pokud není praktické v nové verzi kontejneru uchovávat i starší verze zdrojů, je nutné implementovat nějakou komponentu (zde pojmenovanou ingress controller), která při obdržení odpovědi s HTTP kódem 404 přepošle požadavek na starší verze aplikace. Toto chování je pro uživatele transparentní a projevuje se pouze lehce zvýšenou latencí.

V případě platební brány která může od aplikace požadovat veřejně dostupnou url pro webhooks to může znamenat, že už samotný test vyžaduje nějakou úroveň nasazení aplikace.

Pro testování jsem implementoval blog v PHP, konkrétně za použití frameworku Symfony [45]. Aplikace při zpracování každého požadavku posílá dotazy na články do externí MySQL databáze. Dále komunikuje s key-value serverem Redis, kde spravuje session. Třetí závislost je API třetí strany: `mailgun.com`, které se používá pro posílání transakčních emailů.

2.1.3 P3: Aplikace distribuovaná jako kontejner

Aplikace v kontejneru mají stejné nároky na testovací prostředí jako aplikace v předchozí skupině a v rámci CI/CD se liší pouze v buildu a nasazení. Jelikož samotné CI často používá Docker, bude při tomto testu nutné řešit DinD nebo nějakou alternativu. Vystavěný Docker obraz je potřeba nahrát do nějakého repozitáře. Teoreticky lze obrazy exportovat/importovat i jako soubory, ale to není praktické, mj. i proto, že se pak neuplatní sdílení jednotlivých vrstev obrazu. Některé systémy mají přímo zabudovaný registr obrazů a pro ostatní budu muset používat nějaký externí registr.

2.2 GitLab

GitLab je především systém pro správu repozitářů a vzniknul jako otevřená alternativa služby GitHub. Nabízí ale velmi kvalitní vlastní CI a má nástroje podporující CD. GitLab je poskytován jako SaaS a to jak v managed variantě, tak v placené self-hosted variantě. Existuje i funkčně velmi ořezaná self-hosted verze zdarma. Veřejné jádro je poskytované jako *GitLab Community Edition*, placené části jsou vedeny jako *GitLab Enterprise Edition*.

Oficiálně GitLab podporuje celou řadu instalačních možností od balíčků pro klasické Linux distribuce, přes Docker kontejnery až po složitější šablony pro Kubernetes nebo OpenShift. Do podzimu 2018 byl GitLab distribuován jako monolitická aplikace, které se velmi obtížně škálovala a spouštěla distribuovaně v několika replikách. Podařilo se ale GitLab rozdělit na jednotlivé komponenty (gitaly: git repozitáře, gitlab-shell: HA api nad gitaly, mailroom: správa příchozích emailů, sidekiq, task-runner, unicorn: ruby webserver). Dále ma GitLab řadu externích závislostí: relační databázi (konkrétně PostgreSQL, podpora pro MySQL už neexistuje), key-value storage (Redis), úložiště pro objekty (AWS S3 nebo otevřená alternativa s kompatibilním api Minio). V oficiálním docker image, kterému říkají *omnibus*, jsou všechny tyto závislosti přibalené. Nově vznikly separátní docker image pro každou GitLab službu zvlášť a šablony pro Kubernetes Helm, usnadňující jejich spuštění a konfiguraci. Protože jsou oba přístupy diametrálně odlišné, rozhodl jsem se nasadit GitLab jak v původním *omnibus* variantě distribuované jako balíček pro Debian, tak ve variantě mikroslužeb v prostředí kontejnerů a porovnat obě varianty.

2.2.1 GitLab Omnibus

Instalace podle oficiální dokumentace je velmi jednoduchá. Jde jenom o přidání vlastního repozitáře a instalace balíčku [46]. Po instalaci je rovnou spuštěna celá aplikace a při přístupu na HTTP endpoint se rovnou zobrazuje dialog pro nastavení administrátorského hesla.

Aplikaci lze konfigurovat na několika místech: z webového rozhraní, `gitlab.yml` a `gitlab.rb`). V každé části je bohužel trochu něco jiného. V rámci této práce jsem upravil konfiguraci v `gitlab.rb`, kde jsem nastavil externí URL, na kterou GitLab generuje externí linky (například v emailové komunikaci), počáteční heslo pro administrátora a token pro autentifikaci GitLab CI.

2.2.2 GitLab mikroslužby

GitLab začal oficiálně na konci roku 2018 podporovat také nasazení do Kubernetes přes Helm [47]. Kubernetes je distribuovaný orchestrátor kontejnerů a Helm je balíčkovací aplikace. Každá komponenta GitLabu byla oddělena do samostatného docker obrazu a v Helm balíčku (*Helm chart*) je pomocí Kubernetes zdrojů popsáno jak se dané kontejnery mají spouštět a jak jsou provázané. Teoreticky je možné kontejnery spouštět i ručně v jiném systému než Kubernetes, ale není to oficiálně podporované a neexistuje k tomu dokumentace.

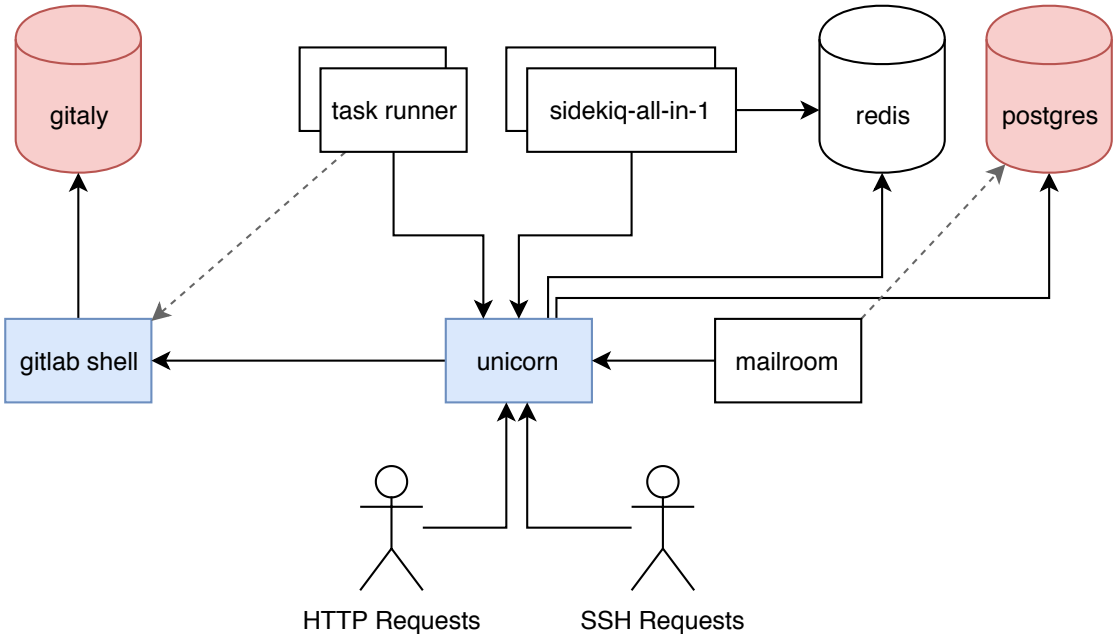


Fig. 2.3: Architektura GitLab mikroslužeb a jejich propojení. Modře zvýrazněné komponenty jsou bezstavové, můžou běžet ve víc replikách a je před nimi nějaký load balancer. Zdvojené komponenty jsou workery a lze je horizontálně škálovat. Červeně jsou znázorněny SPoF, komponenty které ukládají stav a nelze je snadno distribuovat.

Hlavní výhody nasazení GitLabu přes mikroslužby jsou:

- Přehlednost. Administrátor přesně ví, jaké komponenty systém obsahuje. Jednotlivé části nejsou skryté v Omnibus balíčku, kde se těžko dohledávají logy.
- Metriky mohou přímo využívat exportery nasazené v clusteru. V Omnibus jsou k dispozici pouze metriky pro celý balík, nebo je musí další proces uvnitř exportovat. To je další komplexita a nekonzistence se zbytkem prostředí.

- Lepší škálovatelnost. Můžeme škálovat pouze některé komponenty. Navíc díky přesným metrikám můžeme škálovat vytížené komponenty dynamicky.
- Využití upstream služeb. Řada komponent, které GitLab využívá, jsou nezměněné aplikace třetích stran. To je například ElasticSearch, Redis a relační databáze. V Omnibus balíčku jsou přibalené; nelze je aktualizovat a pokud na to GitLab nemyslel, nelze je vyčlenit a použít externí službu. U mikroslužeb má administrátor plnou kontrolu nad tím, které služby nasadí a pro které například použije cloudovou managed variantu.

Instalace GitLabu jako mikroslužeb je oproti verzi Omnibus velmi komplikovaná. Vyžaduje minimálně uživatelskou znalost Kubernetes a velmi podrobnou znalost Helm. Dokumentace je zatím (k lednu 2019) nekompletní a hodně nastavení a jejich význam je nutné dohledávat v šablonách Helm chartu. Některé konfigurace nejsou podporovány vůbec. Narazil jsem například na nutnost nakonfigurovat Redis clusteru s heslem [48]. Například managed varianta od AWS podporuje přihlášení heslem pouze při použití TLS tunelu, protože bez něj se posílá heslo po síti v čistém textu a nedává moc smysl vůbec heslo použít. To souvisí s tím, že GitLab mikroslužby jsou velmi nové; dá se očekávat, že čím víc firem bude takto GitLab nasazovat, tím lepší dokumentace a podpora pro různá prostředí vznikne.

2.2.3 GitLab CI

CI není přímo komponenta distribuovaná s GitLab, je ale dobře zaintegrovaná. Díky skvělému návrhu GitLab pouze vystavuje události na API. K API se může registrovat libovolné množství runnerů, které se periodicky API dotazují na nové práce ke spuštění [49]. Runner může běžet na stejném serveru jako GitLab, ale velkou výhodou je možnost spustit runner i na jiném vzdáleném serveru – například pro mobilní aplikace se často používá Mac mini s macOS, které je pro kompilaci nezbytné. Při registraci runneru je možné ho přidělit pouze některým projektům podle tagu, nebo ho zveřejnit pro všechny projekty.

Runner při získání práce z API má k dispozici soubor s definicí (obsah souboru `.gitlab-ci.yml` [50]) a přístupové údaje k repozitáři. Architektura GitLab CI se skládá z tzv. *stages* (etapy) a *jobs* (kusy práce), jak ilustruje diagram 2.5. Jeden push do repozitáře typicky vygeneruje jednu *pipeline*, což je sada *stages* a *jobs* popsaná konfigurací CI v daném commitu.

```yaml
stages:
  - build
  - deploy

build:
  stage: build
  artifacts:
    paths:
      - vendor/
  script:
    - make build

deploy:
  stage: deploy
  only:
    refs:
      - "deploy/prod"
  dependencies:
    - build
  script:
    - make deploy
  environment:
    name: prod
    url: "http://p2.ditemiku.local"
```

Fig. 2.4: Zjednodušená ukázka konfigurace GitLab CI pomocí souborem `.gitlab-ci.yml`. Tento útržek definuje dvě stage, z nichž každá obsahuje jeden job. Samotná logika spuštění kompilace a nasazení na servery je delegována na `Makefile` v repozitáři.

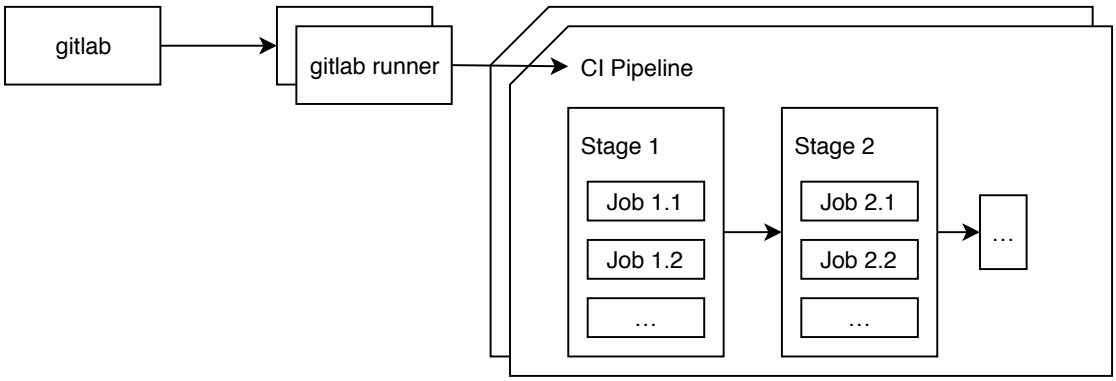


Fig. 2.5: Architektura GitLab CI. Runnerů i pipeline může existovat mnoho. V rámci jedné pipeline je několik stages, uvnitř kterých je sada paralelně spuštěných jobs.

V konfiguračním souboru lze nastavit celou řadu pokročilejších funkcí [50]. Lze spouštět joby pouze pro určité kontexty podle kombinace proměnných prostředí a informace z verzovacího systému. Dále je možné joby provazovat podle závislostí a tím je serializovat. Toto ale není implementováno dokonale a všechny joby v rámci jedné stage běží najednou a čekají na doběhnutí všech jobů v předchozí stage. Celá pipeline tak může trvat delší dobu než je nutné.

V oficiální implementaci nabízí runner celou řadu možností, jak jednotlivé *jobs* spouštět, tzv. *executors* [51]. Základní je `shell`, při kterém neexistuje prakticky žádná izolace procesů a čištění prostředí. Jakékoliv závislosti, které je potřeba instalovat do sytému, ovlivňují všechny ostatní procesy na stejném serveru. Procesy běží pod neprivilegovaným uživatelem `gitlab-runner`, takže je lepší potřebné závislosti nainstalovat předem administrátorem. Tento executor může být vhodný pro dedikované prostředí pokud zajistíme, že tento server využívá pouze jeden projekt, případně pouze důvěryhodné nekolidující projekty. Velmi podobně funguje executor `ssh`, který místo na lokálním stroji spouští příkazy vzdáleně. Na cílovém stroji nejsou potřeba žádné závislosti (samozřejmě s výjimkou ssh serveru); v cloudovém prostředí může být praktické mít čisté virtuální stroje, na které se pak ssh runner připojuje. Dále lze ssh executor využít pro systémy, na kterých runner nemůže běžet lokálně. Velkou nevýhodou tohoto prostředí je nízká opakovatelnost. Jelikož se všechny joby mohou navzájem ovlivňovat a hostující systém se průběžně mění, při opakovaném spuštění nemusí starší joby už fungovat, přestože dřív procházely úspěšně. Ostatní executory tento problém do značné míry eliminují.

Zbylé executory mají kompletní izolaci všech jobů. Možnosti `parallels` a `virtualbox` startují pro každý job nový virtuální stroj, z jednoho obrazu podle konfigurace runneru. Kontejnerizace jobů je nabízena v executoru `docker`: pro každý job je možné zvolit vlastní image. Varianta `kubernetes` je pro runner běžící v Kubernetes clusteru a vytváří pro každý job nový pod. Všechny tyto executory mají dobře opakovatelné joby. Pokud nezměníme obraz virtuálního stroje, případně pokud používáme stále stejný docker image, budou až na případné externí závislosti joby spuštěné identicky jako dřív.

Nastavení CI se dále komplikuje při buildu Docker obrazů. Při použití většiny executorů máme dvě možnosti. První varianta je použít hostitelský Docker. U `shell` varianty (a potažmo `ssh`) executoru stačí vyřešit oprávnění (typicky je potřeba přidat uživatele `gitlab-runner` do skupiny `docker`). U `docker` případně `kubernetes` executoru lze připojit ovládací Unix socket `/var/run/docker.sock` do našeho kontejneru. Pro virtuální

stroje by teoreticky šel socket připojit také, ale když přijdeme o izolaci nemusíme pak VM používat vůbec. Nevýhodou tohoto řešení je, že všechny aplikace využívající CI vidí obrazy a cache ostatních aplikací. Pokud se jedna z aplikací přihlásí do vzdáleného registru a stáhne nějaký soukromý obraz (`docker pull`), všechny ostatní aplikace tento obraz pak vidí a mohou ho číst i přesto, že nemají k vzdálenému registru oprávnění. Podobně jako při použití `shell` executoru tak musíme při sdílení Docker daemonu na CI stavět pouze důvěryhodné aplikace.

Druhou variantou buildu Docker obrazů je tzv. *Docker in Docker* (DinD). To je Docker démon zabalený do Docker kontejneru. Při spuštění stále sdílí Linuxové jádro s hostitelským systémem, ale samotný Docker proces je izolovaný: má samostatný seznam procesů, vlastní limity a oddělenou build cache. DinD může být spuštěný právě pro jeden job. Ukázková specifikace pipeline pro Docker v kontejnerovém prostředí je popsána v kódu 2.6.

```
services:
  - "docker:dind"

variables:
  DOCKER_HOST: "tcp://docker:2375"

build:
  stage: build
  script:
    - docker build --tag demo-build .
```

Fig. 2.6: Ukázkový soubor `.gitlab-ci.yml` pro nastavení DinD.

Při samostatně spuštěném DinD pro každý build máme perfektní izolaci všech klientů, ale přicházíme o některé výhody Dockeru, konkrétně o stažené obrazy a cache předchozích buildů. To je známý problém a GitLab ho bez úspěchu řeší už několik let [52]. Největší výhoda tohoto řešení je perfektní dostupnost: každá pipeline má vlastní Docker a případné aktualizace se dějí na úrovni změny obrazu v `.gitlab-ci.yml`. Přes GitLab mechanizmus pro cachování lze uložit Docker data a při dalším jobu je opět nahrát do nového Docker daemona [53]. Toto řešení je ale velmi pomalé. Obrazy včetně cache budou běžně kolem stovek megabajtů a zapisujeme je dokonce hned čtyřikrát: jednou na disk při exportu, potom do GitLab cache což je často externí objektové úložiště (jako je AWS S3) a po čtvrté při načítání do prázdného Dockeru. Pro aplikace s extrémně dlouhým buildem v řádu desítek minut to může být

přínosné, ale pro typické webové aplikace na které se tato práce soustředí není toto řešení vhodné.

Velmi úspěšně lze ale DinD provozovat jako dlouhožijící službu oddělenou od samotné CI pipeline. Získáme tím tak výhody obou řešení a veskrze eliminujeme všechny nevýhody. Pro každou skupinu navzájem si věřících aplikací spustíme samostatný DinD proces. Ten může běžet na hostitelském serveru s CI nebo libovolném externím. V daných pipeline specifikacích pak stačí neuvést `services: "docker:dind"` a nakonfigurovat proměnnou `DOCKER_HOST` (např. `tcp://group-7.external-docker.local:2375`). Každá skupina aplikací uvidí pouze svoje obrazy a bude mít persistentní cache. Nevýhodou tohoto řešení je vysoká komplexita. Je nutné nějakým způsobem zajistit, aby se k Docker socketu mohly připojit pouze autorizované aplikace. Na to je praktické použít nějakou pokročilou abstrakci, například Network Policies v Kubernetes. Samotná ACL logika může být v síťové vrstvě jako je například Weave, Contrail nebo Calico, nebo v samostatném dedikovaném firewallu.

2.2.4 Podpora CD praktik

Ve svém marketingovém copy se GitLab prezentuje jako jednotný systém pro CI/CD, monitoring a bezpečnost. Jak jsem popsal v předchozí sekci, CI nabízí GitLab perfektní. Co se týče podpory CD, funkce GitLabu jsou omezené:

Environments přináší přehled všech prostředí (například beta, produkce, ...) a nasazení do těch to prostředí [54]. Užitečná je možnost spouštět z tohoto zobrazení deploy znovu. Lze spustit i deploy nějaké starší verzi a udělat tak rollback. Při vhodném rozdělení pipeline na build a deploy je spuštění z tohoto přehledu je rychlejší, než spouštět celou pipeline. GitLab nenabízí žádnou možnost jak rollback omezit a je pouze na uživatelích, aby nespustili rollback na nějakou starou nekompatibilní verzi.
Review Apps jsou prodávány jako systém, díky kterému lze nasadit zkušební verze aplikace do oddělených dynamických prostředí [55]. Prakticky to jde udělat v libovolném CI systému; pro vybrané větve se spustí deploy a dynamicky se předá nějaký identifikátor pro veřejnou URL a další konfiguraci, to může být například název dané větve. Jediné co přináší GitLab je podpora pro dynamické *Environments* bez nutnosti programovat to přes API, a tlačítko *Stop* v webovém rozhraní, které spustí pipeline job která má na starost job vypnout. Veškerá praktická implementace je ale na vývojáři: GitLab neřeší nasazení a vypnutí aplikace, pouze zavolá uživatelem definovaný job.

Auto DevOps [56]. Tato kontroverzní [57] funkce GitLabu je zjednodušeně řečeno jenom hodně komplikovaný `.gitlab-ci.yml` soubor, který uživatel nevidí a použije se, pokud není v repozitáři jiná konfigurace pipeline. Auto DevOps pipeline má kolem 1000 řádek a obsahuje pokročilé Yaml konstrukce jako jsou reference a vnořování, které zhoršují čitelnost. Navíc přimíchávají bash scripty a využívají docker kontejnerů. Celá pipeline je ve výsledku hodně složitá. Auto DevOps podporuje pouze nasazování do Kubernetes clusteru. Build aplikace funguje jenom když je aplikace samotná stavěná pomocí `Dockerfile`. Distribuce obrazu musí být pomocí GitLab Container Registry. Deploy probíhá pomocí Helm. Lze použít výchozí Helm Chart distribuovaný s Auto DevOps, nebo použít vlastní. Ve vlastním Helm Chartu, resp. v konkrétních Kubernetes zdrojích, lze pak ručně nakonfigurovat škálování a dostupnost pro konkrétní aplikaci. Ve výchozím nastavení není rolling-update nakonfigurován.

Ze všech systémů které řeší CI a zároveň CD je GitLab se svým Auto DevOps nejdál. Je ale velmi dogmatický a nebude fungovat s žádnou odchylkou v infrastruktuře.

Drobnější funkce GitLabu podporující CD:

- Možnost spouštět vybrané joby v pipeline pouze manuálně. Jedno z možných použití je deploy aplikace z `master` větve, kde vývojář může ručně spustit job pro deploy po doběhnutí předchozích automatických jobů. Bohužel všechny automatické joby definované v následujících stages se spouští bez čekání na manuální job a to i tehdy, když mají definované závislosti [58].
- Plánované spuštění pipeline. Přestože lze build spouštět programaticky z API z libovolného cronu, je užitečné že to má GitLab přímo implementované. Typické použití je tzv. *nightly build*.
- **Web terminals** umožňují připojit se do aplikačního kontejneru v Kubernetes clusteru. Podobně jako u Auto DevOps je tato integrace dogmatická a nekompatibilní s klasickým použitím Kubernetes labels [59].

2.2.5 Open-source

GitLab je vyvíjen jako otevřený software, v tom smyslu, že veškerý kód je veřejně čitelný. Některé části jsou opravdu open-source i co se licence týče: GitLab CE je pod MIT licencí. Jiné části, například GitLab EE, jsou pod vlastní licencí. Licence dokonce ani nedovoluje EE verzi provozovat bez zakoupené licence.

Příspěvky a návrhy na změny kódu od lidí mimo GitLab Inc. jsou možné, ale vesměs ignorované. Jako přispěvatel jsem hodně cítil, že firma GitLab je čistě remote a nemá žádné kanceláře a tým na jednom místě [60]. Většina návrhů na změnu – od jednořádkových oprav po drobnější funkční úpravy – co jsem poslal zůstala bez povšimnutí. To je tím spíš mrzuté, když razí cestu „minimum konfigurace, radši pošlete návrh na úpravu kódu pro všechny“ [61].

U jednoho úkolu do kterého jsem se víc zapojil trvalo rok a půl začlenit drobnou změnu. Na tyto změny čekalo několik desítek firem. Šest vývojářů poslalo nezávisle na sobě návrhy na změnu. Změny byly vesměs triviální, bez dopadu na ostatní části kódu, byly otestované a dobře zdokumentované. Po opakovaném urgováním přes oficiální kanál podpory se u úkolu vyjádřil zaměstnanec GitLab. Bohužel se pouze zeptal, kdo má tuto část kódu na starost a tím na dva měsíce úkol opět usnul. Když se po dalším urgování přes podporu úkolu někdo chopil, zadání vesměs nepochopil a snažil se protlačit nepoužitelné řešení. Nechal si naštěstí problém znovu vysvětlit a řešení přehodnotil. Nakonec byl celý problém vyřešen a změny jsou zařazeny do vydání 11.8, tedy skoro 2 roky po otevření úkolu. Celé vlákno je veřejně na GitLab Issue Trackeru [62]. Podobnou zkušenost mám i z několika dalších GitLab projektů, včetně aktivně vyvíjeného GitLab Helm Chart.

2.2.6 Rozšiřitelnost

Primární bod pro integraci s GitLabem jsou webhooky, které jsou vyvolané v reakci na systémové nebo projektové události [63]. Alternativou jsou tzv. pluginy, což jsou spustitelné soubory na disku aplikačního serveru, které přijímají identický vstup a události jako webhooky [64]. Obě varianty jsou pouhé informace pro externí aplikace a neumožňují nijak rozšiřovat zabudované funkce nebo UI GitLabu.

2.2.7 Zálohování

GitLab nabízí nástroj, který zálohuje všechny komponenty: databázi, objektová úložiště (přílohy, výsledky CI/CD, Docker Registry, Git LFS) a Git repozitáře. V Omnibus balíčku bez externích komponent je tedy zálohování snadné. Při rozdělení na mikroslužby se zodpovědnost za zálohování částečně přesouvá: u databáze a objektového úložiště (např. *AWS S3*) typicky na poskytovatele cloud řešení.

Záloha kterou GitLab vytváří není atomická. Databáze se sice zálohuje v jedné transakci, ale ostatní komponenty žádné snapshoty obecně nepodporují. To lze simulovat vypnutím aplikace a vytvořením záloh. Nicméně, díky tomu že GitLab nejprve zálohuje databázi a potom ostatní data, nejpravděpodobněji se stane to, že ve výsledném balíčku budou nahrána i data na která nic neodkazuje. Nemůže se např. stát, že by komentář obsahoval neexistující obrázek.

2.2.8 Zabezpečení

Podle CVE Details měl GitLab nejvíc kritické problémy – hodnocené podle podle CVSS – zjištěné v posledním roce (2018) [65]. To může souviset s exponenciálním růstem, který zažili díky `#movingtogitlab` a akvicizi konkurenční firmy GitHub firmou Microsoft [66]. Není zatím jasné, jestli dramatický nárůst počtu uživatelů přilákal bezpečnostní analytiky, nebo jestli bezpečnostní problémy jsou výsledkem zrychleného vývoje a zhoršení kvality produktu.

Správa problémů (*issues*) u projektů na GitLabu má možnost vytvořit nový záznam v důvěrném režimu. Takové problémy jsou pak viditelné pouze pro správce repozitáře. To je vhodné pro *responsible disclosure* (zodpovědné zveřejnění) bezpečnostních chyb. GitLab toto úspěšně používá i pro vlastní projekty.

GitLab zavádí v kontextu CI/CD nový pojem *continuous security testing* (kontinuální bezpečnostní testování) [67]. Na rozdíl od testování po začlenění změn a nasazení, GitLab umožňuje každému vývojáři nasadit danou větev verzovacího systému do vlastního testovacího prostředí. To je znázorněno v diagramu 2.10.

2.2.9 Dostupnost

S GitLabem mám praktické zkušenosti a provozuji ho pro přibližně 50 lidí. Zhruba 5 měsíců jsem používal balíček *Omnibus* a s GitLab mikroslužbami pracuji podobně dlouho. Vyzkoušel jsem si i migraci z Omnibus na mikroslužby.

Obě varianty nasazení byly při běžném používání (tzn. ne při správě) dobře dostupné. Většina výpadků nastala při problematických datech, které měly za následek extrémní zpomalení z pohledu uživatele. Jeden uživatel například nahrál 300 MiB dump databáze a GitLab, resp. Gitaly, přestal příkaz *diff* fungovat a na všechny požadavky po minutě odpovídal chybou. Jeden projekt tak omezil všechny ostatní projekty. GitLab by

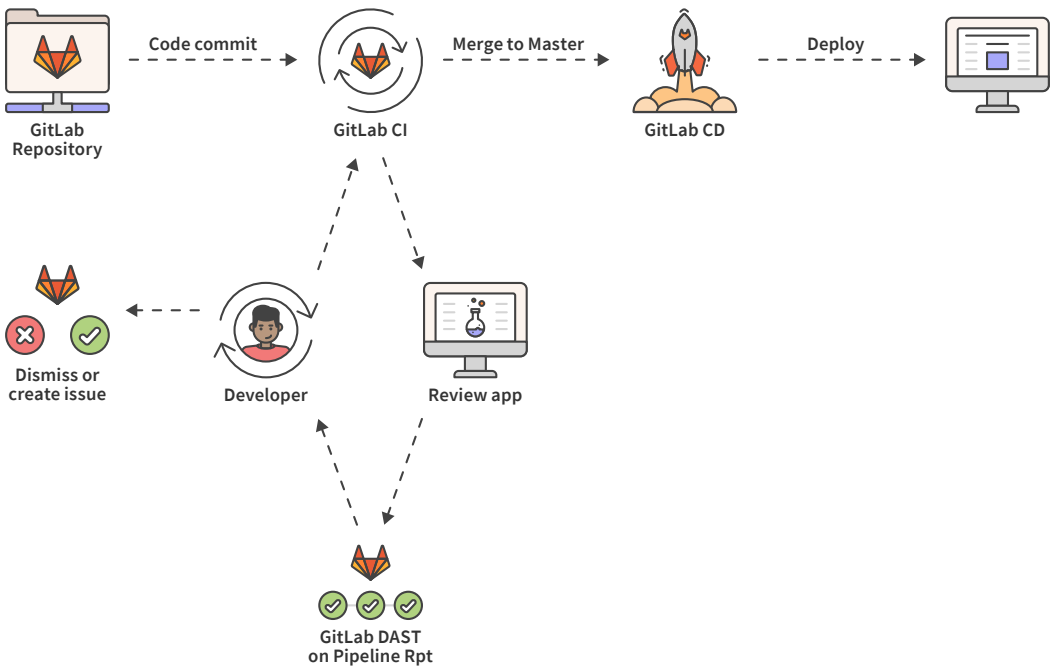


Fig. 2.7: Cyklus kontroly kvality aplikace GitLab s integrovaným DAST [67]. Vývojář může vyhodnotit bezpečnostní problémy před začleněním změň do sdíleného kódu.

mohl mít nastavené lepší limity aby těmto výpadkům předešel, ideálně dynamicky podle dostupných zdrojů. Při nasazení GitLabu pomocí mikroslužeb jde tento problém částečně obejít automatickým škálováním.

2.2.9.1 GitLab omnibus

Omnibus distribuce GitLabu, tedy balíček který obsahuje všechny komponenty, je primárně pro snadné nasazení a nepočítá se s tím, že by běžel s vysokou dostupností (HA). Umožňují ale aktualizovat systém bez výpadku: doporučený postup je aktualizovat balíček, spustit databázové migrace a obnovit webový frontend a konzumenty fronty [68]. Důsledně se snaží dodržovat kompatibilitu napříč verzemi a databázové migrace dělají zpětně kompatibilní. V každém vydání nové verze jsou změny rozepsané a případné nekompatibility jsou znázorněny v tzv. *upgrade barometer*.

V základním nastavení nelze GitLab Omnibus replikovat. Některé komponenty lze ale vyčlenit. Nezbytné jsou PostgreSQL a Redis. Jediné co pak zbývá a sdílí se mezi replikami jsou samotné git repozitáře na souborovém úložišti.

Vyzkoušel jsem, že GitLab Omnibus správně funguje ve víc replikách, když se vyčlení databáze a repozitáře se sdílí přes NFS. Jedná se ale o nezdokumentované nasazení a nemá oficiální podporu.

Při nasazení ve víc replikách lze docílit 100 % dostupnosti při aktualizaci na novější verzi.

2.2.10 GitLab microservices

Na diagramu 2.3 (strana 22) je znázorněna architektura GitLab mikroslužeb. Mikroslužby jsou obecně vhodné pro HA systémy:

> […] each microservice in microservice architectures is operationally independent from others, and the only form of communication between services is through their published interfaces. This is fundamental since this allows one to change, fix or upgrade a microservice without compromising the system correctness, provided that the interfaces are preserved. *Dragoni et al. [69]*

Klíčové komponenty GitLabu můžeme provozovat ve víc replikách a aktualizaci dělat pomocí rolling update, naprosto transparentně pro ostatní části systému. Některé komponenty jako jsou konzumenti fronty a manažer příchozích emailů můžeme dokonce aktualizovat s krátkým výpadkem, bez pozorovatelného dopadu pro uživatele.

Správa stavových služeb je v distribuovaném systému nejkomplikovanější. GitLab jich bohužel používá celou řadu: relační databázi PostgreSQL, key-value storage Redis a vlastní služby Gitaly pro správu repozitářů. PostgreSQL s master-slave architekturou lze provozovat s vysokou dostupností použitím hot standby instance [70]. Redis také používá master-slave architekturu, ale dokáže si v případě výpadku sám zvolit v clusteru nový master díky komponentě Redis Sentinel [71]. Tyto služby poskytovatelé cloudu nabízí i se správou, kde zodpovídají za dostupnost a některé aktualizace.

Služba Gitaly je v oficiálním Helm chartu SPoF. Vystavuje gRPC API nad git repozitáři, se kterým pracuje většina ostatních komponent. Gitaly běží v jedné replice a je závislé na stavových datech na disku. Konzultoval jsem tento problém s oficiální podporou a je možné Gitaly provozovat nad NFS ve více replikách. Jde o kompromis mezi dostupností a rychlostí. Gitaly je z části limitováno propustností disku a NFS některé operace zpomalí.

V oficiální distribuci má GitLab SPoF v Gitaly a při aktualizaci této komponenty je nedostupný. Při použití mikroslužeb s Gitaly nad NFS je GitLab bez výpadku dostupný pro použití i pro správu a dostupnost lze zvyšovat přidáním dalších replik.

2.2.11 Integrace

GitLab CI vyžaduje repozitáře na GitLab. Kromě celé řady variant importu z různých externích úložišť (včetně GitHub, Bitbucket a desítky dalších) lze také vytvořit mirror libovolného repozitáře. Lze tak de facto použít GitLab CI s jakýmkoliv repozitářem.

Integrace s externími službami má GitLab suverénně nejlepší ze všech testovaných systémů. V základu je zabudovaných přes 30 služeb, od chatovacích služeb po správy úkolů a některé ovlivňují i UI. Například *External Wiki* přidává do panelu odkazů novou položku.

GitLab má zabudovanou podporu pro nasazování do Kubernetes. Autentizace je možná pouze přes token a nelze využít například certifikáty nebo externí autentizační službu. Po počáteční instalaci komunikuje GitLab s Kubernetes přes balíčkovací systém Helm. Deploy do Kubernetes může fungovat bez další konfigurace díky Auto DevOps [56]. Dokonce v základu fungují i Review Apps [55].

2.2.12 Praktické nasazení projektů

2.2.12.1 Projekt 1

Nastavení pipeline pro statický projekt není vůbec přímočaré. Z webového rozhraní jsem vytvořil projekt včetně repozitáře, v lokálním git repozitáři jsem nastavil přidělený remote a zdrojové kódy jsem nahrál na GitLab. Potom jsem ručně napsal konfiguraci pipeline v souboru `.gitlab-ci.yml`. Protože jsem GitLab runner nasadil s executorem `shell`, nainstaloval jsem předem do sdíleného prostředí nezbytné závislosti. Popis instalace Ruby, Gem a balíčků Jekyll a Bundler jsem popsal v příloze B. Konfigurace GitLab pipeline obsahuje značné množství klíčových slov, ale má vynikající webovou dokumentaci. Uvítal jsem také podporu schématu v IDE IntelliJ IDEA [72].

Pipeline jsem specifikoval dvoukrokovou. V prvním kroku se generují výstupy a výsledky se archivují jako tzv. artefakty. V kroku druhém se tyto artefakty stáhnou a nahrají na webový server. Díky tomuto rozdělení lze na GitLabu deploy opakovat bez

nutnosti znovu generovat všechny artefakty. To umožňuje mj. velmi rychlý rollback a obecně přesun mezi verzemi.

Implementoval jsem podporu pro GitLab review apps. Změny z větve `deploy/prod` se automaticky nahrávají do produkčního prostředí. Ostatní větve se nasadí pro dynamicky vygenerovaného prostředí a vývojář je má po kontrole možnost ručně vypnout.

Při použití `docker` executoru je pipeline stejně jednoduchá, ale odpadá nutnost předem nastavit prostředí a není nutné řešit kolize závislostí napříč projekty. GitLab dokáže artifacts ukládat z kontejneru stejně jako v `shell` executoru.

2.2.12.2 Projekt 2

Pipeline pro dynamický komplexní projekt je překvapivě podobná, jako u statického projektu. Opět jsem vytvořil dvě hlavní stage: build a deploy. Hlavní rozdíl je v samotných příkazech pro sestavení, které jsem abstrahoval do `Makefile`. Protože využívám `shell` executor, předinstaloval jsem PHP, balíčkovací systém composer a další. To je bohužel správa kterou je potřeba dělat mimo konfiguraci pipeline.

Review apps jsem navrhnul teoreticky, protože bez kontejnerů je implementace zbytečně složitá. U relační databáze (ve smyslu RDBMS) je potřeba vytvořit novou databázi, nakonfigurovat oprávnění pro nového uživatele aby nemohl ovlivnit ostatní (a hlavně produkční) databáze, nakonfigurovat aplikaci aby používala jiné databázové údaje a spustit migrace. Dále je potřeba po ukončení review app nějakým způsobem tuto databázi smazat. Velmi podobný postup je potřeba opakovat pro key-value storage, na které je aplikace také závislá.

2.2.12.3 Projekt 3

Opět jsem použil úplně stejnou pipeline jako pro předchozí dva projekty. Pro sdílení vystavěného docker obrazu jsem použil GitLab Container Registry, což je vestavěná služba. Registr je automaticky založený při vytvoření GitLab projektu. V build scriptu jsem provedl autentizaci `docker login`. Heslo jsem hardcodoval rovnou do scriptu. Alternativně může být nastaveno ve webovém rozhraní GitLabu a předáno do jobu jako proměnná prostředí (`env`). To má smysl hlavně pro veřejné projekty, kde nechceme heslo zveřejňovat; pak je ale nutné ohlídat, aby CI job nemohl kdokoliv škodlivě upravit, spustit, a nechat si vypsat heslo do logu.

Pro build aplikace se používá hostitelský docker démon. Celou řadu výhod a nevýhod a alternativní řešení jsem rozepsal v sekci o Dockeru 2.2.3.

Pro spuštění aplikace na webovém serveru se nejprve nahraje soubor pro konfiguraci Docker Swarm stacku. Na webovém serveru se pak spustí přihlášení do registru a příkazem `docker stack deploy` se spustí a případně aktualizuje aplikace.

U tohoto ukázkového projektu si použitím `docker` executoru moc nepomůžeme. Docker jako závislost musí být na hostitelském serveru předinstalovaný a pro zbytek se používají kontejnery.

2.3 Jenkins

Jenkins byl dřív známý jako Hudson a přejmenoval se po neshodě s Oracle [73]. Oba projekty pak nějakou dobu byly udržovány souběžně. Oracle svůj systém Hudson oficiálně nikdy nepřestal vyvíjet, ale poslední vydání je ze začátku roku 2016. V následujícím textu se budu věnovat pouze Jenkins, který je dodnes aktivně udržován a má velkou komunitu.

Hlavní výsadou Jenkins oproti ostatním CI/CD systémům je rozšiřitelnost pomocí pluginů.

Architektura Jenkins je master+agents [74]. Stavová master instance poskytuje API a webové GUI a koordinuje práci jednotlivých bezstavových agentů (workerů). Je zajímavé, že Jenkins nevyužívá tradiční relační databázi, ale persistuje data přímo na disk v formátu XML.

2.3.1 Instalace a konfigurace

Instalace Jenkins z oficiálního balíčku je přímočará. Nejprve jsem zaregistroval Jenkins APT repozitář `pkg.jenkins.io` a aplikaci nainstaloval. Jenkins ale odmítal nastartovat, protože mu chyběl Java runtime. Očekával bych, že aplikační balíček bude mít nezbytné závislosti minimálně v *suggested packages*. Jenkins v aktuální verzi podporuje pouze Java 8 [75]. To je LTS verze z března 2014, která má komerční podporu pouze do ledna 2019 [76].

Při prvním zobrazení webového rozhraní se provádí konfigurace a volitelná instalace rozšíření. Na rozdíl od např. GitLabu nebo Wordpressu vyžaduje Jenkins výchozí administrátorské heslo, které vygeneroval na disk.

V roce 2018 přišel Jenkins s možností nahradit ruční klikání konfigurace ve webovém rozhraní kódem (CasC) [77]. Některá klíčová nastavení, konkrétně třeba správa rozšíření, je zatím nestabilní. Dále je nahlášena celá řada nekompatibilit s různými rozšířeními `jenkins-casc-compat`.

Každý projekt je na Jenkins nutné založit a nakonfigurovat ručně. Některá rozšíření tento problém řeší, například `GitHub Branch Source Plugin` [78] sleduje všechny repozitáře vybraných GitLab uživatelů a pokud v nich najde `Jenkinsfile`, založí pro

```
pipeline {
    agent none
    stages {
        stage('Jekyll Build') {
            agent {
                docker 'composer:1.8'
            }
            steps {
                checkout scm
                sh 'make build'
            }
        }
        stage('Deploy') {
            agent {
                docker 'ditemikuthesisdemo/deploy:1.0'
            }
            steps {
                sh 'make deploy'
            }
        }
    }
}
```

Fig. 2.8: Ukázka definice deklarativní pipeline v `Jenkinsfile`. Pro správu závislostí se využívají se předpřipravané Docker obrazy. V prvním kroku se používá oficiální univerzální obraz, v kroku s nasazením je obraz vytvořený pro tento projekt.

repozitář nový projekt na Jenkins. Původně byly Jenkins projekty konfigurovatelné jenom z webového rozhraní a API. V roce 2014 byla publikován `Pipeline Plugin`, který umožňuje popsat konfiguraci projektu skriptem. Na to v roce 2016 navázalo rozšíření `Pipeline Model Definition Plugin`, které přináší podporu pro deklarativní popis pipeline, kde Jenkins říkáme co se má stát, ale ne nutně *jak*. Deklarativní pipeline je doporučený způsob nastavení Jenkins projektů [79].

Problém Jenkins je uživatelské rozhraní. Ve výchozím stavu má UI celou řadu velmi problematických částí: několika-úrovňové menu, které vyžaduje přesný pohyb myší; nesjednocené symboly, navíc často nekonvenční (například modrá koule u úspěšného jobu místo klasické zelené); konfigurace a všechny formuláře jsou složité a běžně obsahují 100+ vstupů. Rozšíření Blue Ocean nabízí alternativní UI pro zobrazení průběhu a výsledků buildů [80]. Pokud uživatelé konfigurují projekty výhradně přes `Jenkinsfile`, nemusí do původního rozhraní vůbec přistupovat. Osobně mi rozhraní přišlo hezké, ale oproti konkurenčním CI/CD pomalé.

2.3.2 Rozšiřitelnost

Jenkins má v oficiálním registru přes 1 500 rozšíření. Používaných je jich ale jenom zlomek; jak ukazuji na grafu 2.14 často instalovaných rozšíření je jenom kolem 200. Velmi překvapivé bylo zjištění, že rozšíření jsou často aktualizovaná. Z grafu 2.9b lze vidět, že přes 600 rozšíření mělo vydání v posledním roce.

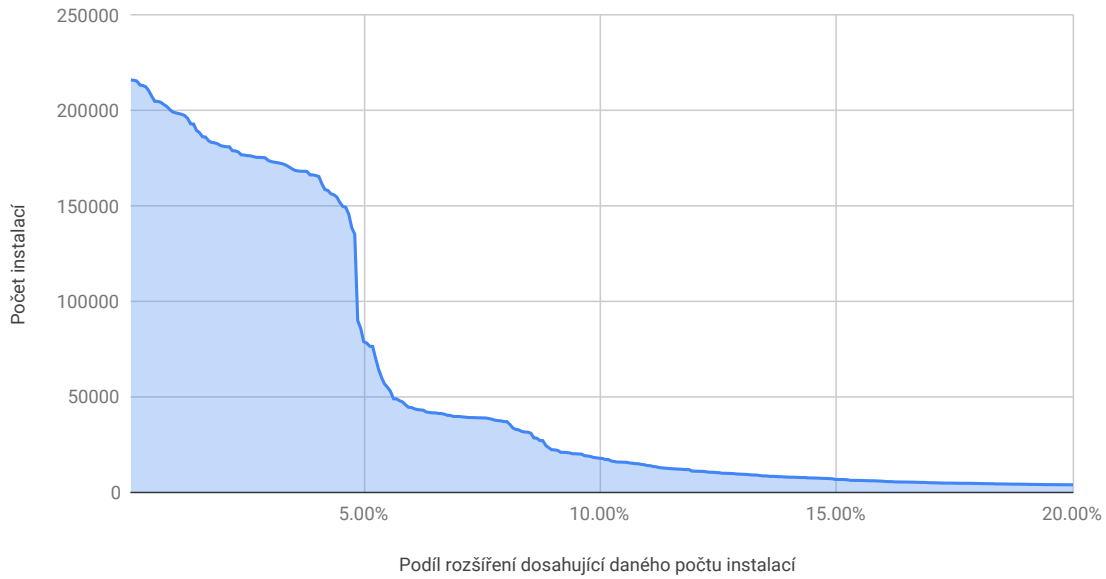
Celkově působí Jenkins velmi roztříštěně. Některá rozšíření mají dokumentaci na Jenkins Wiki, jiné na portálu Jenkins Plugins a zbytek na vlastních dedikovaných stránkách nebo na GitHubu. Část pluginů má nějaký obsah na všech těchto místech a dohledat konkrétní informace bývá obtížné. Velký problém je poznat, jaká rozšíření nainstalovat. Pipelines jsou pro moderní použití Jenkins nezbytné, ale administrátor to musí sám vyčíst z blogů a odkoukat od ostatních. Je na trhu prostor pro svéhlavou distribuci Jenkins, která by obsahovala *best-practice* rozšíření a konfiguraci.

Jenkins umožňuje upravit prakticky cokoliv. Na rozdíl od GitLabu, kde vývojář může konfigurovat pipeline a spouštět libovolné procesy, na Jenkins lze upravit samotné rozhraní. Šlo by například rozšířit Jenkins o stránku s přehledem nasazených prostředí, podobně jako má GitLab své Environments. Je pak ale na zvážení, jestli není praktičtější podobně velké úpravy vyčlenit do samostatné webové aplikace.

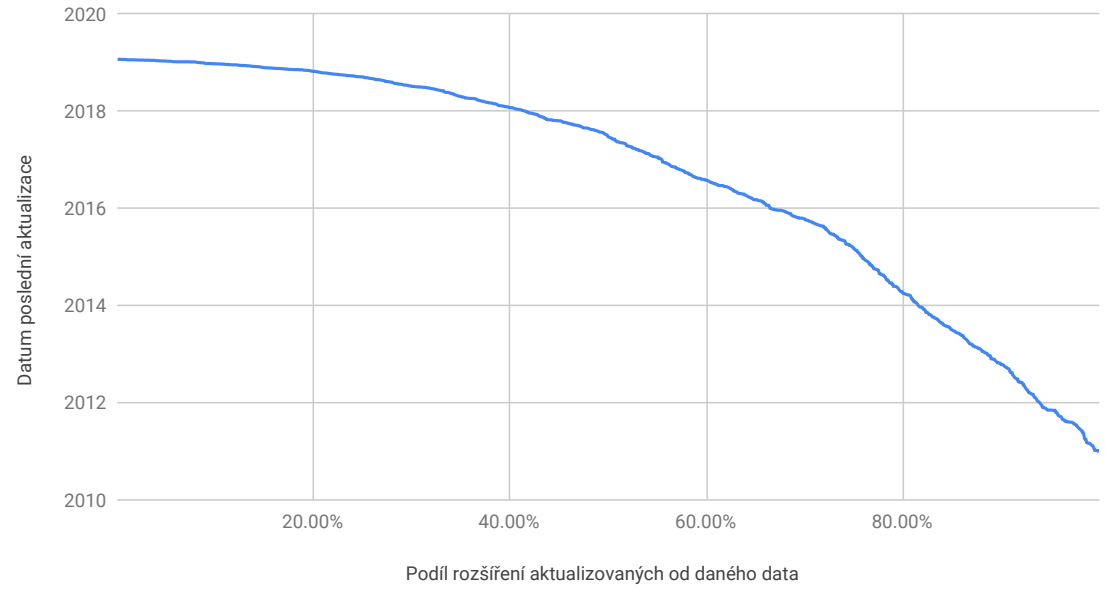
Kromě toho existují rozšíření, které umožňují spouštět skripty v rámci jobů. Například *Pipeline: Groovy* načte zdrojový kód umožňuje ho v jobu spustit. To bývá užitečné pro vyčlenění společné funkcionality napříč několika projekty a pro zpřehlednění pipeline.

2.3.3 Zabezpečení

Stejným způsobem kterým Jenkins deleguje funkcionalitu na pluginy, přenáší zodpovědnost i za bezpečnost. Jádro Jenkins mělo za rok 2018 nahlášeno 53 CVE, rozšíření pro Jenkins skoro 100 [81]. Čím víc rozšíření je do Jenkins nainstalováno, tím větší je plocha pro potenciální útočníky. Izolace klientů o něco horší než u GitLabu. Přestože oba systémy podporují různé úrovně izolace samotných jobů, nastavení práv pro webovou administraci Jenkins je velmi složité a nepřehledné. Tak jako všechno ostatní deleguje Jenkins i ACL na rozšíření. Základní rozšíření *Matrix-based security* umožňuje nastavit každému uživateli nebo skupině práva k nějaké akci. Jenom u zdroje *Job* je ale 9 oprávnění a není dobře zdokumentované, co přesně umožňují.



(a) Rozdělení počtu instalací. Oříznutých 80 % je klesající dlouhý ocas. Pouze zhruba 80 *rozšíření má víc instalací než* 100 000*. To jsou pluginy, které se nabízí administrátorům při první konfiguraci Jenkins. Necelých* 200 *rozšíření má víc než* 10 000 *instalací. Přes 60 % publikovaných rozšíření má méně než* 1 000 *instalací.*



(b) Rozdělení podle poslední aktualizace. Skoro 30 % byla aktualizována v posledním půl roce. 40 % rozšíření mělo alespoň jednu aktualizaci za poslední rok.

Fig. 2.9: Zdroj: data vytažena z `https://plugins.jenkins.io/`*, agregace a vizualizace vlastní. Data jsou dostupná na přiloženém mediu v* `appendix/jenkins-plugin-*.csv`*.*

Některé integrace, například GitHub, podle dokumentace umožňují dynamicky přiřazovat práva podle uživatelů přiřazených k danému repozitáři. Tuto funkci se mi ale nepodařilo zprovoznit; buď byl projekt úplně veřejný, nebo úplně nedostupný.

Izolace v rámci samotných agentů/workerů může být perfektní. Je dokonce i možné dynamicky reagovat na využití a zapínat a vypínat agenty v cloudu. Některá rozšíření umožňují využívat i krátkožijící virtuální stroje v cloudu jako je AWS CodeBuild [82] nebo GCP Cloud Build.

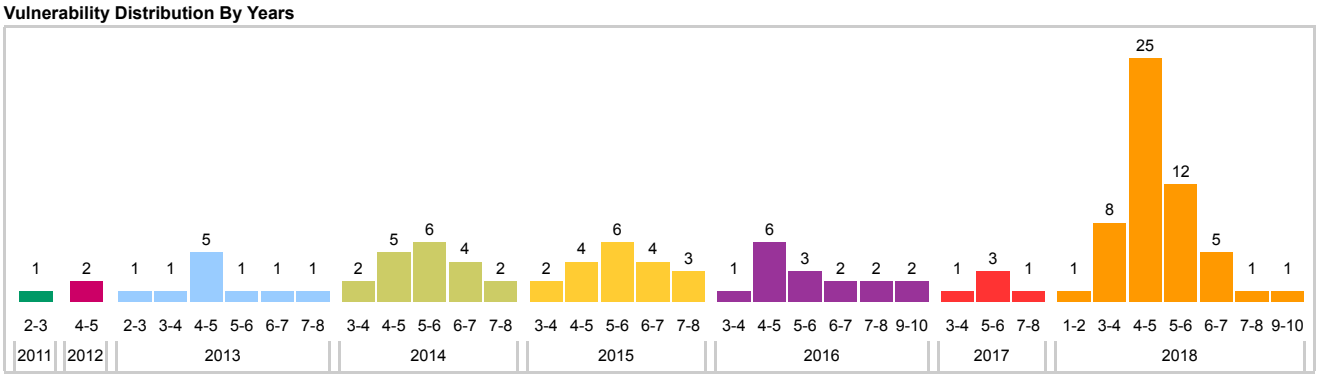


Fig. 2.10: Rozložení Jenkins CVE a přiřazené skóre podle CVSS. Většina nahlášených bezpečnostních chyb bylo XSS a únik informací. Nejvážnější problém v jádru v roce 2018 byla možnost neomezeného spouštění procesů na masteru, které mohl využít každý uživatel s právem přidat nový agent [81].

2.3.4 Dostupnost

Tím že Jenkins je výhradně ke spouštění jobů, dostupnost neřeší. Myšlenka je taková, že Jenkins periodicky skenuje různé repozitáře a při detekci změny začne nový job. V případě výpadku se pouze job zpozdí. Na rozdíl od GitLabu neposkytuje další vývojářům další funkce jako je repozitář verzovacího systému, správa úkolů a podobně. Přesto je výpadek Jenkins problematický i při každodenním užívání: projekty nemůžou využívat spouštění jobů z API a vývojáři nemůžou zobrazovat logy a využívat artifacts.

Při instalaci ale i při aktualizaci rozšíření vyžaduje Jenkins restart. Některá rozšíření lze nainstalovat i bez restartu, ale pro upgrade je vyžadován [83]. Nabízí možnost restartovat po dokončení všech právě běžících jobů, nebo manuální restart. V mém

testovacím prostředí trvá restart přes 30 vteřin. Stejně tak je nutný restart při upgrade celého jádra.

Jenkins je stavová aplikace persistující na disk. Není možné spustit víc replik a sdílet úložiště například přes NFS, protože Jenkins drží část informací jenom v paměti. Nejlepší dostupnosti lze dosáhnout použitím failoveru (*cold standby*). Je nutné zajistit, aby aktivní proces byl nejprve ukončen a persistoval všechny stavy na disk a až poté je možné zapnout standby instanci [84]. Ve výsledku tedy čekáme na vypnutí a zapnutí celé aplikace a doba výpadku je stejně dlouhá jako bez použití failoveru, ale infrastruktura pak není závislá na dostupnosti jednoho datacentra. Stojí za povšimnutí, že tuto formu failoveru získáme bez práce při použití libovolného orchestrátoru kontejnerů (Docker Swarm, Kubernetes, OpenShift, …).

Pokud se Jenkins restartuje, běžící joby se ztratí. V administraci a v API je možnost *safe restart*, která pozastaví frontu jobů, nechá už běžící joby doběhnout a poté systém restartuje.

Jelikož Jenkins nevyužívá databázi a všechna data persistuje na disk, je dostačují pravidelně vytvářet snapshot souborového systému (typicky dat v `/var/lib/jenkins`).

2.3.5 Integrace

Jenkins lze používat s libovolným externím repozitářem (GitHub/GitLab/Bitbucket/…) a pomocí vhodného rozšíření lze na danou službu i posílat oznámení o stavu. Jako jediný z porovnávaných systémů dokonce podporuje i jiné verzovací systémy (Mercurial, SVN, …) a dokonce je možné vytvořit pipeline která není na žádný repozitář vázaná.

Nasazení na cílové servery je mimo kompetenci Jenkins. Existují rozšíření, která přidávají Groovy funkce použitelné v Jenkins Pipeline, ale v porovnání s GitLab Jenkins nijak CD nepodporuje (v tom smyslu, že pokrývá pouze CI).

2.3.6 Praktické nasazení projektů

2.3.6.1 Projekt 1

Pro statický projekt jsem připravil pipeline využívající Docker. Kompilace Jekyll zdrojů potřebuje hodně závislostí a je nepraktické je instalovat přímo na hostitelské servery.

To jsem otestoval v sekci o GitLab 2.2.12.1. V Jenkins jsem ručně založil novou Pipeline a v nastavení jsem vybral možnost *Pipeline Script from SCM* – to umožňuje verzovat Jenkinsfile přímo v projektu. Alternativa je ručně psát pipeline přímo ve webové administraci Jenkins, což odporuje *Infrastructure as Code*. Dále jsem musel ručně zadat cestu k repozitáři. Narazil jsem na chybu v Jenkins, která znemožňuje v Pipeline klonovat lokální Git repozitáře. Tento problém jsem obešel uvedením vzdáleného repozitáře na GitLabu.

Samotná deklarativní pipeline není složitá. Uvedl jsem docker obraz, ve kterém se má kompilace provést. Dále jsem rozdělil práci na dvě stages: build a deploy. Stejně jako GitLab umí Jenkins opakovat jednu stage bez nutnosti spouštět znovu celou pipeline, což se hodí pro přechodné chyby, pro deploy, rollback a podobně. V samotných stages se pak spouští předpřipravené scripty, které využívám pro všechna testovaná CI/CD prostředí.

2.3.6.2 Projekt 2

Pro toto nasazení jsem rozdělil Jenkins Pipeline na samostatné kontejnery: ke každé stage je přiřazen odlišný Docker Agent, který používá přesně daný image. Tím jsou zamknuty externí závislosti, jednotlivé projekty se navzájem neovlivňují a není potřeba žádné knihovny a podpůrný software předinstalovávat na testovací server.

Při použití vlastních Docker obrazů pro build je vhodné, aby se vystavovaly v CI. V případě sdílení daného obrazu mezi více projekty může být úplně vyčleněn do samostatné pipeline.

2.3.6.3 Projekt 3

Při nasazování kontejnerizovaného projektu jsem využil DinD (vizte sekci 1.0.5). Narazil jsem na bezpečnostní opatření, kvůli kterému se kontejnery spouští se stejným UID jako samotný Jenkins. To je na jednu stranu dobré, protože root může z kontejneru uniknout snadněji než neprivilegovaný uživatel, na druhou stranu to část kontejnerů rozbíjí, protože ve svém `/etc/passwd` nemají záznam pro dynamicky přidělené UID. Uživatelům ale nic nebrání v Jenkinsfile definovat pro Agenta `args '-u root:sudo'`, čímž tuto ochranu obejdou.

2.4 Concourse

Concourse je minimalistické CI. Zjednodušeně řečeno obsahuje pouze tři základní koncepty: pipeline (projekt), job a přihlášení.

2.4.1 Architektura Concourse, možnosti konfigurace

Concourse využívá – podobně jako Jenkins a GitLab – architekturu kontrolerů (které nazývají *web node*) a pracovních uzlů (*worker node*). Na rozdíl od ostatních systémů ale má všechny části bezstavové. Data se persistují do externí PostgreSQL databáze. Díky tomu lze řídící rovinu libovolně škálovat a při provozu ve víc replikách má výbornou dostupnost.

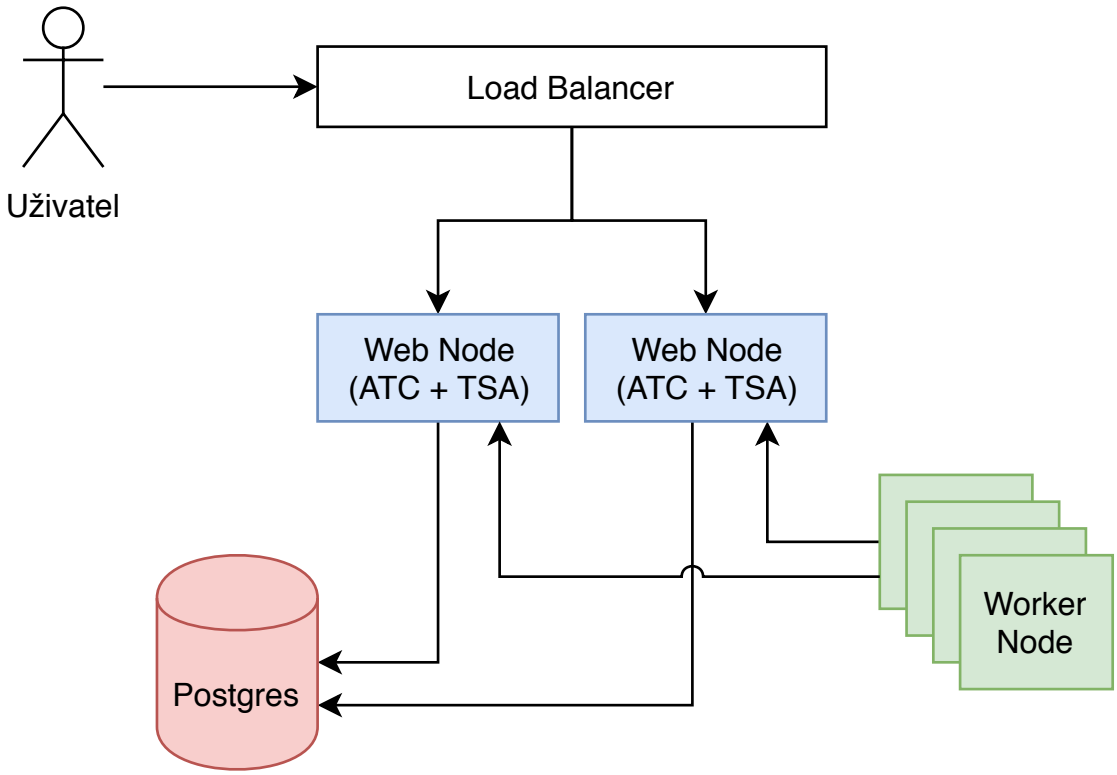


Fig. 2.11: Architektura Concourse je jednoduchá, přehledná a zároveň nabízí vysokou dostupnost. Červeně zvýrazněná PostgreSQL databáze je SPoF, ale díky dekompozici lze pomocí rolling update aktualizovat kontrolní rovinu bez výpadku.

```yaml
resource_types:
- name: rsync-resource
  type: docker-image
  source:
    repository: mrsixw/concourse-rsync-resource
    tag: latest
resources:
- name: repo
  type: git
  source:
    uri: "git@10.0.0.10:root/project-2.git"
    branch: master

- name: webhost
  type: rsync-resource
  source:
    server: 10.0.0.50
    base_dir: "/srv/p2"
    user: www-data
    disable_version_path: true


jobs:
- name: build
  public: true
  plan:
  - get: repo
    trigger: true
  - task: hello-world
    file: repo/concourse-task-build.yml
  - put: webhost
    params:
      sync_dir: .
```

```yaml
platform: linux
image_resource:
  type: docker-image
  source:
    repository: composer
    tag: 1.8
inputs:
  - name: repo
outputs:
  - name: vendor
run:
  path: composer install
```

Fig. 2.12: Ukázka definice pipeline v Concourse. První soubor je nutné ručně nahrávat na server. Druhý soubor je definice konkrétního tasku a lze ho verzovat v aplikačním repozitáři.

Nasazení Concourse je skvěle zdokumentované a průvodní texty dokonce popisují očekávané využití procesoru a paměti pro jednotlivé komponenty. Podobně jako u dalších distribuovaných systémů je potřeba před spuštěním jednotlivých komponent vygenerovat celou řadu klíčů. Na podepisování uživatelských session a interní komunikaci, identifikátor pro SSH server a pro každý *worker node* jeden klíč k registraci ke kontroleru. Další nastavení je volitelné `concourse-cluster`.

Pro rychlé otestování funkčnosti a drobné projekty nabízí Concourse zjednodušenou variantu spuštění. Jedním příkazem (`concourse quickstart`) se spustí kontroler a worker a automaticky se pro ně vygenerují a zaregistrují potřebné klíče.

Concourse neřeší balancování Web instancí. Je možné – a dokonce doporučené – mít víc než jednu repliku kontroleru, ale je na administrátorovi aby příchozí požadavky nějakým způsobem směroval, Concourse pouze na každé Web instanci vystavuje HTTP `concourse-cluster`. V Kubernetes, OpenShift nebo podobném orchestrátoru můžeme přímo využít vestavěné služby, pro jiné systémy se ale nasazení Concourse komplikuje přidáním dalšího software pro load balancing.

Worker instance musí být spuštěné pod root uživatelem. Podle firmy DigitalOcean je to proto, že worker potřebuje spravovat kontejnery `concourse-do`. To může v základním nastavení Dockeru ale libovolný linuxový uživatel v `docker` skupině [85]. I s přístupem k Dockeru ale Concourse worker odmítal nastartovat a stále vyžadoval root uživatele.

2.4.2 Rozšiřitelnost

Oproti Jenkins, kde se instalují pluginy, přistupuje Concourse k rozšiřitelnosti obráceně. Poskytuje několik základních zdrojů (pipeline, job) a API. Uživatelé můžou v rámci spuštěných kontejnerů volat cokoliv potřebují. Alternativní rozhraní a podobně je možné implementovat jako externí aplikaci. Concourse se snaží dělat jednu věc a dělat jí dobře.

Concourse lze částečně rozšířit přes koncept *resources* (zdroje). Ty se balíčkují jako kontejnery a jediný požadavek na ně je, že musí obsahovat programy `check`, `in` a `out` v cestě `/opt/resource` [86]. Podle konfigurace konkrétní pipeline se pak periodicky volá `check`, který má na za úkol vrátit seznam všech nových verzí. Volání `in` a `out` má za úkol dostat nějaké informace do Concourse pipeline, resp. je nahrát z pipeline ven. Příklad zdroje může být RSS, kde lze periodicky sledovat feed nějaké

závislosti a při vydání nové verze automaticky spustit pipeline. Jiný příklad je zdroj git, který může nejenom reagovat na nové verze, ale i udělat nějaké změny v repozitáři. Resource se může napojit na libovolné API a existují stovky opensource integrací na všechno od monitorovacích nástrojů po CD systémy jako je Kubernetes/Helm a řada dalších [87].

Protože některé `check` operace můžou být drahé a případně pomalé, přišel Concourse s možností definovat pro zdroje také token, kterým lze programově vynutit okamžité spuštění `check` z API. To je něco co by používala organizace pracující aktivně na několika málo z hodně repozitářů, kde je žádoucí spustit pipeline do pár vteřin od změny, ale není praktické s takovou periodou kontrolovat desítky repozitářů.

2.4.3 Integrace

Díky vysoké abstrakci v porovnání s ostatními CI systémy nemusí Concourse žádné integrace řešit. To je vhodné pro správce, kteří mají míň práce. Pro hodně systémů existuje nějaká open source knihovna, která přidává integraci pomocí Concourse resource. Na rozdíl od Jenkins ale neexistuje jednotný portál, na kterém by byly všechna rozšíření zveřejněna. Na GitHubu existují stovky rozšíření, ale drtivá většina je nepoužívaná a neznámá (má malé desítky hvězd; oblíbené a často používané projekty mají stovky hvězd, například Concourse samotný jich má přes $3{,}5$ tisíce) [87].

S tím kolik práce je nějaké funkční Concourse rozšíření najít a ověřit, je často jednodušší naskriptovat integraci v rámci pipeline. Integrace CI do jiných systémů by měla být co nejjednodušší a Concourse se svojí obecností v tomto ohledu strádá.

Výhoda Concourse zdrojů oproti vlastním skriptům je dekompozice. V rámci jedné organize je rozumné vyčlenit integrace používané ve víc projektech a pak je používat. Nejen v tomto ohledu je Concourse nástroj vhodný pro velké korporace.

2.4.4 Zabezpečení

Concourse má koncept týmů a přihlášení. V rámci jednoho týmu jsou ale všichni uživatelé oprávnění k čemukoliv. V rámci komunity je to známý a nevyřešený problém [88]. V roce 2018 vzniklo RFC které specifikuje podporu pro RBAC, ale implementace zatím nevznikla. Klasické řešení které se nyní používá je nasazení samostatných Concourse clusterů pro každý tým a přidělení přístupu jenom administrátorům.

Ve veřejné databázi CVE nemá Concourse moc záznamů [89]. Žádné nejsou kompletně zařazené a otagované a část z nich nemá dopočítané ani CVSS. Ani při hledání v repozitáři jsem nenašel žádné důležité nahlášené bezpečnostní problémy.

2.4.5 Dostupnost

Dostupnost Concourse za běžného provozu je perfektní. Díky oddělenému externímu úložišti a bezstavové kontrolní ploše lze Concourse provozovat ve vysoké dostupnosti a téměř libovolně škálovat. Ideální by bylo místo PostgreSQL používat nějakou distribuovanou HA databázi. V současném návrhu je databáze SPoF a nelze ji jednoduše ničím nahradit.

Díky tomu, že Concourse má veškerý stav v Postgre, stačí zálohovat databázi. To lze dělat atomicky a často [90].

Concourse lze upgradovat na novější verze bez výpadku pomocí rolling update kontrolní roviny.

2.4.6 Praktické nasazení projektů

Navzdory skvělé dokumentaci pro nasazení Concourse clusteru bylo pro mě vytváření pipeline a jejich správa utrpení. Na rozdíl od ostatních CI není kanonická verze pipeline v repozitáři, ale v Concourse. To souvisí s tím, že concourse je hodně abstraktní a nesoustředí se jenom na stavbu aplikací. Existují zdroje [91] které umožňují pipeline editovat, ale použití by reálně znamenalo vytvořit jednu pipeline která by byla „napevno“ nastavená v concourse a stahovala aktuální stav repozitáře, upravila by jinou pipeline podle specifikace a následně danou pipeline spustila. Výsledek je zbytečně komplikovaný a pomalý.

Pipeline se tak zakládají a upravují z konzole, ne úpravou konfigurace v repozitáři a pushnutí, takže systém nenutí vývojáře specifikaci pipeline verzovat.

Pro mě nejvíc zmatená část Concourse je koncept *task*. Tasky umožňují spustit část pipeline ze staženého resource (což může být repozitář, ale i cokoliv jiného). Lze tak drobnou část pipeline editovat jenom s přístupem k repozitáři, ale v tasku nelze nastavit definici resources, jejich stažení ani jejich nahrání. Tasky mají vstup a výstup, ale jejich integrace do pipeline se řeší mimo.

Po praktické zkušenosti s nasazením všech tří projektů mi toto řešení přijde míň vhodné, než mít kompletní definici pipeline v jednom souboru spravovaném přes konzoli.

2.4.6.1 Projekt 1

Při nasazení projektu jsem se potýkal s náhodnými problémy, které ostatní CI systémy neměly. Jedna z chyb byla například kryptická zpráva *volume graph is disabled*. To je známá chyba už z roku 2016 a je o tom, že v dokumentaci je uveden klíč `image` ale Concourse očekává `image_resource` [92]. Banalita, ale při kombinaci se špatnou zastaralou dokumentací a malou komunitou je to zdlouhavý problém k řešení. Další problém který jsme při implementaci měl bylo stahování Docker kontejnerů z veřejného repozitáře. Přestože jsem nejnovější Concourse spustil na čisté VM přesně podle dokumentace, bylo pravděpodobně nastaveno špatně DNS. Po ruční editaci nameserverů jsem skončil na další těžko rozluštitelné chybě `unknown handle: uuid`. Na to nepomohlo ani smazat celou pipeline a vytvořit ji znovu. Rozhodl jsem se smazat celou databázi a začít znovu. Pro testování je to bezproblémové řešení, ale Concourse mě tímto odpudil a v produkčním prostředí mu nedůvěřuji. Ještě na další problém jsem narazil po restartu, kde se nějakým záhadným způsobem poškodil stažený Docker obraz a Concourse končil chybou, ale nestáhnul obraz znovu.

Pro kompilaci projektu a nahrání výsledků na web server jsem využil dva zdroje. Prvního zabudovaný zdroj `git` sleduje repozitář projekty a při detekci změny spustí zbytek pipeline. Následně se v jobu repozitář naklonuje a v rámci tasku se spustí kompilace. Výsledné vystavěné soubory se pak posílají do zdroje `rsync-resource`, který je definovaný externím Docker obrazem. Přijatá data nahraje přes rsync na server.

V ukázkových definicích jsem vědomě zapsal soukromé klíče přímo v holém čitelném textu. Concourse podporuje různé systémy pro předávání tajemství a popisuji je v sekci 2.4.4. Nasazení těchto produktů a jejich použití bylo ale mimo rozsah této práce.

2.4.6.2 Projekt 2

Ukázkový projekt 2 jsem bez obtíží a rychle nasadil. Pro testy vyžadující databázi bych využil integraci s Docker Compose [93], díky které lze v jednom jobu spustit několik navzájem síťově provázaných kontejnerů.

2.4.6.3 Projekt 3

Pro kontejnery má Concourse speciální typ zdroje, který pro uživatele kompletně abstrahuje interní implementaci DinD. Na rozdíl od předchozích dvou ukázek, které v pipeline načítaly soubor z repozitáře definující kroky pro kompilaci, pro kontejnery toto není potřeba. Všechny verze a kroky kompilace jsou uvedeny v `Dockerfile`. Na rozdíl od ostatních CI (GitLab, Jenkins) je v Concourse kompilace Docker kontejnerů dokonce jednodušší, než ostatní typy operací.

2.5 Drone

Drone je minimalistické CI postavené na kontejnerech. Je to svým způsobem jenom Docker orchestrátor, který má navíc drobné webové rozhraní. V UI lze dělat pouze dva úkony: číst výstup jobů a zapínat/vypínat sledování repozitářů. Kromě toho je celý web implementovaný jako SPA. To má sice pozitivní vliv na rychlost přepínání stránek, ale při testování občas aplikace nereagovala a několikrát se zasekla úplně. Dokonce ve webové administraci není ani seznam agentů, joby čekající na zpracování a podobně.

Některé důležité funkce skrývá Drone za placenou licenci. Nejde ani o korporátní podporu, ale o nezbytné vlastnosti bez který lze CI těžko provozovat. Mezi ně patří: dynamický runner, který by umožňoval využívat cluster nebo externí služby (například AWS CodeBuild) podle vytížení; sdílení tajemství napříč organizací a obecně podpora pro externí správce tajemství; šablony pro pipeline.

2.5.1 Architektura Drone, možnosti konfigurace

Dokumentace Drone je dostatečně obsáhlá, ale není vyčerpávající a odkazy jsou navíc netypicky pojmenované. Instalace je rozdělena podle externího správce úložišť, tradičně bývá odlišná dokumentace pro způsoby instalace (bare metal, kontejnery, …). Není možné v jedné Drone instanci využívat zároveň například GitLab a Bitbucket, podporováno je pouze jedno úložiště. Dokumentační stránka o instalaci na Kubernetes je dokonce úplně špatně. Doporučuje spouštět Drone server jako `Pod` (mělo by jít o `Deployment`, nebo Helm chart) a agenti dokonce naprosto chybí. Další problém dokumentace je, že používá ve všech ukázkách Docker obrazy bez tagu. To je špatně, protože výchozí tag `latest` se může kdykoliv změnit na nekompatibilní verzi. Vždy je lepší explicitně uvést tag.

Stejně jako ostatní CI se Drone skládá z jednoho masteru a agentů. Master persistuje data do sqlite databáze, ale některá data o frontě požadavků udržuje pouze v paměti [94]. Nemá žádné externí závislosti.

```yaml
kind: pipeline
name: default

steps:
- name: build
  image: composer:1.8
  volumes:
  - name: vendor
    path: "/drone/src/vendor"
  commands:
  - make build

- name: deploy
  image: eeacms/rsync:1.2
  environment:
    SSH_KEY:
      from_secret: deploy-ssh-key
  volumes:
  - name: vendor
    path: "/drone/src/vendor"
  commands:
  - mkdir -p "/root/.ssh"
  - echo "$SSH_KEY" | base64 -d > "/root/.ssh/id_rsa"
  - chmod go-rwx "/root/.ssh/id_rsa"
  - apk add --no-cache bash make
  - ssh www-data@webserver-plain whoami
  - make deploy

volumes:
- name: vendor
  temp: {}

---
kind: secret

# cat src/cookbooks/ssh-ring/files/id_rsa | base64 | xargs drone encrypt root/project-1
data:
  deploy-ssh-key: 1psu0UpA...zkráceno
```

Fig. 2.13: Ukázka definice pipeline v `.drone.yml`. Opět jsou definovány dva lineární kroky: kompilace a nasazení. Oproti jiným CI se zde artefakty předávají explicitně definovaným `volume`. V druhém dokumentu je definován privátní klíč, který by se v ideálním případě měl načítat z externího úložiště, například HashiCorp Vault.

2.5.2 Rozšiřitelnost

Drone rozlišuje dva koncepty. První jsou *pipeline plugins*, což jsou obyčejné kontejnery spuštěné v pipeline. Jediný rozdíl je drobné syntaktické zjednodušení, které umožňuje

v konfiguraci pipeline psát místo `environment[PLUGIN_KEY]=x` jenom `settings.key=x`. Kdyby Drone tuto funkci neměl, byla by tvorba pluginů transparentnější a přístupnější i těm nejméně zkušeným uživatelům.

Druhý koncept rozšíření – dostupný pouze v placené Enterprise verzi – upravuje nějakým způsobem definici pipeline. Jediné zdokumentované rozšíření je zatím Jsonnet, které umožňuje používat stejnojmenný šablonovací jazyk místo YAML [95].

2.5.3 Zabezpečení

Díky kontejnerové architektuře má Drone základní izolaci i v rámci jednoho agenta. Lepší izolace lze dosáhnout v placené verzi, která nabízí dynamické agenty, které můžou spouštět VM nebo nějakým způsobem využívat cloud.

Vynikající vlastnost kterou ostatní v základu CI nenabízejí je možnost zašifrovat tajemství a umístit ho do veřejně čitelné pipeline. Využívá se k tomu asymetrické šifrování, při kterém CI nikdy nezveřejňuje svůj privátní klíč. Dál je podporován mj. HashiCorp Vault a AWS Secrets Manager.

Při napojení na GitHub je vyžadováno moc práv, včetně čtení a zápisu do všech soukromých repozitářů. Je na to otevřené issue od začátku roku 2018, ale Drone je zatím bez opravy [96].

Nenašel jsem žádná Drone CVE, ani jsem nenašel žádné zdokumentované bezpečnostní problémy ve veřejném seznamu chyb a úkolů celé Drone organizace (ať otevřené, nebo vyřešené).

2.5.4 Dostupnost

Podobně jako ostatní CI má Drone jeden stavový master, který není možné load balancovat. Jeden z autorů prohlásil, že na stabilním cloudu má Drone dostupnost 99,999 % [94]. Není ale možné, aby zahrnoval i aktualizace. Kromě toho s nástupem smýšlení *Infrastructure as Code* není běžné udržovat dlouhožijící *pet* servery.

Stejně jako Concourse persistuje i Drone data do databáze (rozdíl je v tom, že Concourse má master nestavový). Stačí tak zálohovat standardními prostředky které podporuje vybraný RDBMS. Podporované systémy jsou MySQL a Postgres [97].

Dále je nutné zálohovat objektové úložiště, které může být volitelně používáno pro ukládání výstupu pipeline.

Drone není dostatečně aktivně vyvíjený projekt a má teprve verzi 1.0. Roky existoval ve verzích 0.x a vznikl zatím nekompletní oficiální nástroj na migraci databáze z původní verze 0.8 na aktuální 1.0 [98]. Vzhledem k tomu, že Drone vývojáři kromě tohoto vydání databázové schéma neupravují, je možné rychle aktualizovat na minor a patch verze a případně i dělat rollback. Nelze ale udělat rolling update a je pro aktualizaci nutné akceptovat krátký výpadek služby.

2.5.5 Integrace

Drone podporuje jenom vybrané správce repozitářů: GitHub, Gitlab, Bitbucket, Gitea a Gogs. Důležité je, že neexistuje podpora pro obecný git remote.

Ani přes úzkou integraci na repozitáře není ale výsledná integrace bezchybná. Například oznámení výsledku pipeline na GitHub (tzv. *Commit Status* nebo novější *Checks*) nejsou zabudované a je nutné použít plugin, což dále komplikuje pipeline kterou uživatel musí napsat a udržovat.

2.5.6 Praktické nasazení projektů

2.5.6.1 Projekt 1

Pro statický projekt jsem jako u ostatních CI vytvořil dvoukrokovou pipeline, kde první část kompiluje zdrojové soubory nástrojem Jekyll a druhá část nahrává výsledný web na webový server pomocí `rsync`. Protože všechny kroky běží v kontejneru a nelze použít zdroje na hostitelském stroji, místo přednahrání RSA klíče je nutné ho umístit do pipeline. Aby byl dostupný pouze pro CI a nemohl ho nikdo jiný zneužít, je zašifrovaný CLI nástrojem `drone encrypt`. Tato utilita nepodporuje načítání z `stdin`, takže binární, víceřádkové nebo obecně složitější data je nutné nějakým způsobem překódovat. V tomto případě jsem klíč nejprve překódoval s `base64`. Toto řešení není ideální, protože vyžaduje práci v kontejneru, kde dekódovací nástroj nemusí být nainstalovaný.

Další problém na který jsem při tvorbě pipeline narazil byla chybějící indikace stavu pipeline ve webovém rozhraní. Při prvním dlouhém stahování docker obrazu pro

danou pipeline vypadá Drone rozbitě/zaseknutě. Chybí jakákoliv indikace postupu nebo dokonce času k dokončení.

Přes všechny počáteční nesnáze je konfigurace Drone pipeline relativně snadná a přímočará. Především tomu pomáhá koncept *Volumes*, což definuje složku souborů sdílející se mezi všemi joby. Ostatní CI toto řeší mnohem složitěji, například přes read-only *artifacts*.

2.5.6.2 Projekt 2

Implementoval jsem opět dvoukrokovou pipeline: kompilaci a deploy. Narazil jsem na drobný problém při přejmenování použitého volume kterým se předávají data mezi jednotlivými kroky. Drone nijak nevaruje, když název disku v kroku neodpovídá žádnému klíči name v poli `volumes` a pouze ho tiše vytvoří. Bylo by vhodné uživatele o chybě informovat, protože v lepším případě je definice pipeline nekonzistentní a v horším případě úplně nefunkční, což nastane když když dva kroky místo sdílení dat využívají chybně jiné disky.

2.5.6.3 Projekt 3

Stejně jako u Concourse je i u Drone kompilace kontejnerizované aplikace výrazně jednodušší. Využil jsem oficiální rozšíření `drone-docker`, které pro uživatele abstrahuje DinD. Bohužel je to velmi naivní implementace, díky které nefunguje Docker cache a všechny buildy jsou výrazně pomalejší, než musí být. Je ale možné místo tohoto oficiálního rozšíření použít vlastní dedikovaný Docker daemon. Toto je detailně popsáno v sekci 1.0.5.

2.6 GoCD

Systém GoCD vychází z projektu Cruise [99]. Oba projekty vznikly za podpory firmy ThoughtWorks, kde pracoval průkopník a zastánce praktik *extrémního programování* M. Fowler [28]. Fowler systém Cruise doporučoval ve známém článku o CI [3].

2.6.1 Architektura GoCD, možnosti konfigurace

GoCD stejně jako většina ostatních CI staví na architektuře jednoho kontrolního serveru a řadě interních nebo externích agentů, které se starají o spuštění jednotlivých jobů. Ve výchozím nastavení jsou data persistována v embedované H2 databázi na server procesu. Instalace GoCD je díky tomu jednoduchá, protože stačí zaregistrovat externí repozitář a nainstalovat balíček pro server a pro agenty.

Po prvním zobrazení webového rozhraní serveru se zobrazuje quick-start, který provádí vytvořením nové pipeline. Přestože GoCD podporuje *Pipelines as code*, očekávaný primární vstup je klikání v administraci [100]. Načítání externích souborů je vyřešeno rozšířením, jehož použití se definuje v hlavní XML konfiguraci na webu. Separovaná konfigurace tak není kompletní a závisí na další ruční konfiguraci.

2.6.2 Rozšiřitelnost

GoCD nabízí několik možností, jak rozšiřovat výchozí funkcionalitu pomocí pluginů [101]. Na rozdíl od Jenkins, kde lze upravit prakticky cokoliv, vystavuje pro pluginy GoCD jenom některá API. Nelze tak například ovlivňovat uživatelské rozhraní.

Zhruba 80 pluginů je na oficiálním registru [102]. ThoughtWorks Inc. zaštiťují 8 pluginů a a z toho je 6 placených.

Z veřejně dostupných rozšíření jsou všechna kromě jednoho hostována na GitHub. Jak jsem ale vizualizoval v grafu 2.14, GoCD rozšíření jsou ještě hůř udržovaná než ta pro Jenkins. A to je jich méně než dvacetina.

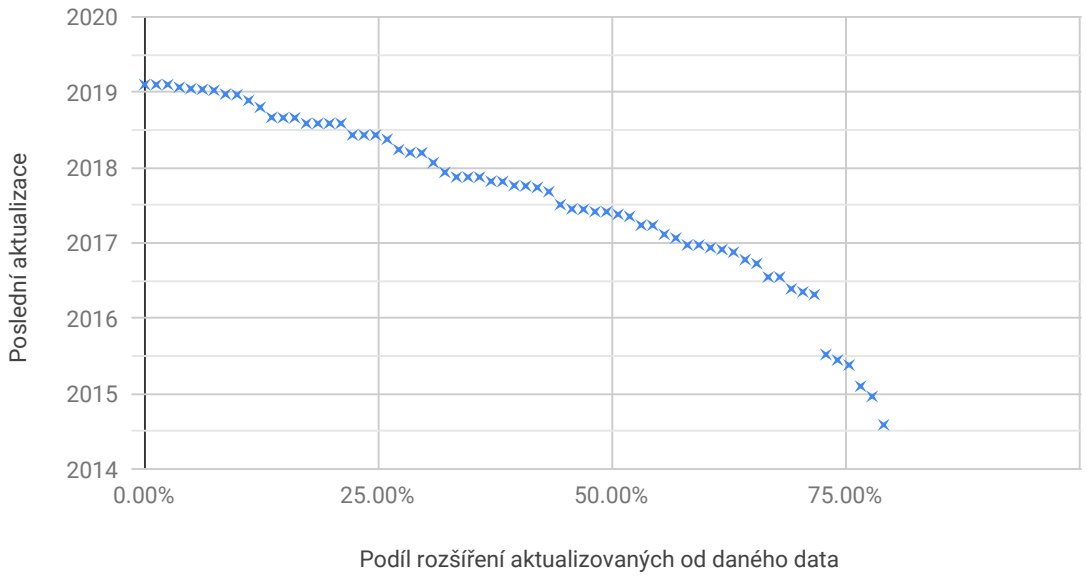


Fig. 2.14: Rozdělení podle poslední aktualizace. Pouze 17 rozšíření (20 %) bylo aktualizováno v posledním půl roce. Pouhých 30 % rozšíření mělo alespoň jednu aktualizaci za poslední rok. Přibližně 20 % rozšíření nemá žádné stabilní vydání. Zdroj: data vytěžena z GitHub repozitářů, dostupná na přiloženém mediu v `appendix/gocd-plugins.csv`.

2.6.3 Zabezpečení

Po instalaci serveru je v základu celá administrace dostupná pro všechny, bez autorizace. Lze zapnout zabudovaná rozšíření, která zprovozní přihlášení heslem a nebo přes LDAP. Nic v UI k tomu ale správce nenabádá. Velmi překvapivě ale není v Shodan databázi žádná nezabezpečená instance GoCD na výchozím portu [103].

Izolace klientů závisí na použitých agentech. Klasický agent používá sdílené prostředí a zdroje, ale GoCD podporuje tzv. *Elastic Agent*, které se zapínají a vypínají dynamicky podle poptávky. Lze je využít pro zapnutí nového prostředí v Dockeru, Kubernetes, OpenStacku a několika dalších. Tato jednorázová prostředí nabízí lepší izolaci. Nenašel jsem rozšíření, které by dokázalo spravovat Elastic Agents pro VMware nebo jiný virtualizační nástroj, který by měl ještě lepší izolaci klientů.

Na webu CVE Details jsem pro GoCD nenašel žádná historická CVE. Projekt využívá platformu HackerOne, kde za poslední tři roky zpracovali 39 nahlášených chyb [104].

Ze zveřejněných reportů je vidět, že správci reagují rychle a bezpečnostní chyby opravují v co nejkratším možném termínu.

2.6.4 Dostupnost

GoCD agenti jsou z principu stavové aplikace a stejně jako u všech ostatních CI jejich výpadek způsobí, že přijdeme o spuštěné joby. Může jich ale běžet mnoho a pomocí rolling update je lze aktualizovat bez výpadku.

Server je bohužel také stavový a může běžet pouze v jedné replice. GoCD prodává velmi drahý *Business Continuity Addon*, který dokáže udržovat standby repliku [105]. Failover proces ale není nijak automatizovaný a povýšení na primární repliku vyžaduje restart GoCD serveru. U GoCD nelze udělat perfektní HA bez ztráty žádného požadavku když vypadne primární replika.

Upgrade serveru na novější verzi vyžaduje restart a tím pádem nedostupnost celého prostředí. Navíc se při startu nové verze aplikace automaticky spustí databázové migrace, které na větších instancích podle dokumentace mohou trvat přes 10 minut [106]. Celkově je tak při rutinní aktualizaci potřeba počítat s výpadkem minimálně 15 minut.

GoCD má jednoduchý proces pro vytvoření zálohy, který lze spustit buď z UI nebo z API [107]. Záloha obsahuje dump databáze, vyklikané XML konfigurace aplikace a konfigurace repozitářů, nastavení webového serveru a klíče. Nejsou zálohovány historie a výstupy spuštěných úloh, ani nainstalovaná rozšíření! Proces vytváří archiv na disku aktuálního serveru, který je poté dobré zmigrovat na nějaké externí úložiště.

2.6.5 Integrace

Lze propojit GoCD s dalšími nástroji; některé funkce – především ty co mají vliv na UI – jsou vestavěné. To je například odkazování na externí správu úkolů podle regulárního výrazu. Ostatní integrace jsou dostupná jako rozšíření, která je nutné doinstalovat: mezi ně patří například oznámení stavu na GitHub/Stash/Gerrit a podobně. Pro Bitbucket ani Gitlab podpora neexistuje. Lze ji ale doprogramovat jako nové rozšíření.

```xml
<?xml version="1.0" encoding="utf-8"?>
<pipeline name="p2">
  <materials>
    <git url="file:///src/p2-dynamic/" shallowClone="true" />
  </materials>
  <stage name="BuildStage">
    <jobs>
      <job name="BuildJob">
        <tasks>
          <exec command="make">
            <arg>build</arg>
            <runif status="passed" />
          </exec>
          <exec command="make">
            <arg>deploy</arg>
            <runif status="passed" />
          </exec>
        </tasks>
      </job>
    </jobs>
  </stage>
</pipeline>
```

Fig. 2.15: Přestože GoCD je primárně určeno ke konfiguraci z GUI, lze editovat přímo konfigurační XML soubory. Vzhledem k chybějící izolaci úkonů v tomto CI byla tato pipeline navržena jako prosté spuštění dvou příkazů.

Navzdory svému jménu nenabízí GoCD žádnou podporu pro continuous deployment. Ve webovém rozhraní (ani jinde) není možnost spravovat nasazená prostředí a verze aplikací, podpora pro ručně potvrzené kroky je velmi limitovaná, a integrace s Kubernetes/OpenShift nebo jiným nebo neexistuje.

2.6.6 Praktické nasazení projektů

2.6.6.1 Projekt 1

Nasazení GoCD pro statický projekt bylo vesměs stejné, jako u GitLab shell executoru: na server jsem předinstalovat Ruby a v rámci pipeline se pak pouze volá instalace Ruby gems a samotná kompilace. Narazil jsem na drobný problém při použití RVM (správce Ruby verzí), který dynamicky upravuje `$PATH`, ale GoCD tato nastavení nerespektoval. Musel jsem tak ručně cestu k Ruby upravit v souboru `.profile` a restartovat GoCD agent aby se změny projevily.

2.6.6.2 Projekt 2

Pro druhý ukázkový projekt byla implementace v GoCD velmi jednoduchá, protože jsem závislosti předinstaloval při tvorbě prostředí. Pro víc projektů je to nepraktické řešení, ale pokud bude v systému vždy pouze jeden projekt využívající dané knihovny a podpůrný software, nenarazíme na skoro žádné problémy. Obtížnější az neřešitelné může být testování zároveň několika verzí závislostí (například současná stabilní verze a nejnovější nestabilní verze).

2.6.6.3 Projekt 3

Při implementaci CI pro kontejnerizovaný projekt jsem narazil na to, že GoCD neinterpoluje proměnné prostředí. Je tak nutné explicitně volat `sh -c "$CMD"`, což je zbytečná komplikace bez jasných výhod. Dále jsem se potýkal s tím, že GoCD nedefinuje běžné proměnné prostředí. Konkrétně `CI_COMMIT_SHA` je dostupné jen pod názvem `GO_REVISION`. Přes tyto nedostatky poskytuje GoCD možnosti pro kompilaci a nasazení kontejnerizované aplikace.

2.7 SaaS: CircleCI, Semaphore CI, Travis CI

V této sekci shrnu výhody a nevýhody moderních SaaS CI. Vyzdvihnu významné rozdíly, pokud na nějaké narazím, ale primárně budu popisovat CircleCI, Semaphore CI a Travis CI dohromady.

2.7.1 Architektura SaaS CI a možnosti konfigurace

Travis CI a Semaphore CI mají prakticky z pohledu uživatele prakticky stejnou architekturu. Pro každý job zapnou samostatný virtuální stroj. Travis CI poměrně překvapivě přešel v roce 2019 kompletně na virtuální stroje; dříve umožňoval spouštět i Docker kontejnery, což využívalo 45 % všech jobů [108]. Travis CI jako důvod uvádí složitější kompilace Docker obrazů pomocí DinD. To ale může znamenat, že pro firmu je to dražší řešení na podporu, ne nutně že jde o lepší řešení pro uživatele. V rámci virtuálního stroje má uživatel veškerou volnost a může instalovat a spouštět vesměs cokoliv. V základním obrazu je předinstalovaná celá řada často používaných nástrojů a runtime ve spoustě verzí. Velmi praktická funkce Travis CI, kterou ostatní CI nástroje v základu nemají, je *Build Matrix*: uživatel specifikuje různé verze různých závislostí a CI pak spustí job pro *všechny* kombinace [109]. Některé kombinace lze navíc označit jako volitelné a jejich selhání je jenom informační. To se hodí pro předběžné testování nestabilních RC verzí závislostí a podobně.

CircleCI naopak v roce 2018 zmigroval všechny uživatele z virtuální strojů (CircleCI 1.0) na čistě kontejnerové prostředí (CircleCI 2.0) [110]. V specifikaci pipeline uživatel uvádí všechny kontejnery které chce spustit a jejich prolinkování/pořadí. Umí také spustit některé služby paralelně a na pozadí, což se používá například pro databáze a podobné závislosti.

Semaphore CI kombinuje architekturu dvou předchozích řešení: jednotlivé joby běží ve virtuálních strojích, ale jsou provázané přes koncept bloků a data si předávají přes cache. Oproti CircleCI chybí možnost zrychlit přípravu prostředí a předinstalaci závislostí Docker obrazem a přitom je na Semaphore CI definice pipeline výrazně složitější, než na Travis CI.

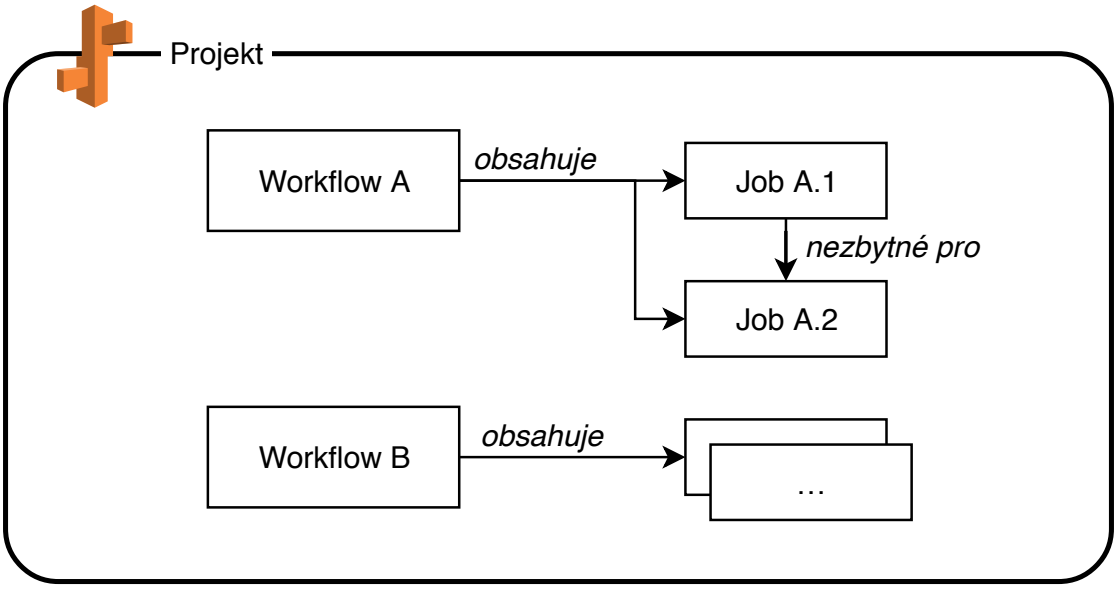


Fig. 2.16: Architektura CircleCI. Každý projekt může mít několik workflows, uvnitř které je libovolný souvislý acyklický graf jobs.

2.7.2 Zabezpečení

Ani jeden z těchto SaaS CI systémů nenabízí bug bounty. Pro Travis CI jsem našel jednu zprávu o bezpečnostní chybě z roku 2018 [111]. Pro CircleCI ani Semaphore CI jsem žádné zveřejněné incidenty nenašel.

2.7.3 Dostupnost

CircleCI, Travis CI ani Semaphore CI nedefinují žádné SLA. Za posledních 12 měsíců měl CircleCI uptime 99.90 % [112], Travis CI 99.93 % [113]. Semaphore CI reportuje za poslední rok podezřele vysoký uptime 100 % [114].

Kromě samotných CI služeb jsou ale tyto služby závislé na dostupnosti úložišť kódu (GitHub, GitLab, Bitbucket, …).

2.7.4 Rozšiřitelnost

Pro SaaS řešení nelze uvažovat systémy pluginů a rozšíření. Veškerá funkcionalita služby musí být přímo integrovaná v systému. Tyto možnosti popisuji v následující podsekci.

```yaml
language: php
php:
 - "7.2"

script:
- make build
- make deploy
```

```yaml
version: v1.0
name: P2 Build and Deploy
agent:
  machine:
    type: e1-standard-2
    os_image: ubuntu1804

blocks:
  - name: Build
    task:
      jobs:
        - name: composer
          commands:
          - checkout
          - phpbrew switch php-7.1
          - phpbrew ext install iconv
          - sudo apt-get install -y --no-install-recommends composer
          - cache restore "vendor-$SEMAPHORE_GIT_BRANCH"
          - make build
          - cache store "vendor-$SEMAPHORE_GIT_BRANCH"

  - name: Deploy
    task:
      jobs:
        - name: deploy
          commands:
            - checkout
            - cache restore "vendor-$SEMAPHORE_GIT_BRANC"
            - make deploy
```

Fig. 2.17: Porovnání funkčně identického Travis CI (první soubor) a Semaphore CI (druhý soubor). Přestože oba systémy nabízí stejné funkce a izolaci, je konfigurace Travis CI podstatně jednodušší.

2.7.5 Integrace

Travis CI lze používat pouze s repozitáři na GitHub, CircleCI a Semaphore CI podporují kromě toho také Bitbucket. Nelze používat vlastní repozitáře a jiné systémy. Teoreticky lze vytvořit vlastní obálku nad cizím API a posílat webhooky ve formátu jako GitHub, ale nejde o oficiálně podporovanou variantu.

Semaphore CI má složité zakládání nového repozitáře. Na rozdíl od ostatních dvou CI je nutné nainstalovat na klientu spustitelný program, který podporuje pouze Linux a macOS a instaluje se přes `curl|bash`. Poté se v repozitáři musí zavolat `sem init`, který ale funguje pouze pokud má projekt nastavený `origin remote` na GitHub. Běžně stačí projekt přidat v administraci CI: díky propojení na GitHub CI ví, jaké projekty existují.

Všechny tři CI podporují GitHub perfektně a i nově zveřejněné funkce implementují rychle.

2.7.6 Praktické nasazení projektů

2.7.6.1 Projekt 1

Ze všech CI vyzkoušených v této práci bylo nasazení na Travis CI suverénně nejjednodušší. Na deseti řádcích se přehledně definují všechny závislosti a volá se build. I nezaškolený uživatel by dokázal vytvořit novou pipeline podle minimální ukázky.

Na CircleCI je konfigurace pipeline znatelně složitější. Rozdělením na kontejnery se ale separují jednotlivé závislosti a správa komplexních projektů by byla jednodušší. Další výhoda CircleCI pro tento projekt je možnost opakovat pouze jenom dílčí kroky a ukládat mezivýsledky do cache, která se může použít při každém spuštění. Toho jsem využil pro instalaci závislostí z Gemfile. Alternativou bylo vytvořit nový Docker obraz a Ruby gemy tam předinstalovat. To má ale řadu nevýhod, předně to zvyšuje komplexitu a bylo by složitější použít jiné gemy (jiná rozšíření pro Jekyll).

Pro Semaphore CI jsem de facto musel zkombinovat obě předchozí řešení: v VM jsem nechal nainstalovat Ruby gemy a uložil je do cache. V druhém jobu se cache stáhne a spustí se kompilace samotného statického webu.

Na rozdíl od ostatních nasazení jsem při testování SaaS neimplementovat celou CI pipeline včetně nahrání na cílový server. Všechny systémy jsem zprovoznil v lokálním virtuálním prostředí na které není vhodné dělat vzdálený přístup. Místo `make deploy` je tak v definicích pouze job s hláškou, kde by se deploy spouštěl.

2.7.6.2 Projekt 2

Pro druhý ukázkový projekt jsem upravil konfigurace vytvořené pro statickou aplikaci. Pro CircleCI a Travis CI jsem finální funkční pipeline vytvořil na první pokus. Pro Semaphore CI bylo z dokumentace nutné nastudovat, jaké obrazy pro VM jsou dostupné a případně jaké jsou poskytované nástroje jsou instalaci a konfiguraci závislostí. Konkrétně pro PHP je předinstalován nástroj phpbrew. Dále jsem musel ladit nastavení cache složky `vendor` ve které jsou závislosti nainstalované nástrojem `composer`, především aby fungovalo sdílení dat mezi krokem pro kompilaci a pro nasazení.

2.7.6.3 Projekt 3

Kompilace Docker obrazu byla snadná na všech třech testovaných SaaS CI. Ve všech případech byla konfigurace krátká a výstižná. Travis CI a Semaphore CI, které jsou založené na virtuálních strojích, byly stejně jednoduché na konfiguraci jako CircleCI, který je založený na kontejnerech a konfiguruje externí Docker daemon.

2.8 GitHub Actions

GitHub je SaaS a největší vývojářská platforma [115]. Překvapivě od svého založení v roce 2008 neměl zabudované žádné CI. Podporoval napojení na externí služby přes webhooks a až v roce 2012 přidali tzv. *commit status API*, které umožnilo CI zapsat výsledek pipeline zpět do GitHubu. V roce 2018 GitHub toto přejmenoval na *check runs API*, ale funkcionalita zůstala podobná.

V roce byla 2019 zveřejněna úplně nová funkcionalita: GitHub Actions. Jde o obecný systém, který reaguje na různé GitHub události (nový commit, změna issue, deploy aplikace, ...) a spouští Docker kontejnery. Nepodporuje žádné složitější koncepty jako jsou služby na pozadí (například databáze jako závislost pro aplikační testy).

V době psaní práce (první čtvrtletí 2019) jsou Actions v beta verzi a zuřivě se vyvíjí. Nebudu zkoumat konkrétní detaily, ale zaměřím se hlavně na vysokoúrovňový pohled a jak Actions zapadají do stávajícího ekosystému.

Actions mohly úplně nahradit služby třetích stran jako jsou Travis, CircleCI, Semaphore a další. Spíš se ale dá očekávat, že se budou Actions s dalšími CI nějakým způsobem kombinovat. Testy jednoúčelové a aplikovatelné na libovolný repozitář jsou pro Actions vhodné. Například statické otestování bash skriptů pomocí shellcheck [116]. Aplikační testy na byznys logiku bude jednodušší spravovat v komplexnějším CI.

2.8.1 Architektura GitHub Actions, možnosti konfigurace

Workflow se konfigurují souborem `.github/main.workflow` ve formátu HCL. GitHub také nabízí grafický editor, který konfigurační soubor generuje a ukládá do repozitáře. Samotná konfigurace obsahuje právě jeden blok `workflow`, ve kterém se definuje API událost která má workflow spustit. Zatím není možné spustit pipeline v reakci na víc než jednu událost.

Zbytek workflow je acyklický graf docker kontejnerů (`action`) ke spuštění. Vývojář může změnit výchozí entrypoint a argumenty kontejneru, a definuje, na jaké jiné actions musí počkat. Kontejnery si mohou předat informace souborem na disku. Z dokumentace není jasné, jestli mají souběžně běžící kontejnery přístup ke stejnému disku a musí případně řešit zamykání, nebo jestli se data předávají jednorázově a můžou vzniknout kolize [117].

Kromě veřejných Docker obrazů dokáže GitHub kompilovat a spouštět i obrazy definované souborem Dockerfile v repozitáři. Navíc skvěle využívá koncept Docker štítků (*labels*) pro definici metadat. Není tak potřeba jeden registr a Actions jsou úplně decentralizované [118]. To dokonce umožňuje reimplementovat proprietární tenkou Workflow vrstvu, jako například vznikla v `nektos/act` [119].

2.8.2 Zabezpečení

GitHub jako SaaS přejímá skoro všechnu zodpovědnost za bezpečnost. Actions umožňují volat libovolné kontejnery a na vývojáři tak zůstává jenom kontrola Docker obrazů. V nejhorším případě kompromitovaný Docker obraz může při spuštění v pipeline ukrást aktuální stav repozitáře a může číst všechna definovaná tajemství, což budou typicky tokeny do služeb třetích stran.

2.8.3 Dostupnost

GitHub nedefinuje žádné SLA pro většinu placených plánů. Pro nejdražší korporátní plán nabízí SLA 99,95 %.

2.8.4 Praktické nasazení projektů

GitHub Actions jsou v neveřejné beta verzi a neměl jsem možnost vytvořit pipeline pro ukázkové projekty.

2.9 Souhrn

	GitLab 11.7	Jenkins 2.150	GoCD 19.1	Drone 1.0.0-rc.5	Concourse 4.2.2
Homepage	`about.gitlab.com`	`jenkins.io`	`gocd.org`	`drone.io`	`concourse-ci.org`
SaaS	zdarma, placené	třetí strany (CloudBees)	ne	zdarma	ne
Self-hosted	zdarma, placené	zdarma	zdarma	zdarma, placené	zdarma
Dostupnost masteru	SPoF (může existovat víc masterů)	SPoF	SPoF	SPoF	HA
Možnosti izolace	žádná, kontejner, VM, fyzická			kontejner, fyzická	kontejner, fyzická
Joby	host, kontejner (podle executoru)	host, kontejner (podle pipeline)	host	kontejner	kontejner
Rozšiřitelnost	ne	všechno, včetně UI	vybraná místa	minimální (na úrovni jobů)	
Složitost pipeline	2	8	5	2	7
UX	1	8	7	4	9
Integrace	GitLab, vestavěné	z rozšíření		ne (řeší na úrovni jobů)	

Tabulka na této dvojstraně vizualizuje silné a slabé stránky porovnávaných CI nástrojů. Zeleně podbarvené buňky reprezentují nejlepší hodnoty.

S výjimkou GitLab jsou všechna CI buď proprietární SaaS nebo zdarma, open-source a self-hosted. GitLab má odlišnou strategii a prodává licence jak pro hostovanou, tak i pro self-hosted variantu (GitLab EE).

Dostupnost kontrolních serverů ma suverénně nejlepší Concourse, který umožňuje provozovat více masterů. GitLab může mít zaregistrováno více masterů (runnerů), ale každý se stará o svoji skupinu pipeline. Dostupnost agentů není uvedena. U všech CI jsou joby stavové a z principu není možné je replikovat.

	CircleCI 2.0	Travis CI	SemaphoreCI	GitHub Actions beta
Homepage	`circleci.com`	`travis-ci.org`	`semaphoreci.com`	`github.com/actions`
SaaS	zdarma, placené	zdarma open-source, placené	zdarma, placené	zdarma (GitHub tarif)
Self-Hosted	ne	ne	ne	ne (jen komunitní implementace)
Dostupnost masteru	99.90%	99.93%	100%	zatím neaplikovatelné
Možnosti izolace	kontejner (mezi organizacemi VM)	VM	VM	kontejner (mezi organizacemi VM)
Joby	kontejner	host	host	kontejner
Rozšiřitelnost	ne	ne	ne	ne
Složitost pipeline	9	1	6	4
UX	2	2	3	4
Integrace	GitHub, Bitbucket	GitHub	GitHub	GitHub

Dále jsou CI klasifikované podle úrovně izolace. Všechny self-hosted řešení nabízí izolaci kontejnery (ať už je přímo zabudovaná, je k dispozici pomocí rozšíření, nebo ji lze implementovat na úrovni samotného jobu). Díky master+agent architektuře lze také dosáhnout fyzické izolace, kde různí agenti běží na odlišných serverech. U CI která nejsou postavená čistě na kontejnerech lze dosáhnout i žádné izolace. GitLab, Jenkins a GoCD ještě umožňují vytvářet dynamicky VM. U SaaS řešení která staví na kontejnerech není úplně jasné, jakou izolaci nabízí, a není to veřejně dostupná informace. Dá se ale očekávat, že uživatelé/organizace budou na oddělené VM.

V řádku *Joby* jsou porovnány možnosti konfigurace. U možnosti *host* běží všechny příkazy na jedné VM a typicky nelze opakovat jenom části pipeline, ale konfigurace bývá intuitivnější. Oproti tomu varianta *kontejner* označuje CI, která rozdělují pipeline na různé části, kde každá se spouští v předem vytvořeném Docker kontejneru.

Přestože jsem do metodiky zahrnul rigorózní testování dostupnosti, byly získané výsledky binární: aplikace jsou buď navrženy distribuovaně a mají tedy pro všechny úkony během správy perfektní dostupnost, nebo je nutné je vypnout a jsou tedy kompletně nedostupné. Při měření času nedostupnosti bych tak pouze porovnával jak dlouho aplikace startují. Místo toho jsem naměřené výsledky promítnul v tabulce jako *Dostupnost masteru*.

Složitost pipeline a UX jsou čistě subjektivní relativní hodnocení. Škála je od 1 do 9, kde 1 je nejlepší skóre.

2.9.1 Použití jednotlivých CI

GitLab Skvělé CI. Podporuje jen repozitáře na GitLab. Dobrá podpora CD díky integraci s Kubernetes.

Jenkins Nejobecnější CI s nejhorším UX. Použil bych až jako poslední možnost, pokud narazím na limitace jiných CI.

GoCD Funkčně stejné nebo horší než Jenkins, má menší komunitu a skoro není udržovaný. Má nevýznamně lepší UX.

Drone Minimalistické CI. Cena stejná jako u GitLab, za řádově míň funkcí. Využil bych pro cloud-ready organizace, které pracují výhradně s kontejnery a mají repozitáře jinde než na GitLab.

Concourse Alternativa ke Drone, má výrazně složitější konfiguraci a horší UX. Největší výhoda Concourse je vynikající dostupnost kontrolní roviny a cena.

CircleCI SaaS varianta Drone. Limitující faktor je cena. V rozhodování můžou hrát roli izolace a rychlost.

Travis CI SaaS varianta CI nezaloženého na kontejnerech.

Semaphore CI Alternativa k Travis CI s komplikovanější konfigurací, za lepší cenu.

GitHub Actions Novinka, pravděpodobně bude používané jako doplněk k dalším CI.

Kapitola 3

Nasazení ve firmě

Proces kontinuálního nasazení jsem implementoval ve firmě manGoweb, s.r.o. Jde o středně velkou firmu věnující se primárně webovým a mobilním aplikacím. Kromě programátorů budou na denní bázi s CI systémem interagovat i manažeři. Mezi používané webové technologie patří: PHP, JS a NodeJS a různé kompilátory (TypeScript, Webpack, Babel, ...), všechny běžné preprocesory pro CSS. Pro mobilní aplikace by byla vhodná podpora pro iOS (Swift) a Android (Java, Kotlin), ale to není nutnou podmínkou pro výběr CI/CD systému, který musí primárně vyhovovat nárokům webových aplikací. Firma má řádově stovky repozitářů a řada z nich je i několik let starých. Především u frontend vývoje je běžné, že nástroje vydávají zpětně nekompatibilní verze a není praktické aktualizovat všechny projekty. U každého projektu by měl být strojově čitelný seznam s verzemi všech závislostí.

Nutné požadavky firmy byly:

- Prostředí testů musí být kvalitně izolované. Aplikace se musí stavět s původními verzemi závislostí, tak jak mají uvedeno v repozitáři.

- Systém musí mít dobrou integraci s Kubernetes clustery provozovanými v AWS, na kterých se provozují beta i produkční verze všech aplikací.

- Uživatelské rozhraní musí být přehledné a naprosto jasné i pro juniory. S předchozím systémem měla firma často problém, že vývojáři nepoznali, zda integrace a potažmo vydání nové verze aplikace skončilo úspěšně, nebo chybou.

- CI/CD systém musí být dobře dostupný. Je nutné počítat s tím, že vývojáři mohou pracovat do noci. Několik programátorů je delší dobu v cizině, s časovým posunem +6 hodin oproti ČR.

Na základě porovnání v předchozí kapitole 2.9 jsem se rozhodl nasadit CI/CD systém GitLab, konkrétně tedy kombinaci GitLab pro správu repozitářů a GitLab Runner využívající Docker executor (jak je popsáno v v sekci 2.2.3). Kontejnerizace umožňuje izolaci a lze tak každou aplikaci stavět s potřebnými verzemi závislostí. Jenkins jsem po důkladnějším testování zavrhl, protože výstup nebyl pro uživatele dostatečně přehledný a vývojáři měli problém se v rozhraní orientovat. CircleCI, Travis CI, Semaphore CI a Drone jsme po diskuzi s vlastníky firmy zavrhli kvůli ceně.

Přestože GitLab nabízí SaaS variantu, využili jsme self-hosted verzi a existující infrastrukturu firmy. Implementoval jsem nasazení jako IaC pomocí nástroje Terraform. Na jedno spuštění se v cloudu (AWS) vytvoří nové instance, vytvoří se Kubernetes cluster a pomocí Helm se poté nasadí GitLab a GitLab Runner. Pro kompilaci Docker kontejnerů jsem nasadil dedikovaný sdílený sec:dind **??** ontejner, což byl vhodný kompromis mezi rychlostí – funkční Docker cache zrychluje následné kompilace, bezpečností a cenou.

Nejprve jsem nasadil GitLab pomocí Omnibus kontejneru. Toto řešení bylo ale těžké udržovat, debugovat a škálovat. Později jsem nasazení předělal na oficiální Helm Chart, který v Kubernetes spouští všechny komponenty jako samostatné kontejnery. GitLab tak lze snadněji integrovat do existujícího ekosystému, což obnáší především logování (syslog a Papertrail) a metriky (Prometheus a Grafana).

Při rutinní aktualizaci GitLabu na verzi 11.8 jsem narazil na to, že oficiální kontejnery používají starou verzi Ruby, nekompatibilní se samotnou aplikací. Tento problém dokonce dostal nejvyšší prioritu a vážnost [120]. Později se ukázalo, že ačkoliv je Helm Chart oficiální a podporovaná cesta pro nasazení GitLabu na Kubernetes, neexistují zatím automatické testy, které tento problém mohly zachytit [121]. Po tomto incidentu jsem upravil aktualizační proces. Aktualizuji pouze na předposlední minor verzi (tedy například při vydání verze 11.11 aktualizuji teprve z verze 11.9 na 11.10). GitLab vydává bezpečnostní záplaty na tři poslední minor verze. Tento postup oddaluje nasazení nové funkcionality výměnou za vyšší stabilitu.

Do aplikace GitLab jsem zmigroval všechny existující repozitáře a s využitím šablon jsem pro všechny projekty připravil CI konfiguraci. Automatizoval jsem nasazení

na beta a produkční Kubernetes cluster pomocí vlastních Helm Charts. Firmě se subjektivně zkrátila doba běhu všech pipeline a celkově prý působí CI robustnější než bylo. Vývojáři už nemají problém se v CI orientovat a jsou odstíněni od nasazení aplikací, které je tak méně citlivé na lidskou chybu.

Závěr

Náplní mé diplomové práce byl průzkum veřejně dostupných řešení pro CI/CD. Porovnával jsem GitLab, Jenkins, Concourse, Drone, GoCD, CircleCI, Semaphore CI a Travis CI. Pro všechny jsem kódem vytvořil virtuální stroj a aplikaci jsem nainstaloval a nakonfiguroval nástrojem Chef. Pro každý z těchto osmi systémů jsem vytvořil tři různé specifikace CI: pro statický web, pro aplikaci s komplexními závislosti a kontejnerizovanou aplikaci. Tím jsem prakticky vyzkoušel možnosti daných systémů. Předem stanovenou metodikou jsem jednotlivé systémy ohodnotil. Díky velké rozmanitosti nástrojů nelze žádný vyzdvihnout nad všechny ostatní.

GitLab je veřejně dostupné self-hosted CI s nejvíc funkcemi, důrazem na bezpečnost a kvalitním uživatelským rozhraním. Drone je minimalistický nástroj, vhodný pokud už máme zavedený nějaký systém na správu repozitářů a úkolů. Jenkins je přes svoji popularitu ve všech kategoriích zastíněn jinými CI/CD.

Nasadil jsem systém pro podporu CI/CD ve firmě manGoweb, s.r.o. Sepsal jsem požadavky a podle nich vybral GitLab jako nejvhodnější a ekonomicky nejsmysluplnější řešení. Vytvořil jsem automatizaci, díky které firma může nasazovat aplikace z CI systému do beta a produkčních Kubernetes clusterů.

V průběhu psaní práce vznikla řada zajímavých CI/CD systémů, které by bylo vhodné prozkoumat. Jedním z nich je například projekt Tekton Pipelines, který využívá primitiv z Kubernetes. Po oficiálním vydání a stabilizaci bude jistě také zajímavé znovu ohodnotit GitHub Actions, které jsou v době psaní ve veřejné beta verzi.

Bibliografie

1. BEYER, B.; JONES, C.; PETOFF, J.; MURPHY, N.R. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Incorporated, 2016. ISBN 9781491929124. Dostupné také z: https://landing.google.com/sre/sre-book/toc/index.html.
2. SHORE, James. *Continuous Integration is an Attitude, Not a Tool* [online]. 2005 [cit. 2018-12-21]. Dostupné z: www.jamesshore.com/Blog/Continuous-Integration-is-an-Attitude.html.
3. FOWLER, Martin. *Continuous Integration* [online]. 2006 [cit. 2018-12-21]. Dostupné z: https://martinfowler.com/articles/continuousIntegration.html.
4. FISHMAN, David M. Application Availability: An Approach to Measurement [online]. 2000 [cit. 2018-12-22]. Dostupné z: http://www.tarrani.net/linda/docs/ApplicationAvailability.pdf.
5. WILSON, Brian. *Backblaze Durability is 99.999999999% – And Why It Doesn't Matter* [online]. 2018 [cit. 2018-12-22]. Dostupné z: https://www.backblaze.com/blog/cloud-storage-durability/.
6. THIBODEAU, Patrick. *Booted up in 1993, this server still runs – but not for much longer* [online]. 2017 [cit. 2018-12-22]. Dostupné z: https://www.computerworld.com/article/3162416/data-center/booted-up-in-1993-this-server-still-runs-but-not-for-much-longer.html.

7. AMAZON WEB SERVICES, INC. *Amazon Compute Service Level Agreement* [online]. 2018 [cit. 2019-02-13]. Dostupné z: https://aws.amazon.com/compute/sla/.

8. GOOGLE INC. *Google Compute Engine Service Level Agreement (SLA)* [online]. 2018 [cit. 2019-02-13]. Dostupné z: https://cloud.google.com/compute/sla.

9. MICROSOFT CORPORATION. *SLA for Virtual Machines* [online]. 2018 [cit. 2019-02-13]. Dostupné z: https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/v1_8/.

10. DI LUCCA, G. A.; FASOLINO, A. R.; FARALLI, F.; DE CARLINI, U. Testing Web applications. In: *International Conference on Software Maintenance, 2002. Proceedings.* 2002, s. 310–319. ISSN 1063-6773. Dostupné z DOI: 10.1109/ICSM.2002.1167787.

11. THOMAS, D.; HUNT, A. Mock objects. *IEEE Software*. 2002, roč. 19, č. 3, s. 22–24. ISSN 0740-7459. Dostupné z DOI: 10.1109/MS.2002.1003449.

12. OULD, Martyn A; UNWIN, Charles. *Testing in software development*. Cambridge University Press, 1986. Dostupné také z: https://books.google.cz/books?id=utFCImZOTEIC&printsec=frontcover.

13. LEUNG, H. K. N.; WHITE, L. A study of integration testing and software regression at the integration level. In: *Proceedings. Conference on Software Maintenance 1990*. 1990, s. 290–301. Dostupné z DOI: 10.1109/ICSM.1990.131377.

14. BIDELMAN, Eric; GIANNINI, Steren. *Introducing headless Chrome support in Cloud Functions and App Engine* [online]. 2018 [cit. 2019-02-13]. Dostupné z: https://cloud.google.com/blog/products/gcp/introducing-headless-chrome-support-in-cloud-functions-and-app-engine.

15. JOHNSON, Pierr. *With The Public Clouds Of Amazon, Microsoft And Google, Big Data Is The Proverbial Big Deal* [online]. 2017 [cit. 2019-02-13]. Dostupné z: https://www.forbes.com/sites/johnsonpierr/2017/06/15/with-the-public-clouds-of-amazon-microsoft-and-google-big-data-is-the-proverbial-big-deal/.

16. MASTER INTERNET, S.R.O. *Živý server: Jediný cloud s férovým účtováním* [online]. 2019 [cit. 2019-02-13]. Dostupné z: https://www.master.cz/zivy-server/.

17. T-MOBILE, Czech Republic a.s. *Plánovaná odstávka* [online]. 2018 [cit. 2018-12-22]. Dostupné z: https://www.t-mobile.cz/odstavka.

18. PAJURKOVÁ, Jitka; O2 Czech Republic a.s. *O2 vylepšuje systémy, chystá proto odstávku online samoobsluhy Moje O2* [online]. 2018 [cit. 2018-12-22]. Dostupné z: https://www.o2.cz/spolecnost/tiskove-centrum/616524-O2_vylepsuje_systemy_chysta_proto_odstavku_online_samoobsluhy_Moje_O2.html.

19. LEACH, Paul J; BERNERS-LEE, Tim; MOGUL, Jeffrey C; MASINTER, Larry; FIELDING, Roy T; GETTYS, James. Hypertext Transfer Protocol–HTTP/1.1 [online]. 1999 [cit. 2018-12-22]. Dostupné z: https://tools.ietf.org/html/rfc2616#section-9.1.2.

20. STETSON, Chris. *Implementing the Circuit Breaker Pattern with NGINX Plus* [online]. 2016 [cit. 2019-03-06]. Dostupné z: https://www.nginx.com/blog/microservices-reference-architecture-nginx-circuit-breaker-pattern/.

21. TARREAU, Willy; HAProxy Technologies. *The PROXY protocol Versions 1 & 2* [online]. 2010 [cit. 2018-12-22]. Dostupné z: https://www.haproxy.org/download/1.8/doc/proxy-protocol.txt.

22. HANSEN, Robert. *Top 3 Security Proxy Issues That No One Ever Told You* [online]. 2013 [cit. 2018-12-22]. Dostupné z: https://www.whitehatsec.com/blog/top-3-proxy-issues-that-no-one-ever-told-you/.

23. PETERSSON, Andreas; NILSSON, Martin. *Forwarded HTTP Extension* [online]. 2014 [cit. 2018-12-22]. Dostupné z: https://www.rfc-editor.org/rfc/rfc7239.txt. Technická zpráva.

24. KAISER, Gail; PERRY, Dewayne; SCHELL, W.M. Infuse: fusing integration test management with change management. In: 1989, s. 552–558. ISBN 0-8186-1964-3. Dostupné z DOI: 10.1109/CMPSAC.1989.65147.

25. FOWLER, Martin. *Continuous Integration (original version)* [online]. 2000 [cit. 2018-12-21]. Dostupné z: https://martinfowler.com/articles/originalContinuousIntegration.html.

26. CHEN, L. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software*. 2015, roč. 32, č. 2, s. 50–54. ISSN 0740-7459. Dostupné z DOI: 10.1109/MS.2015.27.

27. SHAHIN, M.; BABAR, M. A.; ZAHEDI, M.; ZHU, L. Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges. In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2017, s. 111–120. Dostupné z DOI: 10.1109/ESEM.2017.18.

28. FOWLER, Martin. *Open-Sourcing ThoughtWorks Go* [online]. 2014 [cit. 2018-12-21]. Dostupné z: https://martinfowler.com/articles/go-interview.html.

29. SARAI, Aleksa. *CVE-2019-5736: runc container breakout (all versions)* [online]. 2019 [cit. 2019-02-13]. Dostupné z: https://www.openwall.com/lists/oss-security/2019/02/11/2.

30. WICHERS, Dave; Open Web Application Security Project. *OWASP: Source Code Analysis Tools* [online]. 2019 [cit. 2019-02-13]. Dostupné z: https://www.owasp.org/index.php/Source_Code_Analysis_Tools.

31. PETKOV, Nikolay Dimitrov; Open Web Application Security Project. *OWASP: Source Code Analysis Tools* [online]. 2019 [cit. 2019-02-13]. Dostupné z: https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools.

32. ABELA, Robert; Netsparker. *An Automated Scanner That Finds All OWASP Top 10 Security Flaws, Really?* [online]. 2017 [cit. 2019-02-13]. Dostupné z: https://www.netsparker.com/blog/web-security/owasp-top-10-web-security-scanner/.

33. COPES, Flavio. *Should you commit the node_modules folder to Git?* [online]. 2018 [cit. 2018-12-19]. Dostupné z: https://flaviocopes.com/should-commit-node-modules-git/.

34. FARINA, Matt. *Should Go Projects Vendor Dependencies?* [online]. 2015 [cit. 2019-03-06]. Dostupné z: https://codeengineered.com/blog/2015/go-should-i-vendor/.

35. ANDRAWOS, Mina; HELMICH, Martin. *Cloud Native Programming with Golang: Develop microservice-based high performance web apps for the cloud with Go*. Packt Publishing Ltd, 2017.

36. WILLIAMS, Chris. *How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript* [online]. 2016 [cit. 2018-12-19]. Dostupné z: https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/.

37. RUSNAČKO, Ján. Fedora Security Team Secure Ruby Development Guide. 2014. Dostupné také z: https://doc.fedoraproject.org/en-US/Fedora_Security_Team/1/pdf/Secure_Ruby_Development_Guide/Fedora_Security_Team-1-Secure_Ruby_Development_Guide-en-US.pdf.

38. GITHUB INC. *The State of the Octoverse: Projects: Languages* [online]. 2018 [cit. 2019-01-12]. Dostupné z: https://octoverse.github.com/projects#languages.

39. ELLINGWOOD, Justin; DigitalOcean. *CI/CD Tools Comparison: Jenkins, GitLab CI, Buildbot, Drone, and Concourse* [online]. 2017 [cit. 2019-01-12]. Dostupné z: https://www.digitalocean.com/community/tutorials/ci-cd-tools-comparison-jenkins-gitlab-ci-buildbot-drone-and-concourse.

40. FULMER, Jeff. *Siege: an http load tester and benchmarking utility* [online]. 2018 [cit. 2019-01-12]. Dostupné z: https://github.com/JoeDog/siege.

41. THE APACHE SOFTWARE FOUNDATION. *ab - Apache HTTP server benchmarking tool* [online]. 2019 [cit. 2019-01-12]. Dostupné z: https://httpd.apache.org/docs/2.4/programs/ab.html.

42. MCCOY, David. *How to Submit an HTML Form to Google Sheets...without Google Forms* [online]. 2017 [cit. 2018-12-19]. Dostupné z: https://medium.com/@dmccoy/how-to-submit-an-html-form-to-google-sheets-without-google-forms-b833952cc175.

43. PRESTON-WERNER, Tom; Jekyll contributors. *Jekyll: Transform your plain text into static websites and blogs* [online]. 2018 [cit. 2018-12-29]. Dostupné z: https://jekyllrb.com/.

44. GRUBER, John. *Markdown* [online]. 2018 [cit. 2018-12-29]. Dostupné z: https://daringfireball.net/projects/markdown/.

45. POTENCIER, Fabien. *Symfony, High Performance PHP Framework for Web Development* [online]. 2018 [cit. 2018-12-29]. Dostupné z: https://symfony.com/.

46. SPEICHER, Robert; GitLab contributors. *GitLab Installation: Ubuntu* [online]. 2019 [cit. 2019-01-10]. Dostupné z: https://about.gitlab.com/install/#ubuntu.

47. MOUNTNEY, DJ; GitLab contributors. *GitLab Helm Chart* [online]. 2018 [cit. 2019-01-20]. Dostupné z: https://docs.gitlab.com/ee/install/kubernetes/gitlab_chart.html.

48. DÍTĚ, Mikuláš; GitLab contributors. *GitLab Issues: External Redis without password is neither supported nor documented* [online]. 2018 [cit. 2019-01-20]. Dostupné z: https://gitlab.com/charts/gitlab/issues/980.

49. MAAN, Douwe; GitLab contributors. *Configuring GitLab Runners* [online]. 2019 [cit. 2019-01-10]. Dostupné z: https://docs.gitlab.com/ee/ci/runners/.

50. MAAN, Douwe; GitLab contributors. *Configuration of your jobs with .gitlab-ci.yml* [online]. 2019 [cit. 2019-01-10]. Dostupné z: https://docs.gitlab.com/ee/ci/yaml/README.html.

51. PIPINELLIS, Achilleas; GitLab contributors. *GitLab Runner: Configuring GitLab Runner: Advanced configuration* [online]. 2019 [cit. 2019-01-10]. Dostupné z: https://docs.gitlab.com/runner/configuration/advanced-configuration.html#the-executors.

52. GROVES, Daniel; GitLab contributors. *Docker Artifact caching MVC* [online]. 2017 [cit. 2019-01-14]. Dostupné z: https://gitlab.com/gitlab-org/gitlab-runner/issues/1107.

53. PATEL, Anand. *Distributing Docker Cache across Hosts* [online]. 2016 [cit. 2019-01-14]. Dostupné z: https://runnable.com/blog/distributing-docker-cache-across-hosts.

54. PUNDSACK, Mark; GitLab contributors. *GitLab: Introduction to environments and deployments* [online]. 2019 [cit. 2019-01-20]. Dostupné z: https://docs.gitlab.com/ee/ci/environments.html.

55. PIPINELLIS, Achilleas; GitLab contributors. *GitLab: Review Apps* [online]. 2018 [cit. 2019-01-20]. Dostupné z: https://docs.gitlab.com/ee/ci/review_apps/.

56. PIPINELLIS, Achilleas; GitLab contributors. *GitLab: Auto DevOps* [online]. 2019 [cit. 2019-01-20]. Dostupné z: https://docs.gitlab.com/ee/topics/autodevops/.

57. RUUD, rdcl; GitLab forum contributors. *GitLab: Should auto devops be disabled by default?* [online]. 2018 [cit. 2019-01-20]. Dostupné z: https://forum.gitlab.com/t/should-auto-devops-be-disabled-by-default/19631.

58. GOMPA, Neal; GitLab contributors. *GitLab CI pipelines with an automatic job chained on a manual job runs automatically instead of waiting for the manual job to successfully run first* [online]. 2018 [cit. 2019-01-20]. Dostupné z: https://gitlab.com/gitlab-org/gitlab-ce/issues/53454.

59. DÍTĚ, Mikuláš; GitLab contributors. *Kubernetes integration should filter pods using deployment selector* [online]. 2018 [cit. 2019-01-20]. Dostupné z: https://gitlab.com/gitlab-org/gitlab-ee/issues/6389.

60. SPEICHER, Robert; GitLab contributors. *GitLab: Meet our team* [online]. 2018 [cit. 2019-01-20]. Dostupné z: https://about.gitlab.com/company/team/.

61. VAN DER VOORT, Job. *A comment on GitLab Issue Tracker* [online]. 2016 [cit. 2019-01-20]. Dostupné z: https://gitlab.com/gitlab-org/gitlab-ce/issues/15635#note_12741465.

62. BOHN, Dann; GitLab contributors. *Kubernetes integration should filter pods using deployment selector* [online]. 2017 [cit. 2019-01-20]. Dostupné z: https://gitlab.com/gitlab-org/gitlab-runner/issues/2681.

63. READ, Evan; GitLab contributors. *GitLab Webhooks* [online]. 2019 [cit. 2019-03-06]. Dostupné z: https://docs.gitlab.com/ee/user/project/integrations/webhooks.html.

64. RAMSAY, James; GitLab contributors. *GitLab Plugin system* [online]. 2018 [cit. 2019-03-06]. Dostupné z: https://docs.gitlab.com/ee/administration/plugins.html.

65. NATIONAL VULNERABILITY DATABASE, NIST. *GitLab: Vulnerability Statistics* [online]. 2019 [cit. 2019-01-14]. Dostupné z: https://www.cvedetails.com/vendor/13074/Gitlab.html.

66. DOE, Erica; GitLab contributors. *GitLab blog: #movingtogitlab* [online]. 2018 [cit. 2019-01-14]. Dostupné z: https://about.gitlab.com/2018/06/03/movingtogitlab/.

67. BLAKE, Cindy; GitLab contributors. *A seismic shift in application security* [online]. 2018 [cit. 2019-01-14]. Dostupné z: https://about.gitlab.com/resources/downloads/gitlab-seismic-shift-in-application-security-whitepaper.pdf.

68. VOSMAER, Jacob; GitLab contributors. *Updating GitLab installed with the Omnibus GitLab package: Zero downtime updates* [online]. 2018 [cit. 2019-01-14]. Dostupné z: https://docs.gitlab.com/omnibus/update/README.html#zero-downtime-updates.

69. DRAGONI, Nicola; LANESE, Ivan; LARSEN, Stephan Thordal; MAZZARA, Manuel; MUSTAFIN, Ruslan; SAFINA, Larisa. Microservices: How To Make Your Application Scale. In: PETRENKO, Alexander K.; VORONKOV, Andrei (ed.). *Perspectives of System Informatics*. Cham: Springer International Publishing, 2018, s. 95–104. ISBN 978-3-319-74313-4.

70. KIM, Jaemyung; SALEM, Kenneth; DAUDJEE, Khuzaima; ABOULNAGA, Ashraf; PAN, Xin. Database high availability using shadow systems. In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 2015, s. 209–221.

71. REDIS LABS. *Redis High Availability* [online]. 2018 [cit. 2019-01-20]. Dostupné z: https://redislabs.com/redis-features/high-availability.

72. SVENSSON, Christian; SchemaStore contributors. *SchemaStore: Gitlab CI configuration* [online]. 2018 [cit. 2019-01-16]. Dostupné z: https://github.com/SchemaStore/schemastore/blob/master/src/schemas/json/gitlab-ci.json.

73. PROFFITT, Brian; ITworld. *Hudson devs vote for name change; Oracle declares fork* [online]. 2011 [cit. 2019-01-23]. Dostupné z: https://www.itworld.com/article/2746627/open-source-tools/hudson-devs-vote-for-name-change--oracle-declares-fork.html.

74. KAWAGUCHI, Kohsuke. *Jenkins: Distributed builds* [online]. 2018 [cit. 2019-01-26]. Dostupné z: https://wiki.jenkins.io/display/JENKINS/Distributed+builds.

75. NENASHEV, Oleg. *Jenkins: Java requirements* [online]. 2018 [cit. 2019-01-23]. Dostupné z: https://jenkins.io/doc/administration/requirements/java/.

76. ORACLE CORPORATION. *Oracle Java SE Support Roadmap* [online]. 2018 [cit. 2019-01-23]. Dostupné z: https://www.oracle.com/technetwork/java/eol-135779.html.

77. DE LOOF, Nicolas. *GitHub Branch Source Plugin* [online]. 2018 [cit. 2019-01-23]. Dostupné z: https://jenkins.io/blog/2018/08/23/speaker-blog-casc-part-1/.

78. CLOUDBEES, INC. *GitHub Branch Source Plugin* [online]. 2019 [cit. 2019-01-23]. Dostupné z: https://go.cloudbees.com/docs/plugins/github-branch-source/.

79. PEARCE, Jeff; GoDaddy Operating Company, LLC. *Jenkins Best Practices - Practical Continuous Deployment in the Real World* [online]. 2018 [cit. 2019-01-23]. Dostupné z: https://godaddy.github.io/2018/06/05/cicd-best-practices/.

80. DUMAY, James. *Introducing Blue Ocean: a new user experience for Jenkins* [online]. 2016 [cit. 2019-01-23]. Dostupné z: https://jenkins.io/blog/2016/05/26/introducing-blue-ocean/.

81. NATIONAL VULNERABILITY DATABASE, NIST. *Jenkins: Vulnerability Statistics* [online]. 2019 [cit. 2019-01-23]. Dostupné z: https://www.cvedetails.com/vendor/15865/Jenkins.html.

82. MATHEW, Subin. *AWS CodeBuild Plugin* [online]. 2018 [cit. 2019-01-26]. Dostupné z: https://wiki.jenkins.io/display/JENKINS/AWS+CodeBuild+Plugin.

83. KAWAGUCHI, Kohsuke. *Install Plugins Without Restarting Jenkins* [online]. 2011 [cit. 2019-01-23]. Dostupné z: https://www.cloudbees.com/blog/install-plugins-without-restarting-jenkins.

84. SUCKER, Sebastian; JAHN, Sebastian; SCHREIBER, Arne. *A Jenkins Master, with a Jenkins Master, with a ...* [online]. 2018 [cit. 2019-01-26]. Dostupné z: https://endocode.com/blog/2018/08/17/jenkins-high-availability-setup/.

85. LINVILLE, Misty. *Post-installation steps for Linux* [online]. 2018 [cit. 2019-01-27]. Dostupné z: https://docs.docker.com/install/linux/linux-postinstall/.

86. VOHRA, Sameer. *Concourse: Implementing a Resource* [online]. 2018 [cit. 2019-02-01]. Dostupné z: https://concourse-ci.org/implementing-resources.html.

87. *GitHub Search for Concourse Resource Repositories* [online]. 2019 [cit. 2019-02-01]. Dostupné z: https://github.com/search?o=desc&q=concourse+resource&s=stars&type=Repositories.

88. BULLOCK, Topher. *Concourse: Discussion; Fine-grained / Role-based Access Control* [online]. 2017 [cit. 2019-02-01]. Dostupné z: https://github.com/concourse/concourse/issues/1317.

89. NATIONAL VULNERABILITY DATABASE, NIST. *Concourse: Vulnerability Search* [online]. 2019 [cit. 2019-02-01]. Dostupné z: https://www.cvedetails.com/google-search-results.php?q=concourse&sa=Search.

90. POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL Documentation: File System Level Backup* [online]. 2019 [cit. 2019-03-10]. Dostupné z: https://www.postgresql.org/docs/9/backup-file.html.

91. DIMSDALE-ZUCKER, Rob. *Concourse Pipeline Resource* [online]. 2019 [cit. 2019-02-01]. Dostupné z: https://github.com/concourse/concourse-pipeline-resource.

92. JACKSON, Kyle. *Concourse-worker: error: volume graph is disabled* [online]. 2016 [cit. 2019-02-01]. Dostupné z: https://github.com/concourse/concourse/issues/402#issuecomment-223619301.

93. DODD, Benedict; HÄMÄLÄINEN, Ilja. *Concourse CI - how to run functional tests?* [online]. 2019 [cit. 2019-03-10]. Dostupné z: https://stackoverflow.com/a/38085222/326257.

94. RYDZEWSKI, Brad; BOSTOCK, Matt. *Drone: Possible to put web application behind a load balancer?* [online]. 2017 [cit. 2019-02-10]. Dostupné z: https://github.com/drone/drone/issues/756#issuecomment-281381959.

95. RYDZEWSKI, Brad. *Drone: Jsonnet Extension* [online]. 2018 [cit. 2019-02-10]. Dostupné z: https://docs.drone.io/extend/config/jsonnet/.

96. RYDZEWSKI, Brad. *Drone: GitHub Application support* [online]. 2018 [cit. 2019-02-10]. Dostupné z: https://github.com/drone/drone/issues/2422.

97. RYDZEWSKI, Brad. *Drone: Database* [online]. 2017 [cit. 2019-03-11]. Dostupné z: https://docs.drone.io/administration/server/database/.

98. BOERGER, Thomas; RYDZEWSKI, Brad. *Drone-migrate: Migration utility from Drone 0.8.x to 1.0.x* [online]. 2017 [cit. 2019-03-11]. Dostupné z: https://github.com/drone/drone-migrate.

99. THOUGHTWORKS, INC. *Continuous integration* [online]. 2018 [cit. 2018-12-21]. Dostupné z: https://www.thoughtworks.com/continuous-integration.

100. ARAVIND, SV; ThoughtWorks, Inc. *GoCD Pipelines as code* [online]. 2017 [cit. 2019-02-06]. Dostupné z: https://docs.gocd.org/current/advanced_usage/pipelines_as_code.html.

101. JARIWALA, Juhi; ThoughtWorks, Inc. *Available GoCD Extension Points* [online]. 2017 [cit. 2019-02-06]. Dostupné z: https://docs.gocd.org/current/extension_points/.

102. THOUGHTWORKS, INC. *GoCD Plugin List* [online]. 2019 [cit. 2019-02-06]. Dostupné z: https://www.gocd.org/plugins/.

103. *Shodan: Search for open port 8153* [online]. 2019 [cit. 2019-02-06]. Dostupné z: https://www.shodan.io/search?query=port%3A8153.

104. *HackerOne: GoCD* [online]. 2019 [cit. 2019-02-06]. Dostupné z: https://hackerone.com/gocd/hacktivity.

105. THOUGHTWORKS, INC. *GoCD Business Continuity Addon* [online]. 2015 [cit. 2019-02-06]. Dostupné z: https://extensions-docs.gocd.org/business-continuity/current/.

106. PEÑA, Louda; ThoughtWorks, Inc. *Upgrading GoCD* [online]. 2017 [cit. 2019-02-06]. Dostupné z: https://docs.gocd.org/current/installation/upgrading_go.html.

107. SANJANA, B; ThoughtWorks, Inc. *Backup GoCD Server* [online]. 2019 [cit. 2019-03-11]. Dostupné z: https://docs.gocd.org/current/advanced_usage/one_click_backup.html.

108. NAGY, Anna; Travis CI, GmbH. *Travis CI: Combining The Linux Infrastructures* [online]. 2018 [cit. 2019-02-03]. Dostupné z: https://blog.travis-ci.com/2018-10-04-combining-linux-infrastructures.

109. WRIGHT, Samuel. *Travis CI: Build Matrix* [online]. 2018 [cit. 2019-02-03]. Dostupné z: https://docs.travis-ci.com/user/build-matrix/.

110. LUNA, Michelle. *CircleCI: Tips for Migrating to 2.0* [online]. 2019 [cit. 2019-02-03]. Dostupné z: https://circleci.com/docs/2.0/migration/.

111. HAASE, Konstantin. *Incident Post-Mortem and Security Advisory: Data Exposure After travis-ci.com Outage* [online]. 2018 [cit. 2019-02-03]. Dostupné z: https://blog.travis-ci.com/2018-04-03-incident-post-mortem.

112. *CircleCI Status Page and Uptime History* [online]. 2019 [cit. 2019-02-03]. Dostupné z: https://status.circleci.com/uptime.

113. *TravisCI Status Page and Uptime History* [online]. 2019 [cit. 2019-02-03]. Dostupné z: https://www.traviscistatus.com/uptime.

114. *Semaphore CI Status Page and Uptime History* [online]. 2019 [cit. 2019-02-03]. Dostupné z: http://status.semaphoreci.com/uptime.

115. GITHUB, INC. *GitHub Facts* [online]. 2019 [cit. 2019-02-03]. Dostupné z: https://github.com/about/facts.

116. WINTON, Steve; VAN WIGGEREN, Nick. *GitHub Actions shellcheck* [online]. 2019 [cit. 2019-02-03]. Dostupné z: https://github.com/actions/bin/tree/master/shellcheck.

117. GITHUB INC. *Creating a new GitHub Action: Hello world action example* [online]. 2019 [cit. 2019-03-06]. Dostupné z: https://developer.github.com/actions/creating-github-actions/accessing-the-runtime-environment.

118. GITHUB INC. *Creating a new GitHub Action: Hello world action example* [online]. 2019 [cit. 2019-02-03]. Dostupné z: https://developer.github.com/actions/creating-github-actions/creating-a-new-action/#hello-world-action-example.

119. LEE, Casey. *ACT: Run your GitHub Actions locally* [online]. 2019 [cit. 2019-02-03]. Dostupné z: https://github.com/nektos/act.

120. DÍTĚ, Mikuláš; GitLab contributors. *GitLab Issues: 11.8 Breaks CI Linting and consequently all CI jobs* [online]. 2019 [cit. 2019-04-03]. Dostupné z: https://gitlab.com/gitlab-org/gitlab-ce/issues/58073.

121. JANKOVSKI, Marin; GitLab contributors. *GitLab Issues: Test cloud native k8s installations with GitLab QA* [online]. 2017 [cit. 2019-04-03]. Dostupné z: https://gitlab.com/gitlab-org/gitlab-qa/issues/92.

122. HASHIMOTO, Mitchell. *Vagrant by HashiCorp* [online]. 2018 [cit. 2018-12-27]. Dostupné z: https://www.vagrantup.com/.

123. SUŠÁNKA, Tomáš. Konfigurace serveru a její automatizace pomocí Vagrant a verzovacího nástroje Chef. 2014. Dostupné také z: https://dspace.cvut.cz/handle/10467/25987.

124. ORACLE CORPORATION. *Develop Using VM Virtual Box, Deploy Anywhere* [online]. 2018 [cit. 2018-12-27]. Dostupné z: https://www.oracle.com/virtualization/virtualbox/.

125. CANONICAL LTD. *Ubuntu: The leading operating system for PCs, IoT devices, servers and the cloud* [online]. 2018 [cit. 2018-12-27]. Dostupné z: https://www.ubuntu.com/.

126. W3TECHS, a division of Q-Success Web-based Services. *Usage statistics and market share of Linux for websites* [online]. 2018 [cit. 2018-12-27]. Dostupné z: `https://w3techs.com/technologies/details/os-linux/all/all`.

127. CHEF SOFTWARE INC. *Chef: Deploy new code faster and more frequently. Automate infrastructure and applications* [online]. 2018 [cit. 2018-12-27]. Dostupné z: `https://www.chef.io/`.

Dodatek A

Použité zkratky

ACL *Access control list*, metoda řízení přístupu
API *Application programming interface*
APT *Advanced Package Tool*, balíčkovací systém
AWS *Amazon Web Services*, poskytovatel cloudu

CasC *Configuration as a Code*, metoda správy konfigurace
CD *Continuous Delivery*, alternativně *Continuous Deployment*, vizte str. 9
CE *Community edition*
CI *Continuous Integration*, vizte str. 9
CI/CD *Continuous Integration and Deployment*, vizte str. 9
CLI *Command-line interface*
CSS Cascading Style Sheets
CSV *Comma-separated values*, formát tabulkových dat
CVE *Common Vulnerabilities and Exposures*, systém pro sdílení zranitelností
CVSS *Common Vulnerability Scoring System*, hodnocení zranitelností

DAST *Dynamic Application Security Testing*
DinD *Docker in Docker*, vizte str. 10
DNS *Domain Name System*

EE *Enterprise edition*, vizte str. 21

FastCGI *Fast Common Gateway Interface*, protokol komunikace aplikace a HTTP serveru

GCP *Google Cloud Platform*, poskytovatel cloudu
gRPC *gRPC Remote Procedure Calls*, protokol pro meziprocesovou komunikaci
GUI *Graphical user interface*

HA *High availability*
HCL *HashiCorp Configuration Language*
HTML *Hypertext Markup Language*
HTTP *Hypertext Transfer Protocol*, internetový komunikační protokol
HTTPS *Hypertext Transfer Protocol Secure*

IaC *Infrastructure as Code*, filozofie správy serverů
IDE *Integrated development environment*
IP *Internet Protocol (address)*

JRE Java Runtime Environment
JS *JavaScript*
JVM *Java Virtual Machine*, závislost pro spuštění mj. Java aplikací

L4 Transportní vrstva OSI modelu
L7 Aplikační vrstva OSI modelu
LB *load balancer*, vyvažování zátěže
LDAP *Lightweight Directory Access Protocol*
LFS Git *Large File Storage*
LLVM *Low Level Virtual Machine*, kompilátor
LTS *Long-term support*
LXC Linux Containers

MIT *Massachusetts Institute of Technology*
MTBI/MTBF *mean time between failures (interruptions)*, střední čas mezi výpadky aplikace

MTTR *mean time to recovery*, střední doba nutná k obnovení dostupnosti
MVC *Model View Controller*

NDA *Non-disclosure agreement*, smlouva o utajení
NFS *Network File System*
NPM *Node Package Manager*

OSI *Open Systems Interconnection*
OWASP *The Open Web Application Security Project*

PHP *PHP: Hypertext Preprocessor*, scriptovací programovají jazyk
PHP-FPM *PHP FastCGI Process Manager*, vrstva mezi PHP aplikací a HTTP serverem
PM2 *Process Manager 2*, Node.js aplikace pro správu procesů

RBAC *Role-based access control*, metoda řízení přístupu
RC *Release Candidate*
RDBMS *Relational database management system*
RFC *Request for Comments*
RMI *remote method invocation*, vzdálené spouštění metod
RSA *Rivest–Shamir–Adleman*
RSS *Rich Site Summary*
RVM *Ruby Version Manager*

SaaS *Software as a Service*, služba kterou hostuje poskytovatel
SAST *Static Application Security Testing*
SCM *Source Control Management*
SLA *Service-level agreement*
SPA *Single page application*
SPoF *Single point of failure*
SRE *Site reliability engineering*, filozofie údržby aplikace
SSH *Secure Shell*

TCP *Transmission Control Protocol*
TLS *Transport Layer Security*

UI *User interface*
UID *User identifier*
URL *Uniform Resource Locator*
UX *User experience*

VM *Virtual machine*

XML *Extensible Markup Language*
XSS *Cross-site scripting*

YAML *YAML Ain't Markup Language*

Dodatek B

Implementace

Tato příloha popisuje technické provedení dílčích částí práce. Především dokumentuje obecné procesy shodující se pro všechny porovnávané CI/CD systémy. Konkrétní detaily jednotlivých systémů jsou zdokumentovány v hlavní části práce.

Self-hosted CI/CD systémy jsem nasazoval v lokálním virtuálním stroji za pomoci prostředí Vagrant [122, 123]. Pro tuto práce jsem využil současně nejaktuálnější verzi `2.2.2`. Jako virtualizační jádro jsem použil Oracle VirtualBox [124] ve verzi `5.2.22-126460-OSX`.

Jako základ každé instalace jsem vybral Ubuntu [125], která je podle W3Techs s 38,1 % nejpoužívanější Linuxová distribuce [126]. Zvolil jsem aktuální vydání LTS (long-term support) `18.04`, oficiálně publikované jako Vagrant box `ubuntu/bionic64`.

K nainstalování závislostí na čistý operační systém i k instalaci samotných aplikací jsem využil software Chef [127]. Jde o systém pro správu konfigurace a podporuje vývoj ve stylu *Infrastructure as Code* (infrastruktura v kódu, oproti „klikacímu“ nastavování někde v GUI). Všechny konfigurace a nastavení systémů jsem tak mohl verzovat a sdílet na přiloženém médiu. Veškeré popsané experimenty by tak měly být naprosto opakovatelné a spustitelné s minimem další práce.

Pro každou komponentu jsem vytvořil samostatný virtuální stroj. Pro zjednodušení práce jsem ručně přidělil každému stroji IPv4 adresu z privátního bloku `10.0.0.0/24`. Nezprovozňoval jsem DNS server a pouze jsem každému stroji nastavil jména ostatních stanic v `/etc/hosts`. Aby spolu stanice mohli komunikovat, především aby CD servery měli přístup k HTTP serverům, přiděluje se každé stanici stejný předgenerovaný

RSA klíč. To je špatná praxe, je to nebezpečné a pro praktické použití je to nepřípustné. V tomto případě jsem ale pro testování volil praktickou možnost, které zároveň nabízí 100% opakovatelnost.

V ukázkových projektech jsem potřebovat využít samostatné git repozitáře, protože to velká část CI/CD systémů vyžaduje, ale chtěl jsem zároveň verzovat celou diplomovou práci v jednom repozitáři. Vědomě jsem nepoužil git submodules, protože chci co nejjednodušší systém a distribuovat celou práci mimo přiloženého media i online, s případnými aktualizacemi. Místo toho jsem vytvořil v každém ukázkovém projektu git repozitář, který se používá pouze při pushování do CI/CD systému. Potom složku `.git` přesunu předpřipraveným scriptem `make unstage`, případně vrátím příkazem `make stage`. Toto řešení mi dále umožňuje rychle iterovat nad projekty, nahrávat změny do CI/CD a upravovat, a po dolazení pak verzovat výsledky bez nutnosti commity rebasovat a přejmenovávat.

B.1 GitLab

K instalaci GitLabu jsem použil oficiální návod pro systém Ubuntu. Vytvořil jsem vlastní – velmi jednoduchý – předpis pro systém Chef. Záměrně jsem nepoužil žádný z řady existujících Chef cookbooks, abych měl přehled co instalace obnáší. Dále můj cookbook konfiguruje GitLab přímo pro potřeby této práce.

Po instalaci je automaticky zaregistrován uživatel `root` s přednastaveným heslem `CVUT_FIT`. Repozitáře jsem nenascriptoval a je potřeba je ručně vytvořit. Dále je potřeba uživateli nastavit veřejný SSH klíč, který slouží ke spárování identity při práci s repozitáři.

B.2 Jednoduchý webserver

Pro nasazení ukázkových projektů jsem vytvořil nový čistý virtuální stroj. Abych mohl provozovat na HTTP portu více aplikací rozlišených podle host, nainstaloval jsem na server nginx. Pro statický projekt jsem pouze nakonfiguroval cestu k veřejnému adresáři. Deploy probíhá dvoufázově: nejprve se na server nahrají statické zdroje (javascripty, kaskádové styly, …) a poté se přepíší HTML soubory které na ně odkazují. Dynamické review apps jsem na straně web serveru implementoval proměnným názvem serveru:

```
server_name "~^(?<domain>[\w-]+)\.pl\.ditemiku\.local$";
root /srv/pl-$domain;
```

Pro druhý ukázkový projekt jsem nainstaloval PHP-FPM a nechal uživatelské požadavky přeposílat přes fastcgi. Deploy jsem implementoval podobně jako u statického webu. Díky tomu, že PHP používá opcache – tzn. nečte a neparsuje při každém požadavku zdrojové soubory znovu – stačí nahrát všechny soubory a pak opcache smazat. Toho se nejsnadněji docílí reloadem PHP-FPM masteru, který postupně vypne staré workery a zapne nové.

Třetí projekt – aplikaci v kontejneru – jsem nasadil pomocí Docker Swarm, kde jsem z webserveru udělal jednouzlový server. Oproti čistému dockeru Swarm nabízí snažší správu závislostí pomocí `docker-compose.yml` a částečně umí udělat tzv. rolling update bez výpadku. V CI systému se staví samotný docker image a nahraje se do sdíleného registru (Docker Registry) pod dvěma tagy: `latest` a `$CI_COMMIT_SHA`. To je hash z verzovacího systému, který právě CI systém staví. Díky otagování každé verze pak máme snažší správu: víme v jaké verzi aplikace zrovna běží a můžeme případně udělat rychlý rollback na starší verze. V případě že bychom pouze používali tag `latest`, tuto informaci bychom neměli a při rollbacku bychom museli spustit znova celou CI pipeline.

B.3 Použitá Jenkins rozšíření

Následující seznam jsou všechna rozšíření, která jsem použil při testování CI/CD systému Jenkins 2.3. Nevypisuji nutná rozšíření distribuovaná s Jenkins a závislosti daných rozšíření.

- `Build Timeout 1.19`
- `Configuration as Code 1.4`
- `Docker 1.1.5`
- `Email Extension Plugin 2.63`
- `Git plugin 3.9.1`
- `Matrix Authorization Strategy Plugin 2.3`
- `OWASP Markup Formatter Plugin 1.5`
- `Pipeline 2.6`
- `Timestamper 1.8.10`
- `Workspace Cleanup Plugin 0.37`

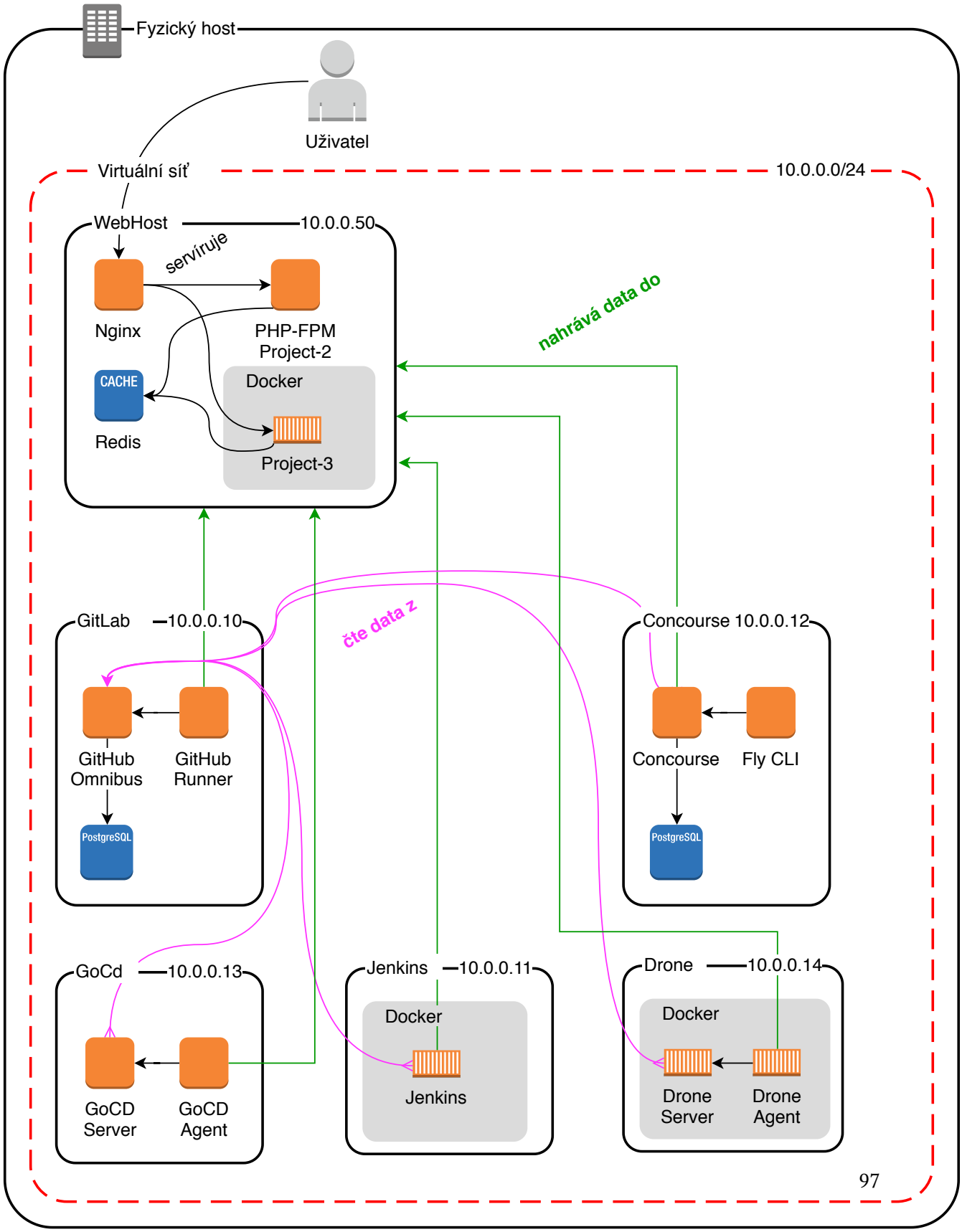


Fig. B.1: Diagram propojení virtuálních strojů a dalších komponent na lokální síti.

B.4 Přiložené médium

Na přiloženém médiu je kompletní dokumentace k této diplomové práci a celá virtuální laboratoř, ve které probíhalo testování CI/CD systémů. Identická data jsou také dostupná online na

`https://github.com/Mikulas/thesis/tree/published`

- `MT_Dite_Mikulas_2019.pdf` ..... text této práce
- `appendix/` ..... syrová neagregovaná data pro grafy
- `chapters/` ..... zdrojové kódy pro sazbu této zprávy o diplomové práci
- `main.ctex` ..... index pro vygenerování PDF, vyžaduje C preprocessor
- `media/` ..... diagramy a další PDF vložená do práce
- `src/` ..... laboratorní prostředí porovnávaných CI/CD systémů
  - `ci-cd/` ..... Vagrant a Chef předpisy pro jednotlivé CI/CD systémy
  - `cookbooks/` ..... sdílené Chef předpisy
  - `projects/` ..... zdrojové kódy aplikací a definice pipeline pro CI/CD
  - `web-servers/` ..... Vagrant a Chef předpisy pro webový server

Adresářová struktura B.1: Hlavní obsah přiloženého media