FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Diplomová práce

Nástroj pro semi-automatickou konfiguraci a vytvoření infrastruktury pro vývoj, testování a nasazení SW produktu

Studijní program: N0613A140028 Informační technologie
Autor práce: **Bc. David Dlouhý**
Vedoucí práce: Ing. Lenka Kosková Třísková, Ph.D.
Ústav nových technologií a aplikované informatiky

Liberec 2024

FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Zadání diplomové práce

Nástroj pro semi-automatickou konfiguraci a vytvoření infrastruktury pro vývoj, testování a nasazení SW produktu

Jméno a příjmení:	**Bc. David Dlouhý**
Osobní číslo:	M22000018
Studijní program:	N0613A140028 Informační technologie
Zadávající katedra:	Ústav nových technologií a aplikované informatiky
Akademický rok:	2023/2024

Zásady pro vypracování:

1. Proveďte důkladnou rešerši stávajících dostupných řešení a technologií využívajících open-source.
2. Navrhněte a vytvořte nástroj, který umožní částečnou automatizaci při konfiguraci infrastruktury užívané při vývoji, testování a nasazení SW produktu metodou CI/CD s využitím DevOps.
3. Infrastruktura bude zahrnovat konfiguraci systému pro řízení vývoje kódu a jeho verzí, dále systém určený ke kompilaci a sestavení produktu a nakonec systém určený k testování výsledného SW.
4. Jako referenční příklad SW produktu zvolte distribuci OS Linux vyvíjenou s podporou Yocto.

Rozsah grafických prací:	dle potřeby dokumentace
Rozsah pracovní zprávy:	40 – 50 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	čeština

Seznam odborné literatury:

[1] Doak J., Justice D.: Go for DevOps: Learn how to use the Go language to automate servers, the cloud, Kubernetes, GitHub, Packer, and Terraform, Packt Publishing, 2022, ISBN: 1801818894.

[2] Onur Y., Süleyman A.: Introduction to DevOps with Kubernetes, Packt Publishing, 2019, ISBN: 9781789808285.

[3] Cowel C., Lotz N., Timberlake C.: Automating DevOps with GitLab CI/CD Pipelines, Packt Publishing, 2023, ISBN: 1803233001.

Vedoucí práce:	Ing. Lenka Kosková Třísková, Ph.D. Ústav nových technologií a aplikované informatiky
Datum zadání práce:	12. října 2023
Předpokládaný termín odevzdání:	14. května 2024

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

doc. RNDr. Pavel Satrapa, Ph.D.
garant studijního programu

V Liberci dne 19. října 2023

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

12. května 2024 Bc. David Dlouhý

NÁSTROJ PRO SEMI-AUTOMATICKOU KONFIGURACI A VYTVOŘENÍ INFRASTRUKTURY PRO VÝVOJ, TESTOVÁNÍ A NASAZENÍ SW PRODUKTU

ABSTRAKT

Diplomová práce seznámí čtenáře s kompletním procesem vývoje nástroje pro semi-automatickou konfiguraci a vytvoření infrastruktury pro vývoj, testování a nasazení SW produktu. Takový nástroj lze využít i při vývoji embedded OS za podpory Yocto Project. Právě pro tento komplexní vývoj SW je v práci demonstrováno použití nástroje. Čtenář si tak pomocí nástroje může snadno zprovoznit infrastrukturu pro vývojem embedded OS podporující všechny fáze metodiky DevOps. Práce nástroj také porovná s existujícími nástroji. K vývoji nástroje je přistoupeno systematicky. Nejprve práce popisuje analýzu požadavků na nástroj. Čtenář je seznámen s problematikou vývoje embedded OS za podpory Yocto Project. Práce také popisuje využití metodiky DevOps. Analyzuje existujíc IaC nástroje. Práce se dále věnuje návrhu řešení a popisuje dílčí volby založené na důkladné komparaci. Čtenář se seznámí nejen s procesy výběru vhodné architektury a programovacího jazyka. Práce popisuje také implementaci nástroje a podpůrné prostředky (CI/CD, testy, dokumentace aj.), které byly při vývoji využity. Na závěr se práce věnuje ověření nástroje v praxi.

Klíčová slova

IaC, CI/CD, DevOps, Yocto Project, Golang

TOOL FOR SEMI-AUTOMATIC CONFIGURATION AND CREATION OF INFRASTRUCTURE FOR DEVELOPMENT, TESTING AND DEPLOYMENT OF SOFTWARE PRODUCT

ABSTRACT

The master's thesis introduces the reader to the complete process of developing a tool for semi-automatic configuration and creation of infrastructure for the development, testing, and deployment of software products. Such a tool can also be utilized in the development of embedded OS with the support of the Yocto Project. The thesis demonstrates the use of the tool for this comprehensive software development, allowing the reader to easily set up infrastructure for embedded OS development supporting all phases of the DevOps methodology. The thesis also compares the tool with existing ones. The development of the tool is approached systematically. Initially, the thesis describes the analysis of tool requirements. The reader becomes familiar with the issues of developing embedded OS with the support of the Yocto Project. The thesis also describes the use of the DevOps methodology and analyses existing Infrastructure as Code (IaC) tools. Furthermore, the thesis focuses on designing the solution and describes partial choices based on thorough comparison. The reader is introduced not only to the processes of selecting suitable architecture and programming language but also to the implementation of the tool and supporting resources (CI/CD, testing, documentation...) utilized during development. Finally, the thesis addresses the verification of the tool in practice.

Keywords

IaC, CI/CD, DevOps, Yocto Project, Golang

PODĚKOVÁNÍ

Tímto bych chtěl poděkovat Ing. Lence Koskové Třískové, Ph.D. za vedení práce, odborné konzultace a za čas, který mi ochotně věnovala vždy, když bylo třeba. Děkuji také za spolupráci Mgr. Heleně Jandové. Velmi rád bych poděkoval i rodinnému kruhu za podporu a pochopení nejen během psaní této práce, ale také během celého studia.

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM ZDROJOVÝCH KÓDŮ

SEZNAM ZKRATEK

AI	umělá inteligence
API	rozhraní pro programování aplikací
APT	balíčkovací systém
APK	formát souborů
ARM	architektura procesorů
AWS	Amazon Web Services, cloud platforma
BSD	licence pro svobodný SW
BMW	německý výrobce automobilů
BFS	grafový algoritmus prohledávání do šířky
C++	programovací jazyk
CD	kontinuální nasazení
CI	kontinuální integrace
CKI	kontinuální integrace jádra Linux
CLI	příkazová řádka
CRM	SW pro řízení vztahů se zákazníky
CPU	procesor
CPS	kyberneticko-fyzikální systémy
CSS	kaskádové styly
CVE	běžná zranitelná místa a rizika
C#	programovací jazyk
DDD	návrh řízený doménou
DNS	systém doménových jmen
DNF	balíčkovací systém
DSL	nativní jazyk pro nástroj Puppet
ELK	Elastic, Logstash, Kibana
EU	Evropská unie
FM	Fakulta mechatroniky, informatiky a mezioborových studií
GCC	sada překladačů vytvořených v rámci projektu GNU

GIT	distribuovaný systém řízení verzí
GIN	HTTP framework pro jazyk Golang
GNU	projekt zaměřený na svobodný SW
GPL	obecná veřejná licence
GO	programovací jazyk Golang
GUI	grafické uživatelské rozhraní
HCL	konfigurační jazyk společnosti HashiCorp
HTTP	internetový protokol Hypertext Transfer Protocol
HTTPS	zabezpečená verze protokolu HTTP
HW	hardware
IDE	vývojové prostředí
IaC	definice infrastruktury jako kód
IoT	internet věcí
IPv4	internetový protokol čtvrté verze
JSON	multiplatformní způsob zápisu dat
K8s	Kubernetes
KAS	nástroj pro projekty založené na BitBake
MaC	definování monitoring pomocí kódu
MIT	svobodná licence Massachusettského technologického institutu
NFS	internetový protokol pro vzdálený přístup k souborům
NTI	Ústav nových technologií
NXP	firma soustředící se na návrh a výrobu polovodičů
OOP	objektově orientované programování
OTA	bezdrátové doručení nové verze SW
PDF	přenosný formát dokumentů
PS	PowerShell
QEMU	rychlý emulátor
RAM	operační paměť
REST	architektura pro webová API
RPM	balíčkovací systém
RTOS	operační systém reálného času

RTF	formát souboru pro uložení textu
SaaS	SW jako služba
SCP	protokol pro bezpečný přenos dat
SD	typ paměťové karty
SDK	sada vývojových nástrojů
SoS	systémy systémů
SSH	program a zabezpečený komunikační protokol
SVG	škálovatelná vektorová grafika
SW	software
TFTP	jednoduchý protokol pro přenos souborů
TLS	kryptografický protokol
TOML	jednoduchý formát pro psaní strukturovaných souborů
TUL	Technická univerzita v Liberci
UI	uživatelské rozhraní
VCS	nástroj pro řízení verzí
VW	Německá automobilka Volkswagen
X11	SW umožňující vytvořit GUI
YAML	formát čitelného souboru pro člověka určený k serializaci
XML	značkovací jazyk pro popis dat

ÚVOD

V této práci čtenáři popíši proces tvorby nástroje pro semi-automatickou konfiguraci a vytvoření infrastruktury pro vývoj, testování a nasazení SW produktu. Budu se věnovat analýze požadavků na nástroj, návrhu a implementaci řešení. V práci také popíši využití nástroje pro vývoj embedded OS za podpory DevOps a CI/CD. V práci se tedy nebudu věnovat pouze nástroji, ale i vývoji embedded OS a semi-automatickému poskytování infrastruktury, které nástroj vykonává.

Práce je inspirována reálnými potřebami nejen na NTI FM TUL. Výrazně přibývá projektů věnujících se vývoji embedded OS Linux [1]. Jedná se o komplexní vývoj SW. Je tedy třeba tuto činnost zjednodušit a poskytnout uživateli podpůrnou platformu, která mu pomůže implementovat metodiku DevOps [2].

Čtenář může na adrese https://oasis.rodina-dlouha.cz/ navštívit web práce. Web poskytuje rychlé seznámení s vyvíjeným nástrojem a funguje jako rozcestník referující na jednotlivé součásti práce (repozitáře, dokumentace a další).

1 ANALÝZA A REŠERŠE

Před zahájením vlastní realizace bylo nutné podrobně analyzovat potřeby zvolené cílové skupiny, abych mohl přesně vymezit požadavky na výsledné řešení. Úvodní analýza zahrnuje také srovnání existujících řešení, které by mohly definovaným požadavkům vyhovět. Na závěr rešerše poukazuji na nedostatky existujících řešení.

1.1 PROBLEMATIKA A CÍLOVÁ SKUPINA

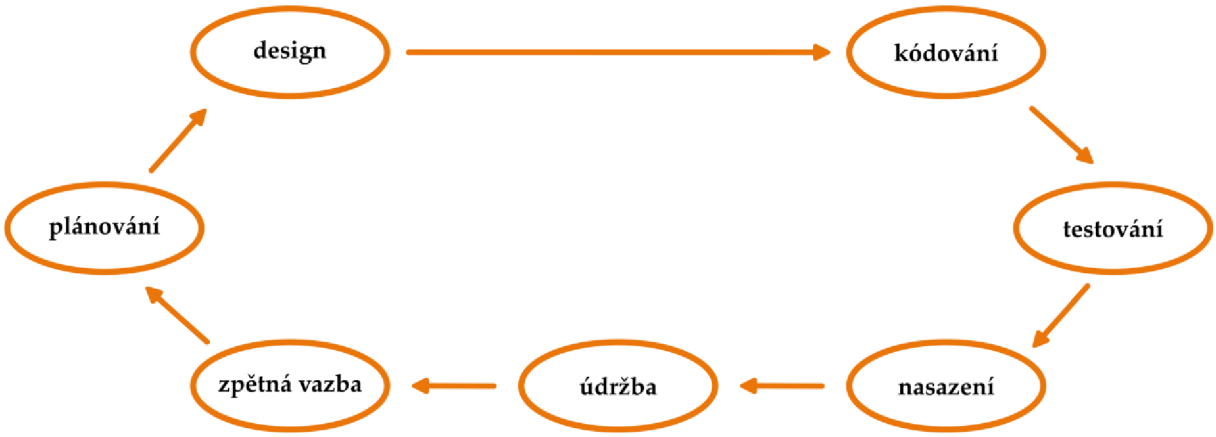
Vymezení cílové skupiny uživatelů mého řešení vychází ze zadání práce, které vyžaduje referenční použití na vývoj distribuce OS Linux s podporou pro Yocto Project. Tyto distribuce se nejčastěji využívají ve světě vestavných zařízení a řešení. V roce 2024 používají více jak dvě miliardy zařízení právě embedded Linux. Jsou to převážně settop boxy, chytré televize, routery, průmyslová zařízení, mobilní zařízení, řídící jednotky v automotive, ve zdravotnických aplikacích nebo systémy využívané v letectví či kosmonautice [1].

Distribuce OS Linux má odlišný životní cyklus od vývoje jedné aplikace nebo webové aplikace. Distribuce OS se skládá z desítek či stovek dílčích softwarových balíků, z nichž každý je vyvíjen nezávisle. Další odlišností je omezená dostupnost koncového zařízení, složitější testování v koncovém zařízení a nutnost plánovat release na dlouhé časové úseky. Setkáváme se zde s odlišnostmi, mezi které se řadí delší vydávání jednotlivých verzí, přístupnost koncového zařízení, tetovací prostředí a další.

Obrázek 1.1 a Obrázek 1.2 jsou ukázkou typických přístupů pro vývoj v popsaných oblastech (webové aplikace vs. vestavná zařízení). Typicky vývoj SW pro vestavné zařízení začíná úvodní definicí požadavků (tzv. Requirements), kdy se analyzují potřeby všech uživatelů výsledku (tzv. stakeholders). Vývoj software je často vázán na vývoj specifického hardware a je omezen řadou průmyslových standardů a metodik, které kladou vlastní požadavky na výsledek. Po analýze vzniká základní plán vývoje s celkovým přehledem. Následuje fáze navrhování, kdy se řeší architektura HW i SW. Návrh HW zahrnuje volbu odpovídajícího typu CPU (mikrokontroler vs. aplikační procesor), zvolené HW platformy a detailů cílového počítače. Definice HW architektury určuje, zda a jaký typ OS bude SW řešení používat. S tím souvisí volba IDE, překladačů, systémů pro sestavení, nástrojů pro ladění, SDK aj. Ne-

smíme zapomenout také na sestavení samotného týmu řešitelů. Velikost a odbornost týmu je individuální a odvíjí se od velikosti projektu a rozpočtu. Výjimkou není ani outsourcing. Na fázi návrhu navazuje implementační fáze, kdy nejprve vznikají HW prototypy. Oproti vývoji webových aplikací jsme silně svázáni s HW, který navíc v rané fázi vývoje není dostupný. Hezkým příkladem je srovnání procesu testování transformace dat ve webové aplikaci a spínání relé mikrokontrolerem. Zatímco ve světě webových aplikací mohu pracovat čistě virtuálně, u embedded potřebují HW farmu zařízení. Farma zařízení je sada fyzických zařízení. Do těchto zařízení se nahrává kód k otestování. Testovací zařízení musí být nejpodobnější plánovanému výslednému produktu. Tato zařízení mají ale další periferie a napojení tak, aby bylo možné sledovat stav průběhu testu. Jakmile je produkt připraven pro uvolnění na trhu, zahájí se sériová výroba HW. Zde vznikají časové prodlevy. Proces sériové výroby HW a následný prodej fyzických zařízení je složitější než prodej SW ve formě služby (SaaS), kdy uživateli stačí internetové připojení a provozovateli pouze cloud. Časové prodlevy u uvedení webové služby do produkce jsou mnohem kratší.

Webová aplikace i vestavené zařízení po prodeji vyžadují podporu ze strany výrobce. Je nutné udržovat SW vybavení a opravovat případné chyby či nově objevené zranitelnosti. V cloudovém prostředí webových aplikací stačí pouze spustit kontejner s aplikací a pomocí komponenty Load balancer přesměrovat síťový provoz. U chytrého senzoru zalitého asfaltem, který musí splňovat určité bezpečnostní standardy, může aktualizace znamenat fyzickou manipulaci se stovkami či tisíci zařízeními v terénu. Vestavná zařízení jsou většinou určena jednoúčelově. Firmware se vyladí a dále se jen provádí bezpečnostní záplaty a nutná údržba. Naopak webové aplikace se mění mnohem rychleji. Denně vycházejí nové funkce a upravují se ty staré. Klíčové je plnění obchodních cílů v co nejkratším čase. Tento trend se velmi pravděpodobně postupně prosadí ve světě vestavných zařízení, zejména u těch, která zůstávají připojena ke komunikační síti. Objevují se už nástroje a přístupy, které toto mohou v budoucnu usnadnit [2]. Mezi tyto nástroje se řadí i tato práce.



Obrázek 1.1: Životní cyklus webové aplikace

Obrázek 1.2: Životní cyklus vestavného zařízení

Při návrhu systému jsem se rozhodl zacílit na malé firmy, začínající startupy a velmi malé vývojářské týmy. Z hlediska plánované SW architektury jsem zadání dále omezil na skupiny, které se věnují tvorbě distribucí OS Linux s podporou Yocto Project.

Takové skupiny nemají většinou ještě pevně dané sady nástrojů [3]. Změna technologií je pro ně obvykle jednodušší. V tomto ohledu jsou flexibilnější. Jedná se obvykle o jednotlivce a malé týmy lidí, kteří jsou zaměřeni určitými technickými směry (programátor, technik, administrátor apod.). Zároveň ale tito pracovníci často nezastávají pouze roli, které nejvíce rozumí. Věnují se i ostatním činnostem. Programátor se například věnuje grafickým návrhům, automatizaci vývoje a jiným činnostem. Osobně jsem se v praxi v takovém prostředí pohyboval, a i to ovlivnilo mou volbu. V této volbě mě podpořila i vedoucí práce, jejíž tým, se soustředí na vývoj linuxových distribucí pro automotive a aerospace. Charakteristika týmu odpovídá právě výše uvedené cílové skupině. Větší firmy dnes již mají nastavené procesy a často mají také svá proprietární řešení [4].

1.2 ANALÝZA POTŘEB

Potřeby cílové skupiny jsem analyzoval formou diskusí s reprezentanty vývojových týmů. Část rešerše jsem mohl udělat na konferenci Embedded Open Source Summit 2023. Zde jsem měl možnost mluvit s lidmi, kteří se dané problematice věnují a mohli by také být potenciálními uživateli mého nástroje. Důležitým podkladem pro mě byl i dokument ECS-SRIA, který tvoří strategický plán v oblastech vývoje systémů pro Evropu s výhledem na 10 až 15 let [5]. Dále jsem také čerpal z oficiálních konferencí projektu Yocto Project, diskusních fór a mailových konferencí vývojářů. Referenčním vývojovým týmem byl tým Laboratoře aplikované informatiky NTI TUL, jejíž pracovníci se věnují přípravě linuxových OS v Yocto Project. Díky úzké spolupráci na vývoji jedné z distribucí jsem se seznámil detailně s existujícími řešeními i problémy, které je třeba řešit.

1.2.1 VÝSTUPY Z KONFERENCÍ

Primární motivací cílové skupiny je potřeba soustředit se na vývoj SW. Vývojář nechce primárně řešit systém pro sestavení jeho produktu. To samé platí i pro proces testování. Dobře nastavený proces vývoje obsahuje fázi kontroly kódu jiným vývojářem (code review) a testování. Zajistí se tak funkčnost kódu a jeho kvalita před další iterací a nasazením. Kontrola také poskytuje další informace validující osobě. Kontrolující vývojář tak musí sám vypracovat statickou analýzu kódu, napsat testy a manuálně je spustit, nebo může využít existující automatizační nástroje. Řada takových nástrojů vyžaduje správné nastavení a správu a často je potřeba systém i vhodně optimalizovat [7]. Urychlí se tak každé automatizované sestavení, ale vývojář se opět musí věnovat konfiguraci a nasazení testovacího nástroje. Proto je důležité, aby takový nástroj byl snadno osvojitelný a měl jednoduchou a rychlou správu.

Dalším vhodnou fází automatizace je také nasazení sestavené a otestované aktualizace. Také nás zajímá i zpětná vazba po nasazení přírůstku. Čím rychlejší reakce, tím lépe. Výstupem analýzy je tedy požadavek na komplexní automatizační platformu, která pokryje dílčí potřeby a zároveň bude snadno upravitelná a ovladatelná. I za cenu nemožnosti specifických konfigurací. Samozřejmostí je i snadné škálování a optimalizace využití výpočetních zdrojů. Díky platformě by mělo být nasazení nového přírůstku levné a spolehlivé. Pro redukci nákladů je vhodné, aby tuto platformu mohl spravovat a ovládat seniorní vývojář. Díky přímo-

čarému a nekomplexnímu ovládání platformy kombinované s jednoduchou a jasně definovanou dokumentací by měl toto programátor snadno zvládnout.

Popis platformy může evokovat filozofii DevOps [8]. Ve světě embedded ještě ale není plošně adaptována, a to kvůli spojení SW a HW [9]. Testování funkčnosti SW v reálném HW je komplexnější než testování registračního formuláře webové aplikace. Naopak otázku monitorování koncových zařízení již firmy adaptovaly úspěšně. Velice často mají tedy implementovány jednoduché a základní koncepce z DevOps. Zbytek nástrojů bohužel musí počkat na nové investice do rozvoje podpory vývoje, případně na časové kapacity [6].

Cílem vývoje platformy DevOps specializované pro embedded je snížení režijních nákladů během vývojového cyklu. Abychom tohoto cíle dosáhli, musí mít výsledný systém nízké náklady na provoz, správu a údržbu. Úkolem vývojáře není složité konfigurování platformy. Ke svým potřebám má čerpat benefity takové platformy [10]. Použití platformy má tedy firmě pomoct zvýšit produktivitu zaměstnanců.

Když firmy vybírají mezi low-cost řešeními a dražšími setupy není problém v ceně řešení, ale spíše v tom, že nejsou dostupná nezávislá srovnání jednotlivých nástrojů. Rozhodnutí jsou často učiněna povrchně a nejsou zcela optimální. Pro mnohé firmy je důležité také zvážit využití open-source řešení. Existuje určitá paranoia vůči komerčním produktům a preferují se spíše volně dostupné alternativy. Toto rozhodnutí může být motivováno jednak snahou minimalizovat náklady, ale také touhou po větší transparentnosti a nezávislosti na jednom dodavateli [6].

Použití externích systémů CI/CD vyžaduje specifické znalosti pro jejich správné nastavení a integraci do stávající infrastruktury. Kvůli nedostatečnému porozumění konfigurace není často plně využit potenciál technologií, jako je Docker a virtualizace. Nutné je vhodně vybrat úložiště kódu. Při přípravě serveru starajícího se o sestavení produktu je nutný detailní návrh infrastruktury (správné nastavení pro paralelní sestavování a efektivní využití dostupných zdrojů). Například koncepty jako GitOps nejsou běžně využívány, i když by mohly přinést mnoho výhod. Častou překážkou využití některých platforem a součástí infrastruktury jsou požadavky na odbornou znalost dané technologie. Typickým zástupcem je například ELK stack. Ten vyžaduje zkušeného správce pro úspěšnou implementaci. Firmy tak raději volí tvorbu vlastních skriptů a automatizací namísto existujících systémů [7].

Takové systémy pro vývoj se také mohou nacházet v prostředí cloudu. Je tak snadno zajištěna škálovatelnost a flexibilita. Firmy využívají cloudová prostředí, jako je Azure, AWS nebo DigitalOcean, aby mohly využít konceptu Infrastructure as Code (IaC), což zajišťuje znovupoužitelnost a udržitelnost jejich infrastruktury. Díky cloudu se také nemusejí starat o fyzické hardwarové vybavení. Také mohou provádět spekulativní optimalizace zdrojů, aby šetřily výkon, respektive finance. Další možná konfigurace je tzv. hybridní infrastruktura, kdy část infrastruktury je fyzicky u dané firmy (self-managed) a v cloudovém prostředí se nachází sestavovací stroje, které vyžadují nárazový velký výkon [7].

Embedded vývojáři zdůrazňují požadavky na jednoduchost a snadnost použití. Požadují jednoduchou konfiguraci CI/CD, přehledný dashboard a snadné pochopení procesů. Žádané je také rychlé a snadné nasazení spolu s možností vše ovládat skrze jednotný nástroj nebo skript. Nově se také vývojáři zajímají i o možnosti instalace a provozu takových systémů na malých clusterech sestavených z minipočítačů s procesory architektury ARM. Lákavé principy pro nasazení jsou koncepty DevOps jako GitOps. Ty ale vyžadují investici do osvojení firmy a zaškolení personálu. Některé existující nástroje, jako komplexní Yocto Autobuilder, často překračují potřeby těchto firem a vývojáři se také potýkají se složitostí nastavení verzování a integrací systémů. Nástroje, které nabízejí mnoho funkcí, často vývojářům připadají jako zbytečně matoucí a jejich použití je složité. Častou problematikou, která souvisí s komplexitou a složitostí, je nutnost vypořádat se s problémy jako je sestavení GitLab Runner a spouštění GitLab CI Jobs. Je tedy důležité pomoci tyto kroky vývojářům automatizovat, nebo připravit vzorové příklady pro použití. Hlavním cílem je tedy maximalizovat jednoduchou a přímočarou použitelnost nástrojů [6].

Neméně důležité je i správné zabezpečení jednotlivých komponent. Klasickým prohřeškem jsou hesla uložená v repozitáři projektu. Chybí také implementace bezpečnostních mechanismů pro CI/CD pipelines. Tato zabezpečení a další bezpečnostní mechanismy, mezi které patří i statická kontrola a testování kvality kódu, pro firmu znamenají větší hodinovou dotaci a s tím úměrně roste také hodnota investice do daného projektu. Nicméně jedná se o důležité podpůrné prostředky. Statická analýza programu mimo jiné zjistí včas bezpečnostní prohřešky. Tedy výše zmíněné uložení hesla přímo v kódu. To vše se děje bez spuštění dané aplikace. Protikladem je pak dynamická analýza. Tyto analýzy se typicky spouštějí na verzo-

vacích platformách při slučování nových přírůstků kódu. Osoba zodpovědná za integraci změny díky automatizaci typicky obdrží přehlednou zpětnou vazbu z těchto analýz. Jedná se tedy o bránu, kterou musí projít každá změna. Tím se zajistí splnění požadavků na bezpečnost pro každý nový přírůstek. Obecně se používají nástroje dodávané týmem tvůrců daného programovacího jazyka a také open source nástroje jako GCC, Clang, CPPCheck, Flawfinder, RATS, Split nebo Scan-build. Výjimkou nejsou ani komerční produkty, ty se častěji využívají v prostředích korporátu. Pro vizualizaci jsou žádané také nástroje Code checker a rozšíření pro Clang static analyser. Pomáhají sledovat volání požadavků sestavení a analyzovat obdržená data. Bezpečnostní nástroje jako Parsec jsou nedílnou součástí infrastruktury a sledují zranitelnosti nejen během sestavování, ale i poté, co je aplikace nasazena Pro zástupce z řad zákazníků jsou také žádané integrace nástroje pro zpětnou vazbu. Klidně za pomoci využití webových rozhraní [6].

Vývojářské týmy usilují o automatizaci svých procesů, zejména v oblasti sestavování aplikací a integrace změn z různých repozitářů. Využívají pravidelné integrační automatizace (CI Jobs). Prakticky žádný tým vývojářů nechce pouze použít automatizaci, ale chce této automatizaci také porozumět. Kritickým požadavkem je také sledování událostí v procesu. K tomu se využívají přehledná grafická webová rozhraní a vizualizační platformy. Součástí takového rozhraní bývají i konfigurace zabezpečení úloh a také plánovač spouštění těchto úloh. Díky implementaci statické analýzy kódu do procesu CI je zajištěna kvalita a bezpečnost vytvářeného softwaru. Kromě toho se týmy také zabývají sledováním a řešením bezpečnostních zranitelností (CVE) a správou licencí (LegalOps). V rámci automatizace se týmy zajímají o využití umělé inteligence (AI). Existuje zájem o poskytování služeb CI jako samostatné služby, což může výrazně usnadnit a zrychlit vývojové procesy. Některé týmy také uplatňují princip IaC a využívají nástroje jako Ansible pro správu infrastruktury, což přispívá k automatizaci a rychlosti vývoje. Vývojářské týmy se neustále snaží vylepšovat své postupy, včetně implementace nových technologií a přístupů, které přinášejí zvýšenou spolehlivost, bezpečnost a rychlost procesu vývoje a nasazení aplikací [6].

U sestavování embedded aplikací a systémů firmy požadují systematické řízení překladu. Mezi primární požadavky na systém sestavení řadili přednášející křížový překlad, absenci kontaminace prostředí pro sestavení a schopnost paralelně využívat různé verze

tohoto systému sestavení. Systém musí při sestavování umět pracovat s konfigurací definovanou proměnným prostředím. Také je zájem o integrační sestavování, kde se reflektují jiné repozitáře a do cílového repozitáře se automaticky integrují nové změny skrze semi-automatické požadavky na začlenění nových změn. Při samotném sestavování pak vedoucí pracovníky a projektové řídící zajímají metriky sestavení. Mohou tak pracovat se zpětnou vazbou a promítnout ji při krátkodobém plánování. Díky metrikám dostane oddělení včas informaci o potřebě vertikálního škálování HW. Typický systém sestavení embedded OS Linux se skládá ze serveru zodpovědného sestavení. Oblíbené je také horizontální škálování sestavovacího serveru. Tento server při sestavení produkuje sstate cache a stahuje soubory potřebné k sestavení dané distribuce. Jak sstate cache, tak i stažené soubory lze sdílet mezi několika sestavovacími servery s paralelně běžícími sestaveními. Standardně se proto tato data ukládají do distribuovaného síťového úložiště. Primárně se využívají cloud služby a NFS. Při paralelním běhu sestavení se musí také logicky řídit pořadí. Například pokud chceme frontu sestavení odbavit co nejrychleji, vyplatí se sestavit nejprve první systém a až poté paralelizovat podobné sestavení. Pokud bychom sestavení paralelizovali ihned na začátku, stane se, že se budou stahovat tytéž soubory, které se následně zbytečně budou paralelně a synchronizovaně ukládat do sdíleného úložiště. Mezi další součásti se řádí systém pro řízení celého procesu kontinuální integrace a webové rozhraní pro sledování CI procesu včetně vizualizace průběhu daného sestavení. Důležitým předpokladem je přehlednost a jednoduchost webového rozhraní. Pro ladění a řešení incidentů se vývojáři a administrátoři neobejdou bez centralizovaného úložiště logů. Po sestavení se vytvářejí artefakty, které je třeba distribuovat a centralizovaně uchovávat. K tomu složí server s rolí úložiště artefaktů [7].

Klasický proces sestavení OS definovaný nástrojem Yocto Project nejprve provede fetch vzdáleného repozitáře (upstream). Změny se tedy načtou a může se s nimi dále operovat. Stažené soubory se následně rozbalí (extract). Zdrojové kódy se mohou upravovat (patch) a následně se konfigurují (configure). Konfigurovatelné zdroje jsou překládány a jsou vytvářeny spustitelné SW balíky (build). Poté, co vzniknou artefakty, proběhne jejich instalace. Finálním krokem je pak zabalení systémového kořenového adresáře daného systému souborů [7].

Obrázek 1.3: Proces sestavení Yocto Project

Testování v procesu vývoje embedded SW hraje klíčovou roli v zajištění kvality a spolehlivosti výsledného produktu. U embedded zařízeních oproti klasickým webovým aplikacím se setkáváme s odlišným chováním testovaného objektu. Vždy totiž při špatných výsledcích testů musíme vzít v úvahu i hypotézu, že příčinou chyb je vadný HW.

Druhou odlišností je potřeba zajistit vhodnou automatizaci testů – například automatizovaný stisk tlačítka a správné měření reakce systému. Týmy tak mají zájem o již existující nástroje, jako je CKI a Google Syzbot pro zlepšení procesu testování a verifikace. Velké firmy (například NXP) investují do vlastního řešení.

Diskutují se možnosti využití modelu digitálních dvojčat pro virtuální integraci, verifikaci a validaci, což může zlepšit spolehlivost a efektivitu testování. Důraz se klade na simulace, které umožňují přesné simulace fyzického prostředí. Zkoumají se také možnosti distribuce testů na různá zařízení a provádění záplat s bezpečnostními testy před jejich nasazením. Zde je však nutné myslet na bezpečnostní mechanismy typu autentifikace při spouštění pokusu spustit test na nějakém zařízení / farmě zařízeních.

Oblíbené je také křížové testování. Pokud již tedy je proces testování zaveden, pak firmy sledují pokrytí kódu testy. Nejčastěji v procentech. Cílem není pokrýt 100 % celého kódu, ale 100 % důležitých částí kódu. Takový přístup zajišťuje vyšší kvalitu výsledného produktu a snižuje množství potenciálních chyb v běhovém prostředí. Testování v kontextu rostoucí složitosti náročnosti HW a SW včetně vzájemného provázání je klíčovým prvkem a je třeba jej implementovat. I díky testování jsou pak cílová produkční prostředí včas zbavena chyb [7].

V příspěvcích na EOSS 2023 bylo zdůrazněno, že dohled a nasazení jsou klíčové součásti vývoje a správy zařízení. Zařízení IoT jsou v řadě případů vysoce distribuovaná, integrují cloud, fog a edge computing. Jedná se integraci HW infrastruktury a SW infrastruktury (senzory a platforma).

Příkladem takové komplexní realizace z EOSS 2023 je sledování a ochranu nosorožců v Africe před pytláky. Tito nosorožci dostanou umístěné sledovací IoT zařízení do vyvrtané díry ve svém rohu. Je-li zvíře ohroženo pytláky, začne utíkat, což detekují senzory a zachytí server, a proto dojde k prověřené situace.

Účastníci vyjádřili potřebu přehledného dashboardu, který poskytuje informace o stavu nasazení v jednotlivých prostředích a o průběhu nasazovacích procesů. Grafické rozhraní by mělo poskytovat práci s logy a monitoring. To by také mělo pomoct ulehčit práci s laděním a zpětným řešením incidentů. Pracovníci by také měli moci rychle reagovat na incidenty a minimalizovat náklady na vyřešení. Při řešení incidentů přednášející zmiňovali, že mezi základní metriky řadí uptime, downtime, stav hlavních služeb, teplotu HW, vytížení a statistku CPU + RAM, stav síťové komunikace, swap, konfiguraci jádra OS a zavaděče.

Vhodné metriky definuje DevOps oddělení v součinnosti s obchodním oddělením [8]. Systém by dále měl umožnit vzdáleně restartovat zařízení a zajištění tovární konfigurace. Dále se hovořilo o potřebě vizualizace závislostí a správy flotily zařízení IoT. Zaznělo také vhodné přirovnání k dobytku a mazlíčkům. Malé domácí / víkendové projekty jsou mazlíčci. Je jich málo a jejich péče nám nezabere tolik času jako dobytek (velké komerční projekty), který vyžaduje více času a musíme proto hledat vhodné nástroje, které nám s tím pomohou. Jak při nasazení na HW zařízení po celém světě, tak i pro QEMU staging prostředí je klíčové mít přehled o stavu prostředí, logech a možnost tato prostředí vzdáleně spravovat. Protože jsou nasazení většinou spjata silně s HW, je třeba monitorovat zdravotní stav těchto HW komponent, detekovat díky monitoringu anomálie a komponenty nahradit za nové funkční. Web UI vizualizace umožňuje monitorovat různé parametry a vrstvy.

Aktualizace a automatizované poskytování cílových systémů je také důležité, ať už pomocí aktualizací balíčků nebo nasazením nových obrazů. Monitoring a analýza dat jsou nezbytné pro zajištění integrity dat a spolehlivosti systému. U monitoringu je také trendem definování jednotlivých součástí jako kód. Obdobně jako u IaC pak mluvíme o přístupu Monitoring as Code (MaC). Dále se hovořilo o potřebě sběru dat od uživatelů v cílovém prostředí, monitoringu flotily a analýze dat z monitoringu s využitím technologií AI/ML. Zejména u edge computing lze využít lokální sběrač dat. Sníží se tak zátěž koncových stanic, zlepší se průchodnost sítě a bezpečnost dat. Žádaná je také implementace možností vzdálené

správy zařízení IoT, včetně aktualizací, konfigurace, monitoringu a diagnostiky. Ať už jde o sledování stavu, automatizaci správy nebo analýzu dat pro zajištění spolehlivosti a bezpečnosti systému, hraje správa flotily, monitoring a observabilita významnou roli. Mimo jiné je také častým obchodním požadavkem sledovat procentuální poměry starších verzích vůči nově vydané verzi. Zájem je také o alerting, který včas upozorní dle definovaných pravidel na nestandardní a potenciálně chybové stavy. Tyto nástroje se netýkají ale pouze firem a velkých týmů. O monitoringu mluvil také hobby vývojář, který tyto nástroje využívá, aby si ulehčil práci i na malých projektech. Vždy je však nutné vybrat vhodné nástroje. Například aktualizace embedded systému je výhodnější a udržitelnější řešit pomocí kompletního nasazení nového image namísto balíčkovacích manažerů (dnf, apt, apk aj.) [6].

Přednášející během konference také představili ukázková řešení, která využívají. Společným jmenovatelem všech představených řešením byla vysoká spolehlivost. Ve středních firmách jsou oblíbená self-hosted řešení využívající Tekton jako CI/CD nástroj provozovaný v K8s clusteru. Díky přítomnosti K8s se jedná o komplexní řešení. Jako celkově oblíbené řešení byl označen GitLab CI.

Důležitým aspektem takových řešení je ale nutnost integrace s dalšími nástroji. Nejde jen o základní nástroje, jakými je BitBake. Oblíbený integrovaný nástroj je také LAVA, což je nástroj určený pro automatickou validaci nového přírůstku a shlukuje řadu užitečných testovacích nástrojů. Je tedy třeba, aby nástroj podporoval široké spektrum integrací. Může se totiž stát, že některé nástroje přestanou lidem vyhovovat budou je chtít nahradit jiným. Na konferenci takto přednášející nahrazovali například nástroj Jenkins. Někomu vyhovuje stavba embedded OS pomocí Yocto Project, jinému zase nástroj Buildroot. Někdo jiný zase preferuje KAS, což je nadstavba nad těmito systémy, která zajišťuje vyšší míru abstrakce a deklarativněji přístup při sestavení [7].

Firma BMW využívá nástroj Zuul pro CI část DevOps cyklu. Přemýšleli i nad využitím K8s, ale to jim přinášelo více komplexity než užitku. Firmware / embedded OS nejprve nasazují a testují na testovacím HW. Dané HW je podobné tomu reálnému, ale nenachází se v automobilu, ale v laboratorním racku. Navíc je tento HW napojen ještě na další HW komponenty, které se starají o monitorování, testování a nasazování pro testovací účely. Pokud je vše v pořádku, proběhne nasazení do skutečného testovacího automobilu, kde se opět pro-

vede testování. V případě, že se nová změna chová standardně a byly splněny všechny předepsané testy, dojde ke schválení a proběhne nasazení do produkce. Tedy do automobilů zákazníků. Distribuce změn probíhá pomocí aktualizace v autorizovaném servisu. U novějších modelů pak zákazník většinou aktualizuje z domácnosti. Částečnou nevýhodou servisního řešení je fakt, že pokud zákazník neví o aktualizaci nebo nechce aktualizovat a nedostaví se do servisu pro aktualizace, zůstane mu stará verze SW. Tím se narušuje automatizace v propagaci změn a nemůžeme jako vývojáři počítat s jednotnou verzí všech nasazených instancí. Z pohledu bezpečnosti a funkčnosti je toto pro koncového zákazníka potenciálním rizikem. Přitom v BMW pravidelně provádějí i opravy a aktualizace své platformy. Díky CI sledují například jádro Linux a manuálně vybírají specifické verze, které následně implementují. Nad nasazeným SW poté provozují dohledovou platformu založenou na ELK stacku. Obchodní a vývojové oddělení tak vidí dohledová data ve webovém grafickém rozhraní nástroje Kibana. Na základě dat mohou plynule navázat na začátek DevOps cyklu, a to plánovací činnost [6].

Ve společnosti Netflix mají sofistikovanější automatizaci přijetí externích změn. Jejich automatizace pracuje s historií externího přispěvatele, jaké jsou jeho dílčí kroky, jaké je procento odhalení chyb, jaká je zpětná vazba od statické analýzy a případně výsledky A/B testování a aktuální procentuální podíl. Dle těchto údajů se stanoví důvěra na základě, které CI systém (ne)provede začlenění změn. Synchronizují takto například jádro Linux a upstream zavaděče Das U-Boot. Tyto změny se zpravidla integrují v podobě nočních sestavení (night builds). Jde o stovky sestavení za jednu noc a vzhledem k pravidelnosti se ve společnosti Netflix rozhodli provozovat tato sestavení na self-hosted infrastruktuře. Hlavní motivací bylo výrazné snížení výdajů, které měli s cloudovými prostředími a externími datacentry. Navíc některá jejich sestavení trvají více jak 8 hodin. Jako CI systém používají platformu GitLab, kde využívají princip Merge request pro validaci nových změn a funkcionalitu Approve (manuální schválení změny). Díky GitLab a výše zmíněné konfiguraci mohou paralelně sestavovat, a dokonce mohou paralelně sestavovat ten samý systém pro různé cílové stroje. Sestavené artefakty ukládají do firemního síťového úložiště, kde jsou sestavení dlouhodobě uchovávána. Následně se tyto artefakty využívají k testování a poté k ostrému nasazení. Jejich desky mají nad sebou ještě HW manager. Ten se stará o napájení dané desky, provádí validace chování dle standardního průběhu chování a také verifikuje. Při testování provedou

nejprve uvedení zařízení do továrního nastavení a až poté nasazují nový obraz. HW manager poté začne hlásit průběh testu a odesílá data o chování desky a její periferii [7].

Během diskusí na konferenci jsem vypozoroval velké úsilí, které firmy věnují pro aplikaci principů SW inženýrství. Hlavní motivací pro aplikaci těchto principů je potřeba vyšší míry efektivity oproti konkurenci a také snížení nákladů v dlouhodobém horizonu, tedy minimalizace technologického dluhu. Celkové workflow firem je pak udržitelnější. Po aplikaci principů by měl být kód výsledného produktu přehledný a udržitelný. Konkrétním příkladem z konference je firma Wind River, kde proběhla modernizace a adaptovali si cloud native nástroje. Zejména mladší vývojáři ocenili toto vylepšení. Během pandemie COVID-19 se řada z nich potýkala s problémy adaptace. Proběhla také standardizace vývojového prostředí, kde se adaptovalo Visual Studio Code. Firma připravila také rozšíření do tohoto nástroje. Nabízeny jsou funkce jako zvýraznění syntaxe, analýza vrstev a receptů s rozšířeními pro BitBake. Ve firmách tak vznikají jejich uzavřená řešení DevOps problematiky. Každá používá něco trochu jiného, dle svých preferencí. Současně ale na trhu existují také IoT platformy a frameworky. Jak komerční, tak i open source [6].

IoT platformy a frameworky existují a je jich hodně [6]. Nabízejí různé věci. Zejména velcí cloudoví poskytovatelé jako Microsoft Azure, Amazon AWS, Google Cloud, Tuya, Alibaba poskytují svá řešení. Tyto platformy mají standardně API pro možné integrace do systémů zákazníka, podporují různorodá zařízení a role serverů. Současně poskytují i optimalizované úložiště velkých dat. Poskytují také nástroje ke zpracování takových dat pro následnou analýzu, kterou taktéž usnadňují.

1.2.2 POTŘEBY Z ECS-SRIA

Strategický program výzkumu a inovací v oblasti elektronických součástek a systémů (ECS-SRIA) popisuje hlavní výzvy, priority a nezbytné úsilí v oblasti výzkumu, vývoje a inovací k jejich řešení v oblasti elektronických komponent, systémů a systémů systémů [5]. Rozhodl jsem se zohlednit jeho znění a analyzovat potřeby tohoto dokumentu.

Dokument tvrdí, že většina dílčích systémů bude sjednocena pod jednotnou platformu. To znamená, že platformy budou muset podporovat masivní množství koncových zařízení a zároveň poskytovat plnohodnotnou funkčnost původních dílčích systémů. Tyto platformy

by měly také podporovat možnosti integrace, a to zejména s IoT a se systémy systémů (SoS). Platformy si tedy budou muset poradit s distribuovanou architekturou, kdy koncová zařízení musí být také adekvátně zabezpečena. Zabezpečení se týká také i komunikace s těmito zařízeními. Platforma by také měla být schopná zaručit kvalitu koncových zařízení [5].

Dokument také zmiňuje, že je třeba více využívat CI na úrovni IoT zařízení a SoS. Výzvu zde tvoří fakt, že SoS je ve skutečnosti složeno ze závislých subsystémů. Mělo by tomu být naopak. Další výzvou je orchestrace IoT zařízení a jednotlivých subsystémů. Neméně důležitá je pak i otázka certifikace takových systémů a integrace umělé inteligence [5].

Metodiky DevOps, DevSecOps, ChatOps a další jsou standardně používány, aby řešily výše zmíněné otázky. Zde je třeba uvést, že tyto metodiky primárně vznikly k produkci čistě SW řešení. Nejsou tedy primárně určené pro vývoj embedded zařízení a HW. Navíc řada zařízení je již prodána a umístěna v produkčním prostředí zákazníka. Řada těchto zařízení a aplikací je velmi specifikována a vyladěna na konkrétní účel použití. Dle principů uvedených metodiky bychom měli držet kopie každého vytvořeného systému. To je ale dlouhodobě neudržitelné a došlo by k porušení [5].

V EU je aktuálně nedostatek vhodných platforem, které začleňují do jednotného ekosystému embedded aplikace a zařízení různých poskytovatelů. Cílem je tedy toto vyřešit. Proto, aby byl cíl naplněn, je třeba poradit si s řadou dílčích výzev spojených zejména se zajištěním odpovídající funkčnosti integrovaných systémů. Nastíněným řešením, které článek uvádí, je architektura mikro služeb. O to více je třeba věnovat pozornost kvalitativním vlastnostem (výkon, bezpečnost, spolehlivost). I bez implementace mikro služeb je bezpečnost ve firmách často opomíjená, a pokud se implementuje, tak je to zejména z funkčního hlediska, kdy novější systémy pro to, aby vůbec fungovaly, vyžadují nějaké minimální zabezpečení. Tímto minimálním zabezpečením je například požadavek na šifrovanou komunikaci a protokol HTTPS. Implementace i pouze těchto nutných mechanismů zabezpečení bývá zejména pro menší firmy nákladná. Absence jednotného přístup implementace vyžaduje více než základní znalosti problematiky. Právě integrační a orchestrační platformy by mohly tuto problematiku řešit samy a nebylo by potřeba zásahů od uživatele. Takto by i platforma mohla pomoci standardizovat a navést drobné implementace v případech, kdy platforma není třeba. Dle dokumentu by platforma měla být pro uživatele prevencí před dílčími problémy a

výzvami, se kterými se setkají při vývoji a použití jejich proprietárního řešení. Také by platforma měla poskytnout řadu integrací, které by jinak musel uživatel implementovat a orchestrovat sám.

Dokument doporučuje vzít všechny tyto informace v potaz již při návrhu platformy a vhodně volit architektu. Uvádí, že architektura je nutností pro zajištění funkčního systému. Platforma by měla používat standardizované přístupy a aplikovat ověřené algoritmy a vzorce. Problematiku by platforma měla řešit co nejvíce automatizovaně, a to platí nejen pro konfiguraci systémů, ale i další inženýrské procesy. Automatizované mechanismy pomohou nejen snížit iniciativní náklady, ale také standardizují interní postupy a včas ukážou na možná rizika a obavy a tím se zvyšuje i pravděpodobnost na minimalizaci potenciálních ztrát. Cílit by se také mělo na usnadnění komunikace integrovaných komponent a dílčích systémů. Při vývoji platformy je třeba při implementaci zdůraznit kvalitativní vlastnosti. Platforma by měla být vyvíjena za podpory principů automatizačního a softwarového inženýrství. Její funkčnost by měla být validována pomocí V&V modelu. Jedná se o model, který je složen z vrstev, kdy na každé vrstvě probíhá validace a verifikace dané změny z pohledu abstrakce systému. Na nejnižší vrstvě se nachází kód a k němu přiřazené jednotkové testy. Na nejvyšší úrovni jsou pak uživatelem definované požadavky, kterým přísluší akceptační testy. Mezi jsou další vrstvy, které podobně validují a verifikují. Dokument upozorňuje na možné využití AI pro validaci a verifikaci [5].

Výzvou, kterou dokument vyzdvihuje, je vyřešení otázky průběžné integrace (CI) embedded SW. Možným řešením podpory systémové integrace (integrace HW a SW) a společného vývoje HW/SW by mohl být model založený na digitálních dvojčatech. Jedná se o digitálně modelovaný HW, který následně simuluje chování HW. Můžeme tak integrovat nové technologie a tyto technologie ověřit právě na digitálním dvojčeti a až poté na reálném HW. Průběžná integrace musí být automatická a pracovat s rozdílnými architekturami, modely a platformami. Průběžná integrace by měla být schopna validovat zadanou konfiguraci a novou integraci, zejména v oblasti spolehlivosti a práce se sdílenými zdroji. K tomu by měla vývojovému týmu pomoct vizualizace těchto informací. Při průběžné integraci by mělo být použito standardizované řešení orchestračních a integračních postupů. Mělo by tedy dojít k oddělení a vymezení prostoru SoS a IoT. Platforma by tedy měla být schopna nepřetržité

integrace a kontroly kvality u Cyber-physical systems (CPS) díky automatizačnímu modelu. CPS jsou úzce spjaté HW a SW komponenty.

Pro zajištění životnosti embedded zařízení je třeba aktualizovat a udržovat zařízení. K tomu se standardně používají systémové aktualizace. Proces aktualizace embedded zařízení musí být bezpečný a spolehlivý. Případně musí být nastaveno automatické obnovení do předchozí verze. Výsledný SW a jeho bezpečnost je také nutné certifikovat. A to i průběžně s každou novou aktualizací a změnou. Zejména u kritických aplikací SW je toto nutnost. Uvedený model validace a verifikace s digitálními dvojčaty toto zajistí. Pro hladší průchod a úsporu lze opět využít AI [5].

1.2.3 POTŘEBY TUL

Analyzoval jsem také potřeby týmu působícího na NTI FM TUL. Tento malý tým se konkrétně zabývá vývojem embedded OS Linux za pomocí projektu Yocto Project. Například v době rešerše se tým konkrétně zabýval vývojem OS pro automotive a také vyvíjí embedded OS určený pro použití v oblasti aerospace. Formou rozhovorů jsem zjistil potřebu jednotné platformy a workflow pro tento vývoj.

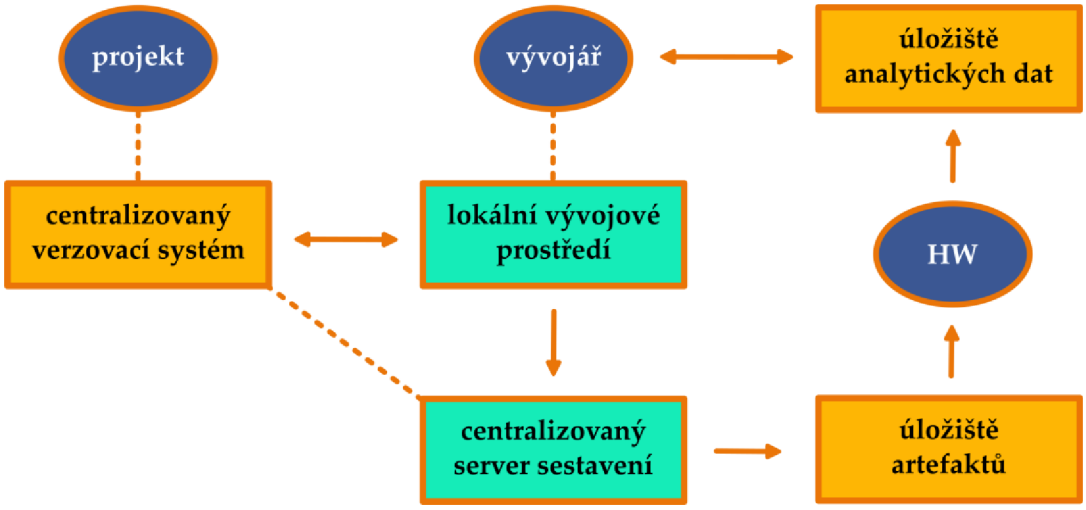
Platforma by měla být jako samostatný nástroj ve verzovacím systému. Její nasazení by mělo být popsáno pomocí IaC a poté by měla být platforma snadno nasazena v lokální infrastruktuře. Charakter infrastruktury není předem znán. Může se jednat jak o self-managed virtualizační prostředí, tak i cloud poskytovatele (Azure, AWS aj.). Platforma bude pracovat s externím verzovacím systémem, kde se budou nacházet jednotlivé projekty. Tedy konkrétně OS pro projekt ESA a OS pro projekt BUSKIT. Budou zde také akceptační a systémové testy definované skrze Ansible. Současně budou ještě v daném verzovacím systému také repozitáře s aplikacemi, které budou do daného sestavení OS přidávány. Tyto aplikace budou taktéž vyvíjeny na TUL.

Vývojář bude pracovat ve svém lokálním vývojovém prostředí. Toto prostředí si musí sám připravit. Následně si vše potřebné naklonuje, provede úpravy a změny pošle do centrálního repozitáře. Takto může obecně pracovat n vývojářů na daném projektu. Pro oddělení verzí lze pak využít vývojové větve. Nový přírůstek bude nutno jednoznačně identifikovat, a to včetně autora změny.

Poté by měla automatizace provést sestavení daného OS. Identifikátor sestavení bude rozšíření předchozího identifikátoru změny. Na sestavovací části infrastruktury, kterou by měla platforma zajistit, se může provádět současně více sestavování. Je tedy vhodné, aby platforma zajistila správnou orchestraci těchto sestavení a průběh byl automatizován. Je také žádoucí, aby systém sestavení uchovával a sdílel stažené soubory pro jednotlivá sestavení a cache. Výstupem sestavení by měl být binární soubor / obraz.

Soubor je poté třeba nasadit do úložiště (artifactory), odkud bude distribuován. Samozřejmostí je dlouhodobé uchovávání obrazů a jednoznačné identifikování těchto obrazů. Automatizace by poté měla nasadit takový obraz na cílový HW. K tomuto transferu je možno využít různé přístupy. Mezi zmiňované možnosti patří proxy transfer s využitím PC, které provede zápis do zařízení, případně na SD kartu. Jedná se o push architekturu. Dále pak je možno provést síťové nasazení pomocí TFTP. Jedná se o pull architekturu, kde samo zařízení je zodpovědné za sledování změn a následný proces stažení a použití nového obrazu. Důležitou vlastností nasazení musí být opět jednoznačná identifikovatelnost v rámci koncového zařízení.

Běžící zařízení poté bude odesílat data. Data mohou být zpravidla testovací, produkční, analytická, monitorovací, ladicí a aplikační. Tato data je třeba odesílat do příslušných úložišť. Tým si je následně potřebuje prohlížet v přehledných vizualizacích dle identifikace sestavení / nasazení.



Obrázek 1.4: Schéma potřeby NTI FM TUL

1.2.4 SHRNUTÍ POTŘEB

Závěrem analýzy jsem se rozhodl shrnout a vybrat nejzásadnější potřeby, na které se budu dále v práci soustředit. Jsou následující:

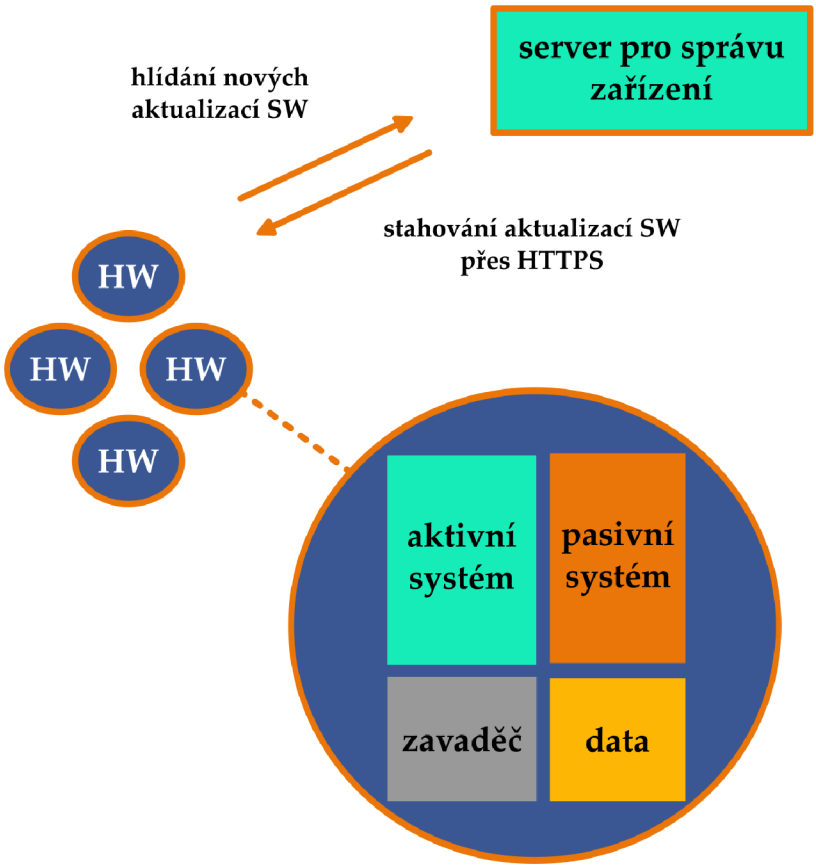
- Evropská open source platforma pro správu infrastruktur pro vývoje embedded SW.
- Pokrytí celého DevOps: plánování, kódování, sestavení, testování, vydání, nasazení, operativa a dohled.
- Podpora implementace bezpečnosti a spolehlivosti.
- Každé sestavení musí být jednoznačně identifikovatelné.
- Platforma musí optimalizovat a orchestrovat paralelní sestavování různých i shodných projektů.
- Během kompilace nesmí dojít ke kontaminaci sestavovacího prostředí.
- Platforma musí mít jednoduchou administraci.
- Platforma bude poskytovat již zabudované integrace. Zároveň musí být rozšiřitelná komunitou o jimi potřebné integrace.
- Platforma musí být testovatelná a pokryta základními testy. Použití platformy musí být pro koncové uživatele podpořeno přehlednou dokumentací.

1.3 EXISTUJÍCÍ ŘEŠENÍ

V současné době existuje řada řešení, která se zejména soustředí vždy na určitý kousek. Tyto nástroje nejsou navrženy tak, aby autonomně pokrývaly celý cyklus vývoje dle DevOps. Řadu věcí je třeba dělat manuálně. Většina existujících platforem není open source, případně není evropského původu. Tato řešení netvoří platformu splňující požadavky dle kapitoly č. 1.2.4.

Již na konferencích jsem se dozvěděl o nástroji Mender, kde jeho prezentace měla místo na hlavní části programu. Nástroj poskytuje zejména možnosti nasazení a bezdrátového nasazení pro IoT (OTA) jak startupům, tak i velikým firmám. Zde nástroj nabízí pokročilé typy nasazení, jakými je například A/B. Samozřejmostí jsou i automatické návraty k předchozí verzi (rollback), pokud při aktualizaci došlo k chybě. Společnosti prezentují ná-

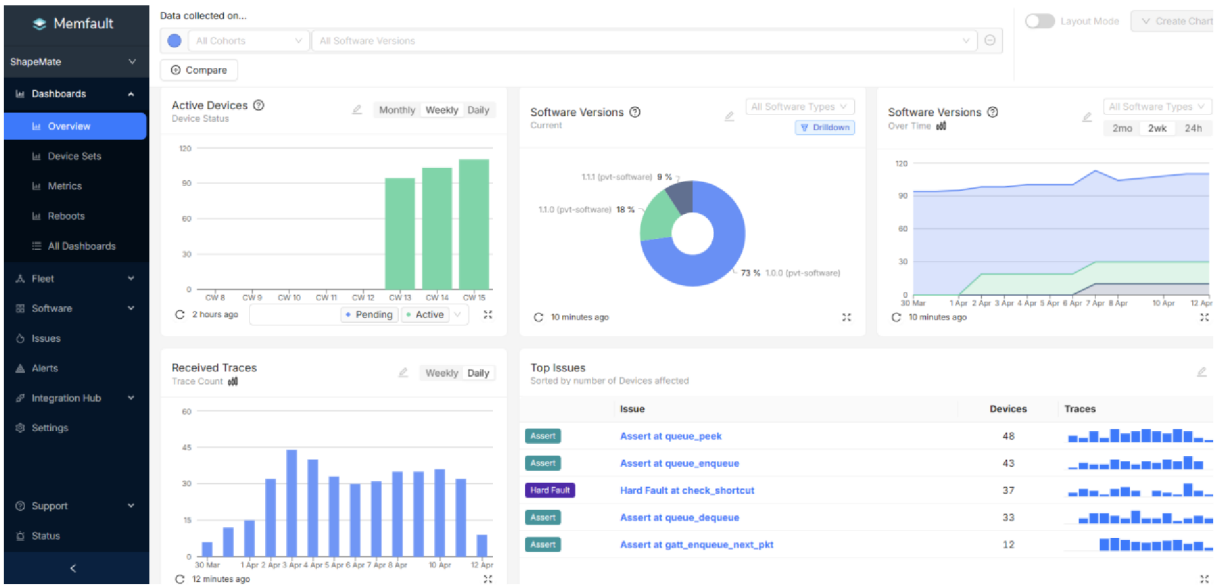
stroj jako spolehlivý a bezpečný. Díky API je snadné Mender implementovat a komunikovat s ním strojově. Platforma také nabízí vzdálenou správu a dohled nad běžícím nasazeními. Lze ji provozovat jak v režimu self-managed, tak i jako hostovanou variantu. Z toho tedy plyne, že se nejedná o komplexní platformu, ale pouze o dílek, který lze využít zejména na CD část DevOps cyklu. Navíce je pouze jeho část vedena jako open source. Pro pokročilejší funkce je třeba platit měsíční poplatky. Mezi pokročilé funkce nástroje Mender patří i aplikace v zařízen, či vzájemné ověření pomocí TLS. Jedná se ale o vyspělý projekt určený k produkčnímu použití [11].



Obrázek 1.5: Mender a jeho princip aktualizací

Platforma Memfault poskytuje podobné služby jako Mender. Navíce přináší velmi propracované dohledové nástroje a uživateli poskytuje přehledné automaticky generované vizualizace naměřených dat. Je ale spíše určena pro použití ve velkých společnostech a korporátech. Primárně je určena ke správě zařízení IoT v řádech stovek tisíc a více. V tomto pří-

padě se jedná o komerční produkt bez open source verze [12]. Jeho hlavní přednost spatřuji v jednoduchém a přehledném způsobu vizualizace důležitých monitorovacích dat. Tyto vizualizace jsou uživateli poskytnuty automaticky a šetří tak jeho náklady. Obdobě existují další takové platformy, jakými jsou Mender a Memfault. V základu jsou podobné, ale liší se zaměřením a funkcionalitami. Například platforma Balena nabízí spíše funkce týkající se správy zařízení na úkor pokročilých funkcí monitoringu. Naopak však nabízí open source edici. Celkově však tyto platformy nesplňují zásadní požadavky, které jsem vytyčil na základně analýzy požadavků.



Obrázek 1.6: Memfault a jeho přehledná vizualizace dat

Poskytovatelé cloud prostředí mají také svá řešení. Jedná se o kompletně uzavřené platformy, které jsou plně závislé na daném cloud poskytovateli. Migrace řešení mezi jednotlivými poskytovali, případně do vlastní infrastruktury není poskytovali podporována. Společnost Amazon má své řešení, které využívá koncern VW či iRobot. Jedná se o služby pro koncová zařízení IoT, průmyslové procesy a chytrou domácnost. AWS IoT se věnuje oblastem nasazení do koncových zařízení a následnému dohledu. Zároveň ale také nabízí možnost využít služby pro analýzu dat, která koncová zařízení produkují. Lze tak analyzovat například výrobní procesy a ty následně optimalizovat. Uživatel si jen definuje, jaké služby

chce využívat a infrastruktura v cloudu se mu sama sestavuje a propojuje [13]. AWS dále pak nabízí DevOps platformu pro vývoj embedded řešení. Toto řešení splňuje většinu požadavků na platformu. K dispozici je také řada možných integrací, jako jsou virtuální ARM HW, GitLab, Azul a další [14]. Nejsou však splněny požadavky na open source a evropský původ. Také už roste celková komplexita pro uživatele, kteří se nevyznají v cloud prostředích.

Možnou cestou je také využití platformy Azure DevOps. Nejedná se o produkt cílený na vývoj pro embedded řešení. Jedná se o obecnou DevOps platformu v cloudu společnosti Microsoft. Platforma zajišťuje kompletní DevOps vývojový cyklus pro SW řešení. Lze ji tedy pomocí konfigurace upravit tak, aby sloužila potřebám vývoje pro embedded. To ale vyžaduje úsilí a znalost platformy. Navíc se nejedná o open source řešení [15].

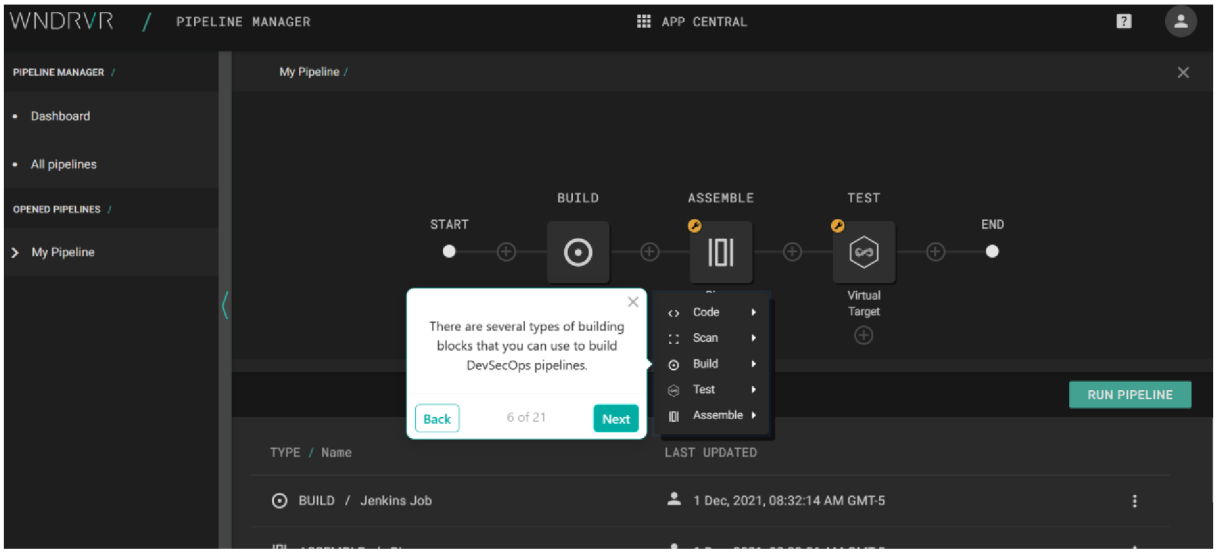
Jako možné řešení lze zvažovat i DevSecOps platformu GitLab. Platforma poskytuje kompletní DevOps cyklus. Uživatel zde musí věnovat pozornost konfiguraci a CI/CD automatizace musí definovat pomocí YAML souboru. Jsou zde i podpůrné funkce jako třeba CRM a kanály pro zpětnou vazbu. Ačkoliv není primárně platforma určena pro vývoj embedded OS, lze ji k tomuto účelu nakonfigurovat a přidat integrace, které toto umožní. Poté už lze těžit z benefitů, kdy platforma řeší i otázky bezpečnosti a spolehlivosti. Nicméně je zde třeba provádět konfigurace [16]. Platforma má i svou open source verzi. Pro splnění požadavků je třeba, aby se nad touto platformou udělala další nadstavba, která zajistí automatizaci a bude danou infrastrukturu s GitLab poskytovat jako nakonfigurovanou službu.

Obdobným řešením je i GitHub, ale zde v současné době chybí řada funkcí, které by zajistil plnohodnotný DevOps cyklus. GitHub není vhodný pro firemní použití a místo něj je vhodné využít Azure DevOps, případně GitLab.

Zuul je systém pro ověřování projektů. Nejde jen o testování navržených změn, ale jde i o testování budoucího stavu větví a repozitářů s mnoha současnými změnami a jejich závislostmi. Zuul komunikuje s různými systémy pro revize kódu, spouští úlohy a zobrazuje výsledky. Nabízí uživatelské rozhraní i možnost spouštění úloh. Zuul lze provozovat jak v self-managed infrastruktuře, tak i v různých cloud prostředích. Jedná se o open source projekt. Zuul pokrývá DevOps cyklus a nabízí funkcionalitu pro plánování, kódování, sestavování, testování, vydání, nasazení, operace a dohled. Integruje se s různými systémy a nabízí pře-

hledné uživatelské rozhraní. Je tedy možné využít Zuul jako klíčovou součást plánované platformy. Nenahradí však celou platformu [17].

Skoro všechny požadavky splňuje platforma Wind River Studio. Překážkami použití jsou neevropský původ a uzavřený kód. Wind River Studio je cloud-native platformou s podporou celého DevOps cyklu pro embedded zařízení. Platforma podporuje automatizaci pro sestavení testování a nasazení software. K tomu uživateli stačí pouze stavební bloky, které dle potřeby pouze správně poskládá. Platforma na pozadí zajistí jak infrastrukturu, tak i celý proces. To platí jak pro malé projekty, tak ale i pro komplexní projekty složené z mnoha koncových zařízení. Platforma nabízí také snadné zabudované sestavení kvalitního RTOS jménem VxWorks. U Linux OS uživatele potěší propracovaný systém sestavení, který je snadný na použití a je vyladěn tak, aby byl co nejrychlejší a podporuje orchestraci několika paralelních požadavků na sestavení. Uživatel nemá starost s infrastrukturou a věcmi na pozadí, vše je mu poskytnuto jako cloud služba. Platforma také v rámci cloudu nabízí možnost poskytování fyzického HW pro účely vývoje, testování a nasazení. K dispozici jsou také i digitální dvojčata. Při automatizaci se využívá mimo jiné i umělá inteligence. Platforma je rozšiřitelná a podporuje široké spektrum integrací. V ceně, kterou uživatel pravidelně platí, je dokumentace doplněná o e-learningové kurzy a lze dokoupit instruktorem vedená školení. Samozřejmostí je podpora na komerční úrovni [18].



Obrázek 1.7: Přehledné sestavování automatizace ve Wind River Studio

2 NÁVRH

Na základě požadavků vyplývajících ze zadání a z analýzy potřeb cílové skupiny jsem pokračoval navržením řešení a jeho součástí. Zprvu jsem si ujasnil, jakým způsobem bych chtěl problematiku řešit. Na tuto úvodní otázku jsem pak navázal průzkumem, jak vyřešit jednotlivé dílčí otázky, které na tu původní navazovaly. Vždy jsem postupoval definicí otázky a jaké parametry zohledňuji při jejím zodpovězení. Následně jsem provedl rešerši vhodných řešení. Tato řešení jsem poté porovnal a zjistil, zda mé požadavky splňují. Na závěr jsem navrhl vodné řešení vyplývající z předchozí komparace. To mi pomohlo si před implementací zodpovědět důležité otázky, jako je volba technologií architektury a dalších. Díky této separaci jsem se mohl v implementační fázi maximálně soustředit na implementaci samotnou.

2.1 ZPŮSOB ŘEŠENÍ

Díky úvodní rešerši jsem měl základ pro rozhodování nad způsobem řešení zadání práce. Zvažoval jsem použití existujících řešení, která by problém řešila. Také jsem hodnotil i tvorbu vlastního nástroje. V rešerši jsem zároveň upozornil na zásadní nedostatky zmíněných řešení. Pro splnění požadavků dle kapitoly 1.2.4 jsem musel zvážit tvorbu vlastního řešení – tvorbu vlastní open source specializované platformy. Respektive vytvořit framework, kdy by použití takového frameworku znamenalo vyřešit problémy uvedené v úvodní analýze.

Řešení jsem mohl realizovat pomocí existujících IaC nástrojů, nebo pomocí naprogramování nástroje, který by mou platformu vytvářel a konfiguroval. Stanovil jsem si tedy kritéria, dle kterých jsem provedl komparaci v Tabulka 2.1. Primárně jsem požadoval, aby nástroj dokázal autonomně pokrýt všechny úkoly nad platformou a maximálně odstínil uživatele od implementace. Řešení by tedy mělo být silně deklarativní. Uživatel by měl říct, jakými zdroji disponuje a jaké jsou jeho potřeby. Například by měl předat informaci o své privátní infrastruktuře a informaci o tom, že chce vyvíjet embedded OS Linux za pomocí projektu Yocto Project. Nástroj by se poté měl postarat o zbytek definice, konfigurace a propojení infrastruktury. Celkové použití nástroje by mělo být jednoduché. Tedy jednoduchá deklarace, ale i příprava prostředí pro nástroj, jeho nastavení, instalace jeho závislostí a přenositelnost. Řešení by mělo být schopno na pozadí definovat infrastrukturu a tuto infrastrukturu

sestavit. Zároveň je třeba takovou infrastrukturu správně nakonfigurovat. To řešení musí také umět. Řešením by také mělo být open source a ideálně evropského původu dle úvodní rešerše. Dále by řešení mělo být optimalizováno z pohledu rychlosti sestavení a mělo by paralelizovat jednotlivé kroky, pokud je to možné. Řešení by mělo hlídat správnost definice a také propojení jednotlivých dílčích částí. Například hlídání vazby projektu na verzovacím systému. Řešení by mělo být schopné úzce cílit na problematiku definovanou úvodní rešerší a mělo by dokázat splynutí požadavků z kapitoly č. 1.2.4.

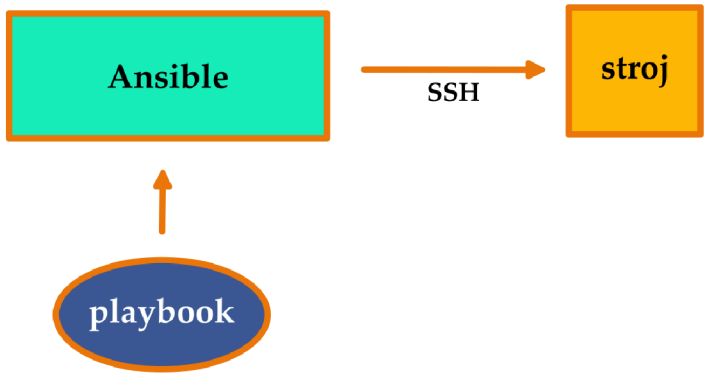
Tabulka 2.1: Porovnání IaC nástrojů a vlastního řešení

	Ansible	Terraform	Chef	Puppet	Pulumi	Vagrant	Vlastní
Silně deklarativní	●	●	✗	●	●	✓	✓
Jednoduché použití	●	●	✗	✗	✗	●	✓
Definice platformy	●	✓	✗	✓	●	●	✓
Konfigurace platformy	✓	✗	✓	✓	✓	●	✓
Open Source	✓	●	✓	✓	✓	✗	✓
Optimalizace rychlosti	●	●	●	✓	✓	●	✓
Kontrola vazeb	✗	✓	●	●	✓	●	✓
Možnost zacílit	●	✗	●	●	●	✗	✓

Nástroj Ansible je automatizační nástroj, který využívá deklarace soubory YAML. Je napsán v jazyce Python. Jedná se o open source projekt a má silnou uživatelskou základnu. Je standardem ve světě DevOps. Uživatel si definuje automatizace sestavení a konfigurace infrastruktury, ale i servisní úkony pomocí konceptů Ansible Playbook a rolí. Podporuje také orchestraci úkolů. Silnou deklaraci by šlo zajistit pomocí dopsání vlastních modulů. Ansible vyžaduje interpret jazyka Python včetně doinstalovaných balíčků a závislostí dle užitých modulů. Toto musí uživatel instalovat sám. To samé platí i o hlídání verzí Python. Nestačí tedy pouze stáhnout jeden binární soubor, který vše připraví za nás. Řešením by mohla být distribuce vývojového kontejneru, kde by bylo vše připlavené k použití. Ansible je primárně navržen na konfiguraci, nastavování a administrační úkony. Lze jím infrastruktury také definovat, ale to není jeho primární poslání. Ansible nehlídá logické zavilosti. Například zda před založením projektu v platformě GitLab již platforma GitLab existuje. To si musí uživatel sám dodefinovat [19]. Ansible využívá interpretovaný jazyk Python, který je pomalejší než jiné jazyky orientované na rychlost, mezi které se řadí třeba Golang [20]. Uživatel také sám musí definovat orchestraci [19]. Pomocí Ansible by šlo zacílit na problematiku této práce, nicméně by nebyly přirozeně splněny všechny požadavky a řešení by muselo být formou zaobalení tohoto nástroje do mnou vytvořeného nástroje.

Terraform je nástroj pro definici infrastruktury vyvinutý společností HashiCorp. Je to rozšířený a vyspělý IaC DevOps nástroj. Infrastruktura je popsána konfiguračním souborem s proprioertální jazykem HCL. Práce s Terraformem začíná definicí infrastruktury v konfiguračních souborech. Poté se provádí inicializace projektu, která načte potřebné providery a moduly. Následuje fáze plánování změn. Po schválení změn je možné změny automatizovaně aplikovat. Terraform lze rozšiřovat pomocí modulů, které umožňují opakované použití a abstrakci infrastrukturních konceptů. Lze také implementovat vlastní poskytovatele. Zásadním nedostatkem je jeho orientace na definici a poskytování infrastruktury. Nemá zabudované mechanismy pro plnohodnotné konfigurace. To lze vyřešit v praxi hojně užívanou symbiózou s nástrojem Ansible. Terraform definuje, Ansible konfiguruje [21]. Druhým zásadním nedostatkem je uzavřená licence. Původně se jednalo o open Source projekt. V roce 2023 se však licence změnila na uzavřenou. Komunita však vytvořila z poslední open source verze nástroj Open Tofu, který je plnohodnotnou náhradou a je zaštítěn Linux Foundation

[22]. Samostatně tedy nástroj nelze využít ke splnění požadavků a ani jím nelze přímo zacílit na konkrétní problematiku.

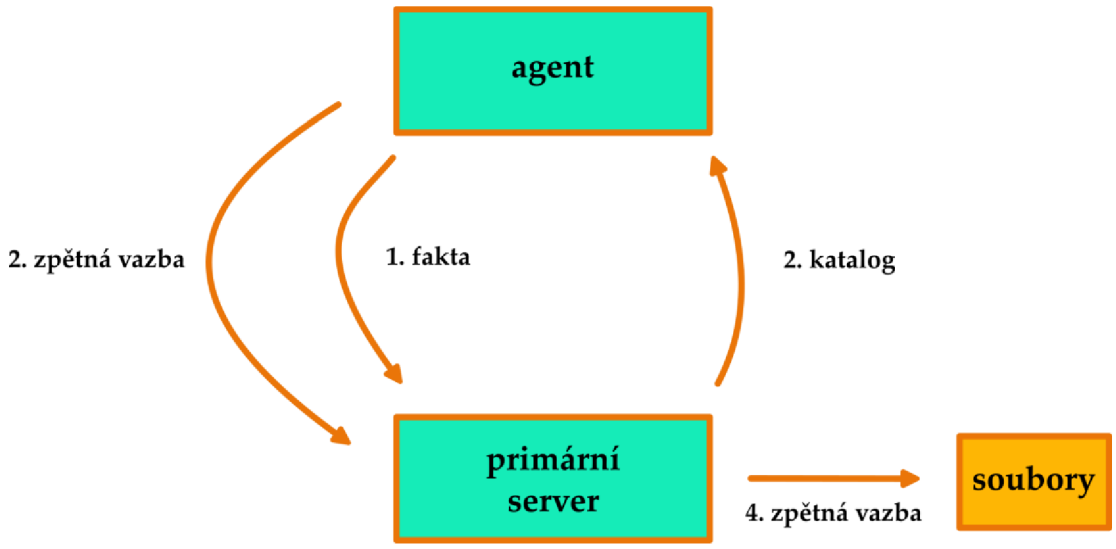


Obrázek 2.1: Princip fungování Ansible

Chef je další nástroj pro automatizaci infrastruktury. Pomocí IaC umožňuje správu konfigurace. Na rozdíl od Terraform, který se zaměřuje spíše na infrastrukturní úrovně, Chef se specializuje na konfiguraci softwaru na jednotlivých strojích. Definice jsou drženy v textových souborech. Ty popisují požadavky na instalaci SW na daném stroji, také jsou zde popsány služby, které mají být spuštěny, a i jaké mají být konfigurační soubory a nastavení. Chef používá architekturu klient – server. Centrální server uchovává konfiguraci a na klientech je nainstalován Chef klient, který komunikuje se serverem a aplikuje konfiguraci na daném stroji. Pomocí rolí lze stavět vysoce deklarativní řešení. Chef lze rozšiřovat pomocí komunitních a vlastních pluginů a rozšíření. Jedná se tedy o robustní nástroj, který má ale zbytečně složitou architekturu. Navíc současný trend ukazuje na jeho zmenšující se uživatelskou základnu [23].

Jako horký kandidát se ukázal i Puppet. Jedná se open source platformu určenou především pro automatizaci konfigurace systémů, ale pomocí této platformy lze i přehledně definovat infrastrukturu. Uživatel definuje požadovaný stav pomocí jazyka DSL. Je to jazyk, který vznikl jako součást platformy. Puppet si takto definovanou infrastrukturu vnitřně přeloží a poté kontroluje a případně mění na požadovaný stav. Puppet je ale platforma, ne nástroj. Je třeba tuto platformu nejprve ve své infrastruktuře připravit. Používá architekturu

master – slave a vyžaduje, aby uživatel držel definice v nějakém repozitáři. To je u ostatních nástrojů vždy rozhodnutí vývojáře a není to povinnost [24].



Obrázek 2.2: Architektura Puppet

Nástroj Pulumi poskytuje alternativní pohled na danou problematiku. Odlišuje se definicí požadovaného stavu infrastruktury. Uživatel totiž definuje infrastrukturu jedním z jazyků Node.js, Python, Golang, C#, nebo Java. Pokud uživatel preferuje YAML, může použít jej. Spolupracuje s více jak 120 cloudy, technologiemi, nástroji a platformami. Díky tomu, že je open source, je možné integrovat další platformy dle potřeby. Jeho přednostní je také rychlost a efektivita, a to také díky tomu, že je napsán v jazyce Golang. Lze jej také provozovat v podobě online služby. Poskytuje také grafické rozhraní. Pulumi by šel použít jako řešení, ale opět by bylo nutné nad ním sestavit strukturu, která by poskytovala vyšší míru deklarativity a zároveň použití pro koncové uživatele by znamenalo nutnost znalost programování [25].

Mimo Terraform společnost HashiCorp vyvinula také uzavřený produkt Vagrant. Jeho použití je jednoduché a silně deklarativní. Na pozadí lze nadefinovat infrastrukturu včetně její konfigurace. Produkt je ale učen k přenositelnosti vývojových prostředí ve virtualizačních platformách [26]. Nelze jej tedy převést na mnou potřebné řešení.

Jak plyne z Tabulka 2.1, vyplatí se mi vytvořit vlastní řešení, tedy nepoužít již existující IaC DevOps nástroje a platformy. Budu tak moci cílit co nejpřesněji na splnění potřeb a zároveň jednotně pokrýt celý proces.

2.2 PROGRAMOVACÍ JAZYK

Jelikož jsem se rozhodl pro tvorbu vlastního nástroje, musel jsem učinit volbu jazyka, ve kterém takový nástroj napíši. Začal jsem definicí základních požadavků, které jsem kladl na výsledný jazyk. Dále jsem našel jazyky, které se běžně používají na obdobné typy úloh, či které by mohly být použity na mou problematiku. Výstupem je pak Tabulka 2.2, která porovnává tyto jazyky na základě definovaných požadavků.

Tabulka se věnuje požadavkům na jazyk, který jsem hledal. Začněme popisem tak, aby bylo jasné, co jednotlivé parametry znamenají, a také přibližně v jakém rozsahu je parametr zohledněn. Základní problematikou je tedy DevOps nástroj, který čte / generuje konfiguraci a generuje sled činností včetně návazností. Nástroj dále generuje úkoly, které vykonává. Nástroj musí umět komunikovat s externími systémy a využívat principy distribuovaného programování. Úkoly mají vzájemné vazby a některé lze vykonávat paralelně.

Jednoduchost je parametr, který vysvětluje, zda je možné jednoduše a intuitivně napsat výše zmíněný nástroj v daném jazyce. Také zda je nutné využívat pokročilé postupy a konstrukce.

Přehlednost vyjadřuje, zda výsledný kód bude přirozeně snadno čitelný, a to nejen pro autora, ale i pro budoucí možné přispěvatele.

Snadná přenositelnost říká, zda bude možné nástroj snadno přenášet a spouštět na různých platformách a architekturách a zda je tato přenositelnost dlouhodobě udržitelná.

Bohatost toolchainu říká, zda má jazyk nějaké oficiální podpůrné nástroje, mezi které se například řadí build system, linter, formátovač, nástroje na statickou analýzu kódu a další.

Snadné škálování vyjadřuje, zda je možné snadno vytvořit nové vlákno a v něm spustit nějaký kus kódu, či zda je nutné psát rozsáhlejší syntaxe a myslet nad vláknem více komplexně. Nebo dokonce myslet nad škálováním na jiných úrovních než v kódu.

Tabulka 2.2: Porovnání programovacích a skriptovacích jazyků

	Golang	Python	Bash	PS	Perl	Java	C++
Jednoduchost	✓	✓	●	●	●	●	✗
Přehlednost	✓	✓	●	✓	✗	●	✗
Přenositelnost	✓	●	●	●	●	✓	●
Bohatý toolchain	✓	✗	✗	●	✗	●	✗
Snadné škálování	✓	✗	✗	✗	✗	●	✗
Rychlost	✓	✗	✗	✗	●	●	✓
Statické typování	✓	✗	✗	✗	✗	✓	✓
Preference struktur	✓	✗	✗	✗	✗	✗	●
Rychlost vývoje	✓	✓	✓	✓	✓	✗	✗

Rychlost vyjadřuje, zda je vykonávání kódu oproti ostatním jazyků rychlejší. Připomeňme, že bereme v potaz již zmíněnou situaci aplikace jazyka na DevOps nástroj.

Statické typování říká, zda je použita statická typová kontrola, či nikoliv. Tento požadavek eliminuje chyby v běhovém prostředí, kde může dojít k nechtěnému přetypování.

Preference struktur ukazuje, zda jazyk používá obecně spíše struktury, třídy, či něco jiného. Jazyk by měl používat struktury místo tříd. Pokud jsou třídy přítomny, pak by měly být struktury preferovány a výhradně používány.

Závěrečný řádek tabulky vyjadřuje, zda by bylo rychlejší napsat nástroj v daném jazyce, či v ostatních porovnávaných jazycích. To platí i pro rychlost přidání dalších rozšíření a funkcí.

Jak z tabulky plyne, jazyk Golang splňuje všechny požadavky a je tak nejlepším kandidátem. Z porovnání je pak další v pořadí Java, která je nicméně více těžkopádná. Fragmenty kódu jsou komplexnější a obsáhlejší než v jazyce Golang. Python, umístěný na třetí pozici, je naopak jednoduchý, čitelný a nástroj by v něm šel napsat rychle, nesplňuje ale další požadavky, mezi kterými je i zásadní škálovatelnost a rychlost. Další v pořadí je pak C++, který poskytuje výkon, ale práce v něm není jednoduchá, proces psaní kódu není rychlý, a to mimo jiné může i zapříčinit chybování vývojáře. Na dalších místech se umísti PowerShell a Bash. Na první pohled by se mohlo zdát, že tyto jazyky by mohly stačit a splnily by vše, co nástroj vyžaduje. Museli bychom ale slevit z řady požadavků (škálovatelnost, rychlost, statické datové typy apod.). Poslední místo obsadil Perl. Hlavním důvodem není, že vyšel z módy a je označován za zastaralý a nemoderní. Použití jazyka Perl by zejména vedlo na pomalejší kód, který nepůjde dobře škálovat, jeho vývoj bude v dlouhodobém horizontu komplikovaný nejen pro autora, ale i pro případné další přispěvatele [20].

Mimo jiné jsem se také inspiroval společností Google. Ta používala na DevOps nástroje převážně jazyka Python. Nicméně došli k závěru, že dlouhodobě udržitelná přenositelnost a synchronizace verzí interpretů jazyka Python na koncových strojích pro ně byla náročná. Navíc nebylo možné snadno škálovat nástroje a také potřebovali vyšší výkon. Zvažovali použití jazyka C++, ale ten byl na dané úlohy příliš komplexní. Kód nebyl dobře a rychle čitelný. Navíc k jazyku neexistuje nativní a zároveň jednoduchý toolchain. Proto se v Google rozhodli vytvořit vlastní jazyk Golang, který by splňoval jejich požadavky na moderní jazyk, který splňuje výše uvedené požadavky [20].

Při volbě jazyka, ve kterém je nástroj Oasis napsán, jsem se rozhodoval mezi řadou jazyků. V každém z nich je možné řešit danou problematiku specifickými způsoby. Protože jsem cílil na jednoduchost, čitelnost, výkon a škálovatelnost, zvolil jsem jazyk Golang. Dle

průzkumů se také trend DevOps nástrojů napsaných v jazyce Golang, kterým je i Oasis, zvětšuje [20].

Při návrhu jazyka Golang se autoři inspirovali jazykem C++. Použití samotného C++ nebylo vhodné. Hlavním důvodem je příliš velká komplexita. U větších částí kódu se ztrácí přehlednost a často nelze jednoduchý problém vyřešit jednoduše a přehledně na pár řádcích. Golang tak přejal řadu vlastností jako například časté využití pointerů. Naproti tomu používá ale i garbage collector [20].

Protože se jedná o kompilovaný jazyk a nestačí jej pouze interpretovat, jako to je třeba u Pythonu. Je třeba řešit křížovou kompilaci a šíření sestavené aplikace. V současnosti je toto velice jednoduché. Typicky můžeme mít kód uložený v prostředí GitLab, kde také máme automatizaci, která nám kód kompiluje dle deklarovaných architektur a OS. Následně výsledné binární soubory vydá a nabízí je ke stažení. Stejně tak můžeme v automatizaci vytvořit kontejnerizovanou verzi. Tu pak šířit přes GitLab Registry. Pokud bychom používali Python, musíme řešit a hlídat verze jazyka Python na koncových stanicích, což může být do budoucna neudržitelné [20].

Build system jazyka a celkový toolchain je propracovaný a odpovídá požadavků moderního programovacího jazyka. Golang bere jako standard automatickou statickou analýzu kódu, automatické formátování kódu a další. Zároveň ale nabízí možnosti potlačení této automatické kontroly. Nicméně takový přístup se nedoporučuje. Pokud je například importovaný balíček, který se v kódu nepoužívá, je z kódu tento import automaticky odebrán. Lze ovšem explicitně říct, že se i tak má balíček importovat. To se ovšem neděje až u kompilace, ale už při psaní nezkompilovaného kódu ve Visual Studio Code. Obdobně na tom jsou i nepoužité proměnné. Řešeny jsou ale i věci jako odřádkování, mezery a podobně. To vše se automaticky kontroluje a je to již standardní součástí jazyka [20].

Při začátcích s jazykem Golang je možné využít online prostředí na adrese https://go.dev/play/ pro seznámení se s jazykem. Proto to, aby si uživatel vyzkoušel práci s jazykem Golang, tedy není nutné, aby si musel připravit lokální vývojové prostředí a řešit kompilaci [20].

Jazyk má bohatou standardní knihovnu. Na řadu úloh si tak vývojář vystačí právě s touto standardní knihovnou. Je také možné importovat volně dostupné balíčky. Protože je hodně nástrojů a aplikací pro cloud a DevOps napsáno právě v jazyce Golang (Docker, K8s aj.), existují balíčky právě pro práci s těmito technologiemi. To byl jeden z dalších benefitů, které jsem využil při tvorbě nástroje, na kterém pracuji sám a musí být hotový relativně rychle [20].

Tvůrci jazyka se chtěli vyhnout dynamickému typování, které je charakteristické například pro Python, Perl aj., a tak zvolili cestu statického typování. Díky tomu je možné při kompilaci zajistit vyšší bezpečnost ve smyslu použití a práce s datových typy [20].

Další výhodou je opět jednoduchost a přehlednost při práci s více vlákny a synchronizací takovýchto úloh. Java a C++ mají naopak syntaxi pro vytvoření nového vlákna mnohem košatější a kód se pak stává hůře čitelným [20].

Golang nepoužívá třídy, ale jejich alternativu, a tou jsou struktury. Objevují se kritické názory vůči principům a použití klasického objektově orientovaného programování. Jedním z argumentů je přílišné komplikování jednoduchých věcí. Tvůrci jazyka Golang tak zvolili použití struktur. To je pro tvorbu DevOps nástrojů ideální. Chceme totiž jednoduchost, čitelnost a počítáme s tím, že nebudeme pracovat s komplexními objekty, u kterých budeme detailně modelovat jejich chování [20]. V případě, že bychom chtěli detailněji modelovat chování, je doporučováno využívat principy Domain-driven design, kde chování definujeme na doménové vrstvě [27].

2.3 ARCHITEKTURA

Při úvodní analýze jsem si stanovil také požadavek na vhodnou architekturu nástroje. Nyní, když jsem stanovil programovací jazyk, mohu vybrat i adekvátní architekturu. Volba struktury je důležitá i z hlediska udržení přehlednosti celého řešení a může pomoct zajistit jednoduchost, zejména u juniorních a mediorních vývojářů. Pokud nový vývojář, který se s projektem setkává poprvé, zná danou architekturu, zrychlí se jeho zapojení do projektu a sníží se tak náklady na jeho zaškolení. Velice důležité je, aby se architektura implementovala s rozumem a maximálně odrážela a řešila poslání projektu. Je vhodné a někdy i žádoucí architektu přizpůsobovat a modifikovat dle potřeb [28].

Jazyk Golang nemá standard, který by doporučoval, ba dokonce vyžadoval specifickou architekturu. Dostupná IDE také nepracují s preferovanou architekturou. Bez zásahu vývojáře se používá architektura ploché struktury. Vývojáři mají tak svobodu ve výběru architektury a zároveň ji mohou implementovat dle svých potřeb. Vznikne tak struktura, která je jednak vhodná pro vývojáře samotné, ale také struktura, která je vhodná pro vyvíjený produkt a jeho obchodní cíle [27].

Plochu struktur lze chápat jako adresář, ve kterém jsou všechny zdrojové soubory. Není zde hierarchie, vzhledem k absenci organizace je tato architektura pro začátek vývoje a prototypování. Případně pokud vyvíjíme takzvané "drobečky", aplikace s velmi krátkým kódem obsaženým v jednom souboru. Tato struktura se využívá i pro malé knihovny. Jakmile projekt roste, je vhodné zvolit již architekturu, která se věnuje organizaci a zajistí logické oddělení tak, aby vše neovlivňovalo vše, ale jen to, co by mělo [28].

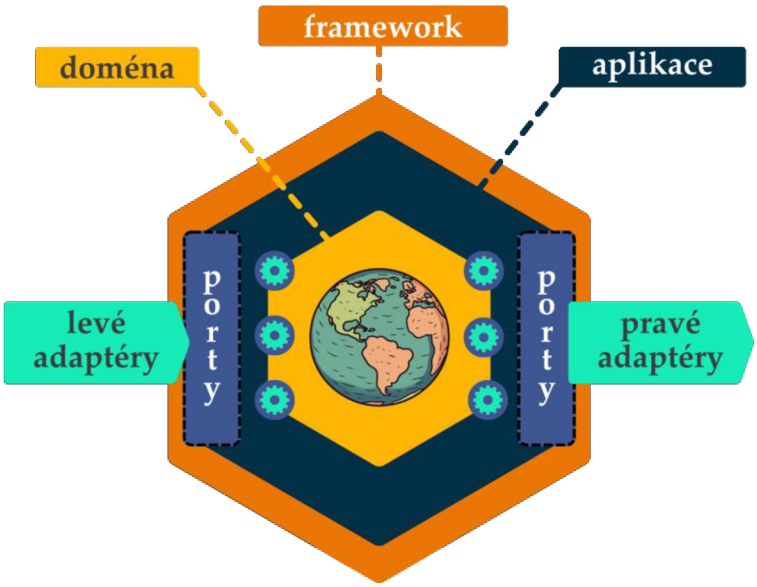
Tuto plochou architekturu můžeme upravit tak, abychom seskupili celky podle funkcí, které spolu logicky souvisí a tyto celky následně separujeme do balíčků struktur. Jako příklad lze uvést projekt, který by měl REST API. V takové situaci by bylo nutné separovat komponenty odpovědné za zpracování požadavků (Controller), logiku aplikace (Service), práci s daty (Repository) a definice datových modelů (Model). Jak si lze všimnout, získáme lepší přehlednost. Stále se ale nevyhneme potřebě hlídat si inicializace, práci se sdílenými proměnnými a polemi jejich působnosti. Proto je třeba striktně dodržovat doporučené postupy [28].

Dalším krokem je možnost seskupovat podle modulů. Tvoříme balíčky, které obsahují související funkce a vše, co je nutné k jejich provedení. Výhodou je snadná údržba a můžeme delegovat práci při vývoji a celkově tak proces vývoje urychlit. Zde se již začínají projevovat první nevýhody architektur. Toto řešení je již komplexnější, než předešlá. Musíme rozumět, jak v jazyce funguje separace, co to jsou moduly a jak je použít. Zároveň nás začínají svazovat pravidla [28].

Využít můžeme i nadstavbu předešlého seskupení a tím je seskupení podle kontextu. Tento přístup úzce souvisí s metodikou Domain Driven Design (DDD). Pokud tento přístup použijeme, získáme hlubší porozumění dané problematice, kterou má kód řešit (doména), zároveň vznikne kód, který toto reflektuje a je tak lépe organizovaný. DDD se totiž zaměřuje

na to, jak návrh systému mapuje reálný svět. Vznikají tak kontexty odpovídající konkrétním oblastem domény problému. Každý kontext obsahuje všechny související funkce a entity, které spolu úzce souvisejí a tvoří uzavřený celek. Tyto kontexty mohou být oddělené do různých modulů či balíčků v kódu, což umožňuje snadnou správu a údržbu systému. Domain Driven Design přináší mnoho výhod, jako je lepší organizace kódu, snazší údržba a testovatelnost a hlavně lepší porozumění doméně problému a potřebám uživatelů [27].

Tento přístup mi přišel vhodný pro tvorbu nástroje, jakým je Oasis. Jazyk Golang tento přístup nativně podporuje a konkrétní implementací je Hexagonální architektura, kterou jsem se rozhodl na základě rešerše použít. Jedná se o softwarový designový vzor, který se zaměřuje na oddělení doménové logiky od vnějších závislostí a infrastruktury. Jádro systému (doména) je obklopeno adaptéry, které slouží k připojení k vnějším systémům nebo ke komunikaci s uživateli. Doména / Jádro systému je tvořeno z doménové logiky a entit. Mohou zde být další typy objektů, jakými je například value object. Zde jsou implementována veškerá obchodní pravidla a operace. Nad touto vrstvou je vrstva služeb. Jedná se o vrstvu, kde jsou funkce, které operují nad doménovou logikou. Zároveň tyto funkce poskytují rozhraní, které slouží pro komunikaci s doménovou vrstvou vrstvám vyšším. Nejvyšší vrstvou Hexagonální architektury jsou adaptéry. Adaptéry poskytují komunikaci mezi aplikací a vnějším světem. Jsou zde vstupní adaptéry, které slouží pro interakci uživatele se systémem nebo externím systémem, který ovládá tuto aplikaci. Logicky zde jsou také adaptéry výstupní, které naopak slouží pro přístup k externím službám [27].



Obrázek 2.3: Hexagonální architektura

Díky Hexagonální architektuře jsem mohl oddělit konkrétní implementace od domény. Mohl jsem tak odděleně modelovat chování a jindy zase odděleně implementovat komunikační adaptéry. Změna chování jedné části pak neměla vliv na fungování jiné. Touto architekturou jsem získal také možnost snadného rozšíření adaptérů pro komunikaci s dalšími externími systémy. Nezávisle na doméně jsem mohl měnit uživatelské rozhraní. Také jsem mohl celý nástroj lehce testovat. Oasis je navržen jako rozšiřitelný nástroj, který navíc může rozšiřovat komunita. Z tohoto důvodu je výhodné využít právě Hexagonální architekturu. Nástroj, respektive jeho kód, tak bude moct být dlouhodobě udržitelný. Postup dokumentace bude intuitivnější. Vzhledem k trendu je třeba počítat s tím, že napojované služby mohou rychle stárnou a přestat být aktuální, naopak je možné, že přibude řada dalších nových technologií. Je tedy zásadní, aby architektura počítala se snadnou rozšiřitelností [27]. Podobně jako to například řeší nástroje Terraform a Ansible [21]. Navíc je třeba brát v potaz i fakt, že každá firma, každý domácí hobby nadšenec pracuje s jinými technologiemi. Mnou dodané základní napojení by nemuselo ve specifických případech někomu vyhovovat. I proto je tedy důležité, aby adaptér šlo snadno a rychle dopsat dle potřeby i někým jiným, než mnou.

2.4 ADAPTÉRY

Dále bylo třeba, abych navrhnul referenční sadu adaptérů. Při návrhu jsem se rozhodl pokrýt celý DevOps cyklus. Konkrétně fáze plánování, kódování, sestavení, testování, vydání, nasazení a dohled. Nutností byla také vhodná volba adaptéru pro práci s infrastrukturou. Při volbě adaptérů jsem vycházel ze zadání práce a požadavků cílové skupiny. Vždy jsem implementoval jednoho referenčního zástupce. Ostatní vývojáři tak mohou dle předlohy snadno doimplementovat svůj potřebný adaptér.

Pro fázi plánování jsem zvolil technologii Jira. Tuto volbu jsem podmínil širokým rozšířením technologie Jira. Také během rozhovorů byl zájem právě o tuto technologii. Konkrétně by měl adaptér založit nový projekt tak, aby jej mohl uživatel plnit úkoly a také by měl adaptér provést úvodní konfiguraci [29].

Fázi kódování jsem se rozhodl pokrýt technologiemi GitLab a Dev Container. Kódy vývojáře se umístí do repozitáře uvnitř GitLab. Vývojové lokální prostředí vývojáře jsem se rozhodl definovat pomocí Dev Container. Vývojář tak bude moci vyvíjet uvnitř kontejneru, který obsahuje všechny potřebné instalace a konfigurace. Výhodou GitLab je i jeho projektové Registry. Jedná se o úložiště pro Docker obrazy (image). Definice vývojových prostředí jsem zvolil na základě rostoucí obliby a trendu. GitLab jsem zvolil jako referenční adaptér z důvodu, že dokáže pokrýt velkou část DevOps cyklu a lze tak uživateli poskytnout jednotný náhled na použití adaptéru této technologie [16].

Z pohledu automatizace sestavení jsem se rozhodl využít GitLab, dle předchozí fáze. Z pohledu sestavení jako takového jsem zvolil BitBake. BitBake je nativní systém sestavení pro projekty vyvíjené pod projektem Yocto Project [6]. Technologický adaptér jsem tedy volil tak, aby bylo možné splnit cíl této práce.

U testování jsem se opět pro automatizaci rozhodl využil GitLab. Pro testy jako testy kódu, hlídání CVE a další jsem zvolil realizaci skrze nástroje Yocto Project.

Nasazení jsem se rozhodl realizovat standardní cestou, a to skrze tag. Uživatel tak vydá tag a spustí se mu automatizace, kdy se na jejím konci spustí fáze pro nasazení. Zde jsem zvolil více adaptérů, a to podle oblasti. Jako poskytovatele infrastruktury jsem zvolil napojení na službu Hetzner. Jedná se o ekonomicky výhodné cloud protření. Je tedy ideální

pro demonstrační účely. Navíc mi bylo doporučeno během rozhovorů a prezentací. Pro komunikaci se stroji jsem zvolil dle přístupu nástroje Ansible protokol SSH [19]. U nasazení jsem se rozhodl implantovat staging nasazení pomocí QEMU. Opět se jedná o přímočaré a intuitivní předvedení [1].

Fázi dohledu jsem se rozhodl implementovat pomocí sady nástrojů Prometheus, Grafana a Node Exporter. Jedná se o rozšířené a vyspělé nástroje, které dokonale poslouží vytyčeným cílům práce [30].

2.5 FUNKCIONALITY

Před implementací nástroje jsem se rozhodl, mimo architektury, také navrhnou i jeho funkcionality. Díky této separaci jsem se vždy mohl více soustředit na danou činnost. Díky přehlednému a jasnému návrhu funkcionalit jsem mohl rychle a jasně provést implementaci. Hlavním úkolem nástroje je zjistit, co uživatel požaduje, a následně toho systematicky docílit. Tento úkol se však rozpadá na dílčí úkoly.

Prvním dílčím úkolem Oasis je přijetí deklarace od uživatele. Tato deklarace obsahuje silně deklarativně popsaný požadovaný stav. Deklarace bude realizována pomocí souborů, kterých může být více. Dalším dílčím úkolem Oasis je přijetí hlavního definičního souboru skrze konzolové rozhraní. Následně dle umístění na disku tohoto souboru musí Oasis projít všechny další soubor v umístění, které také obsahují definice.

Obrázek 2.4: Chronologicky seřazené funkce nástroje Oasis

Z načtených deklaračních souborů musí Oasis vytěžit informace. Tyto informace musí uložit do vhodných struktur po pozdější využití. Nadbytečné informace musí nástroj igno-

rovat. Informace, které chybí, musí rozlišit na podstatné a volitelné. U volitelných musí nástroje pokračovat ve své činnosti s výchozí hodnotou dané položky. Pokud chybí podstatná položka, je třeba informovat uživatele a ukončit činnost.

Současně musí vznikat také vazby mezi jednotlivými součástmi, které si nástroj automaticky vyvodí. V paměti si tak nástroj musí tvořit strom závislostí.

Dalším krokem je příprava fronty úkolů. Nejprve tedy nástroj určí dle deklarace dílčí kroky, které je nutno splnit. Tyto kroky průběžně řadí do fortny a jejich pořadí zohledňuje pomocí stromu závislostí z předchozího kroku. Fronta musí také umožnit paralelní zpracovávání a při implementaci se musí počítat s rozvětvením a paralelizací.

Fronta je následně předána do vykonávací části nástroje. Nástroj dle fronty začne vykonávat sekvenčně jednotlivé kroky. Pokud zjistí, že se fronta rozvětvuje, začne nástroj paralelizovat svou činnost a orchestrovat spouštění a vykonávání dalších úkolů.

2.6 ALGORITMY

Při návrhu nástroje jsem se rozhodl pro použití existujících algoritmů. Hlavním úkolem nástroje je vygenerování posloupnosti úkolů. Tyto úkoly mají zohledněné pořadí návaznosti a paralelní vykonávání. Algoritmus bych mohl zkusit vmyslet sám, ale řešení problematiky již existuje a je optimální. Dále je úkolem nástroje provést výše zmíněné kroky. Vykonávání se musí také orchestrovat. Zde jsem se inspiroval řešeními, jako jsou balíčkovací systémy či BitBake. V kapitolách 2.6.1 a 2.6.2 se nachází podrobnější vysvětlení daných algoritmů. Některé algoritmy jsem si pro své účely částečně upravil. Některé jsem částečně doplnil o mnou potřebné náležitosti, jiné jsem zase očistil o nepotřebné větve.

2.6.1 GRAFOVÉ ALGORITMY

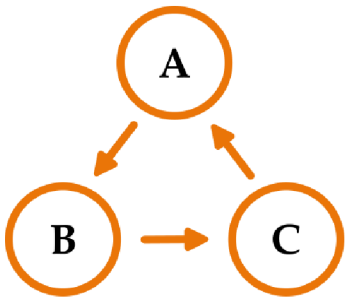
Nástroj Oasis obsahuje struktury reprezentující závislosti. Konkrétním příkladem je zavilost repositáře na verzovacím systému. Oasis také musí vykonat sled událostí, kdy určitý úkol vyžaduje splnění předchozích. Mohli bychom výše uvedenou modelovou situaci převést na činnosti: Vytvoření verzovacího systému > konfigurace verzovacího systému > vytvoření

repositáře ve veršovacím systému. Jak je vidět, pokud bychom vynechali úkol vytvoření verzovacího systému, nemohli bychom vytvořit repositář. Verzovací systém totiž neexistuje.

Nevymýšlel jsem nové algoritmy, ale vybral jsem již existující algoritmy a struktury. Na tento typ úkolů se nejlépe hodí topologické řazení využívající grafové algoritmy [31]. Správce balíčku APT pracuje s balíky, kdy každý balík má soubor, ve kterém jsou definovány přímé zavilostí. Při řešení závislostí balíku se postupuje rekurzivně. Zjistí se nejprve přímé závislosti tohoto balíku. Následně se to samé děje pro tyto přímo závislé balíky. Následně se provádí vzestupné řazení balíků podle počtu závislostí. Posledním krokem je instalace seřazených balíků, kdy se tedy nejprve instalují balíky bez přímých závislostí. Jako poslední se instaluje požadovaný balík. Obdobně funguje i DNF, který je určený pro balíky RPM. Tím vším jsou balíčkovací nástroje silně spjaté s OS. U problematiky vývoje aplikací se také vyskytuje podobný princip. Například správa zavilostí balíčku pro programovací jazyk Golang, softwarový systém Node.js a Python. Nástroj Helm, správce balíků pro Kubernetes, také řeší tuto problematiku. Opět velice podobným způsobme. Jak lze vidět, i když se zvyšuje úroveň abstrakce a posouváme se na vyšší vrstvy nad OS, jádro problému a princip jeho řešení zůstává stejný [32].

V teorii grafů bychom výše zmíněné balíky mohli přivést na uzly grafu. Závislosti lze reprezentovat jako hrany mezi dvěma uzly, respektive balíky. Nutnou podmínkou je také, aby graf, který nám vzniká, byl orientovaný. Nestačí nám totiž pouze informace, že dva balíky mají mezi sebou vazbu. Musíme vědět, který balík je ten závislý. Musíme ale ještě zajistit absenci zacyklení vazeb. Tedy aby nenastala například situace, kdy máme tři balíky A, B a C. Balík A by byl závislý na balíku B, B by byl závislý na C a C by byl závislý na A. Tím by nám vznikl cyklus. Graf, který má takovou vlastnost, se nazývá acyklický. Rozhodl jsem se tedy při pozdější implementaci nástroje využít orientovaný acyklický graf [31].

Obrázek 2.5: Orientovaný acyklický graf



Obrázek 2.6: Orientovaný graf s cykly

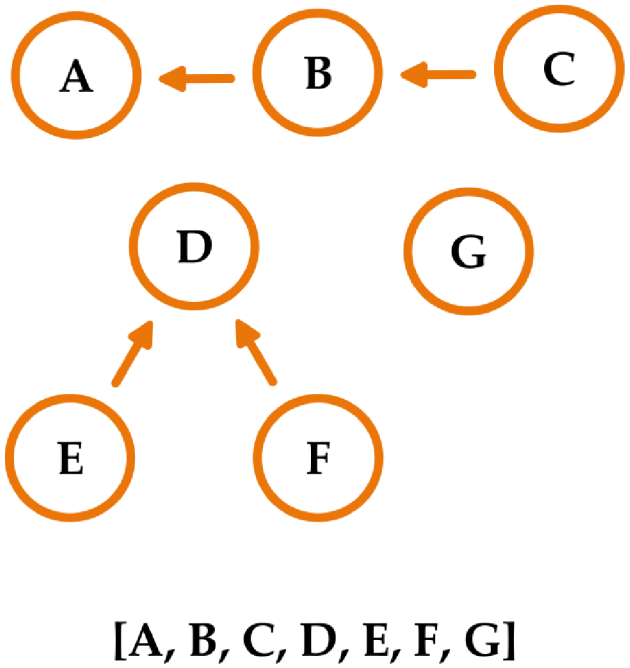
2.6.2 TOPOLOGICKÉ ŘAZENÍ

V kapitole 2.6.1 jsem popsal možnost využití orientovaného acyklického grafu. Graf nám říká, jaké jsou uzly a jaké jsou jejich vazby. Je to matematická struktura. Využijeme-li modelovou situaci s balíky tak, abychom správně určili pořadí instalace jednotlivých balíků, musíme provést topologické řazení.

Algoritmus topologického řazení seřadí uzly grafu následujícím způsobem: Nejprve se spočítají příchozí hrany pro každý vrchol. Tedy na kolika balících je daný balík závislý. Poté se postupně začnou procházet jednotlivé vrcholy. U sousedů daného vrcholu se následně sníží počet závislostí o 1. Pokud je počet závislostí rovný nule, pak tento sousední vrchol přidá do fronty. Takto se postupně tvoří a prochází fronta. Výsledné seřazení není unikátní. Je pouze korektní. Složitost takového algoritmu je rovna O (V+E), kdy V je počet uzlů a E je počet hran. Pokud se algoritmus upraví, je možné jím procházet tak, že výsledné seřazení bude obsahovat více vrcholů na určité pozici, to lze využít například pro paralelní instalaci dvou balíků či vykonávání dvou úloh [31].

Obrázek 2.7 obsahuje vrcholy A až G. Jsou zde orientované vazby. Například vazba z B na A. Dle algoritmu tedy umístíme do seřazené řady vrcholů takové vrcholy, které nemají odchozí hrany. To jsou A, D, a G. Vrcholy z grafu odebereme a postupujeme takto dále. Složitost tohoto konkrétního případu by byla O (11). Jako výsledek jsme na obrázku dostali řadu A, D, G, B, E, F, C. Správně by ale bylo i D, G, A, B, F, E, C. Pokud bychom použili modifikovanou verzi algoritmu, tedy tu, která je vhodná pro paralelní vykonávání, výsledné seřa-

zení by mohlo vypadat takto: [{A, D, G}, {F, E, F}, {C}]. Dostáváme tři množiny, tedy na pořadí vykonání úkolů / instalaci balíků dané množiny nezáleží, ale na pořadí množin záleží. To reflektuje naše požadavky [32].



Obrázek 2.7: Demonstrace topologického řazení

2.7 FORMÁTY SOUBORŮ

Uživatelem deklarovaná infrastruktura je nástrojem Oasis uložena do souboru. Případně uživatel sám může takový soubor vytvořit. Také je možné, aby definice byla rozdělena do více souborů. Musel jsem tedy vhodně zvolit formát takového souboru. Soubor představuje popis infrastruktury, který je možno manuálně upravovat, verzovat ve verzovacím systému, opakovaně spouštět, číst či zpracovávat pomocí AI.

Při volbě jsem zohlednit snadné strojové zpracování. Tedy abych mohl snadno pracovat s daným souborem na úrovni aplikace a v jazyce Golang. Současně jsem ale také hledal formát, který je lidsky čitelný, a to co nejvíce jednoduše a přehledně. Nejen že tak bude možné kontrolovat obsah souboru pouhým okem, ale navíc čtenář dostane jasně definovanou a průhlednou strukturu. Kombinace předchozích vlastností vede mimo jiné i ke snadné do-

kumentaci daného formátu souboru pro koncové uživatele. Při zvažování formátu souboru jsem také vzal v potaz i rozšířenost formátu. Tedy aby co možná největší počet uživatelů již někdy takový formát viděl a dokázal jej číst. Formát musí také podporovat minimálně základní datové typy. Vzhledem k použití je také vhodné, aby jednotlivé soubory mohly obsahovat komentáře, díky kterým by v praxi bylo snadnější a rychlejší se zorientovat v obsahu souboru.

Jak ukazuje Tabulka 2.3, formát YAML splňuje všechny předpoklady, a proto jsem se jej rozhodl využít. Připomeňme, že bereme v potaz užití formátu pro definici požadavků uživatele na infrastrukturu. Mimo jiné se YAML obecně používá na podobné problematiky. Můžeme jej vidět v manifestech pro Kubernetes, v definicích CI/CD pipeline v prostředí GitLab, či ve tvorbě automatizačních procesů v Ansible Playbook [19].

Alternativou by mohlo být vytvoření vlastního formátu souboru. Rozhodl jsem se ale pro využití již existujícího řešení, které splňuje všechny požadavky. Navíc jazyk YAML je standardizovaný a globálně používaný. Můj vlastní formát by toto nesplňoval [16].

Formát TOML, podobný formátu YAML, se jeví také jako vhodný kandidát. Achillovou patou je zde jeho přílišná jednoduchost a přehlednost. Pokud bych se rozhodl jej využít, pak by paradoxně jeho největší přednosti nejvíce zkomplikovaly jednoduchou a přehlednou lidskou čitelnost. Začaly by totiž oproti formátu YAML delší popisy bez struktur a vhodných datových typů.

Tabulka 2.3: Porovnání formátů souborů

	YAML	JSON	TOML	Binární soubor	CSV	XML	Vlastní formát
Strojově čitelný	✓	✓	✓	✓	✓	✓	✓
Lidsky čitelný	✓	●	✓	✗	●	●	✓
Struktury	✓	✓	●	✗	✗	✓	✓
Vazby	✓	✓	●	✗	✗	✓	✓
Intuitivní	✓	✓	✓	✗	✓	●	✓
Jednoduchost	✓	●	✓	✗	✓	●	✓
Přehlednost	✓	●	●	✗	●	✗	✓
Rozšířenost	✓	✓	●	✗	✓	✓	✗
Komentáře	✓	✗	✓	✗	✗	✓	✓

JSON postrádá nativní možnost komentářů. Jeho struktura je sice lidsky čitelná, ale reprezentace dat ve formátu YAML je přehlednější a méně komplikovaná. Podobně na tom je i formát XML. Formát je sice čitelný lidským okem, ale zápisy se stávají nepřehlednými a v řadě případů i málo intuitivními.

Formát CSV je určen zejména na reprezentaci tabulkových dat. Z toho plyne, že na mou problematiku by sice byl aplikovatelný, ale vzniklo by tak již mnoho komplikací. Zásadním nedostatkem je nemožnost tvorby struktur a vazeb. Mezi nedostatky se dále řádí absence tvorby komentářů. V tomto případě je tedy CSV jasnou ukázkou situace, kdy každý formát má své pole působnosti a kde řeší skvěle danou problematiku. Každý formát byl navržen na řešení určité problematiky. Je tedy vhodné použít formát na problémy, ke kterým byl předurčen.

Nejméně vhodnou reprezentací je binární soubor. Ten splňuje pouze požadavky pro strojovou čitelnost. Ale i tato strojová čitelnost by znamenala pro vývojáře nutnost psaní podrobné dokumentace binárního souboru a implementaci čtení a zápisu do souboru dle takové dokumentace. Pro splnění ostatních požadavků by byly nutné podpůrné prostředky. To by znamenalo další nežádoucí komplikace při tvorby a použití nástroje Oasis.

YAML je jazyk, který byl navržena na reprezentaci konfigurace. Píší se v něm konfigurační soubory a reprezentace infrastruktury. Oficiální specifikace jazyka je dostupná online na odkaze https://yaml.org/. Čtenář se tak dozví veškeré podrobnosti. Základní vlastností je reprezentace dat pomocí seznamu. Také se data a seznamy řídí hierarchií. Tyto vlastnosti jsou implementovány pomocí odsazení a dvojice klíč a hodnota [16].

Následující ukázka formátu YAML reprezentuje seznam kačenek různých typů. Soubor začíná trojicí symbolů -. Následuje krátký popisek ve formě komentáře. Dále pak klíč reprezentující skupinu kačenek, který v sobě obsahuje vnořené klíče reprezentující kačenky 1 až 3. Alternativě lze toto zapsat pomocí listu. Vlastnosti každé kačenky jsou pak definovány jako klíče a řetězcové hodnoty. Případně jako další vnořené klíče.

```yaml
---
# Toto je demonstrační obsah souboru, který reprezentuje seznam ka-
čenek
ducks:
  duck1:
    name: Jack
    material: rubber
  duck2:
    name: Ian
    material: rubber
    spec:
      style: gandalf
  duck3:
    name: Andy
    material: rubber
    spec:
      style: gollum
```

Zdrojový kód 2.1: Struktura YAML souboru

2.8 LICENCE

Zadání práce jednoznačně určuje fakt, že má být nástroj open source. Tedy, že se má jednat o nástroj s otevřeným zdrojovým kódem, nikoliv o nástroj proprietární. Aby mohl být nástroj volně šířen a používán, bylo třeba, abych vybral vhodný typ licence. Licence je právní dokument a upravuje práva udělená autorským zákonem. Nástroj tak může být v určitých ohledech méně či více omezen oproti nepoužití licence. Open source obecně rozvolňuje autorské právo a dává uživateli více volnosti. Na druhé straně licence pro proprietární software naopak častěji omezuje. Existují samozřejmě i výjimky [33].

Komplexitu vymýšlení licence jsem se rozhodl odstínit využitím již existující SW licence. GPL licence, kterou je licencováno linuxové jádro, umožňuje redistribuovat zrodový kód, binární soubory a software. To je příklad rozvolněné autorského práva. V současné době jsou relevantní verze č. 2 a č 3. Většina GPL je právě ve verzi 3. GPL ve verzi 2 umožňuje používat linuxové jádro k libovolným účelů. Umožňuje také číst zdrojový kód a editovat jej. Stejná svoboda pak platí i pro šíření jádra jako takového, ale i jeho úprav. Tedy že, odvozená díla musí být uvolněna také pod GPL. Tato vlastnost předchází situaci, kdy by si vývojář chtěl

přivlastnit nějaký open source software. Zde se začíná objevovat pro někoho první nevýhoda. Licence nařizuje, že pokud se provede úprava linuxového jádra, respektive software licencovaného GPL, pak tato úprava musí být zveřejněna a musí být také licencována jako GPL. Něco jiného je pak ale situace, kdy máme distribuci OS Linux. Jedná se totiž o kolekci aplikací, kdy každá aplikace má svou licenci a ty se vztahují vždy a jen pouze k dané aplikaci, nikoliv k celé distribuci. GPL ve verzi 3 navíc eliminuje slabosti předchozích verzí v otázce hardwarového omezení. LGPL je rozvolněnější variantou GPL, kdy program může pracovat s knihovnou s licencí GPL a takový program může být vydán i pod jinou licencí včetně té komerční [33] [34].

Licence BSD byla vytvořena pro operační systém BSD. Oproti GPL umožňuje modifikace distribuovat i pod jinými licencemi. Licence Apache umožňuje šíření pod stejnou, či jinou licencí, ale za podmínky předání textového souboru s názvem dodání původního souboru včetně údajů původních autorů a jejich kontaktních informací. Známým zástupcem je projekt Apache HTTP Server [34].

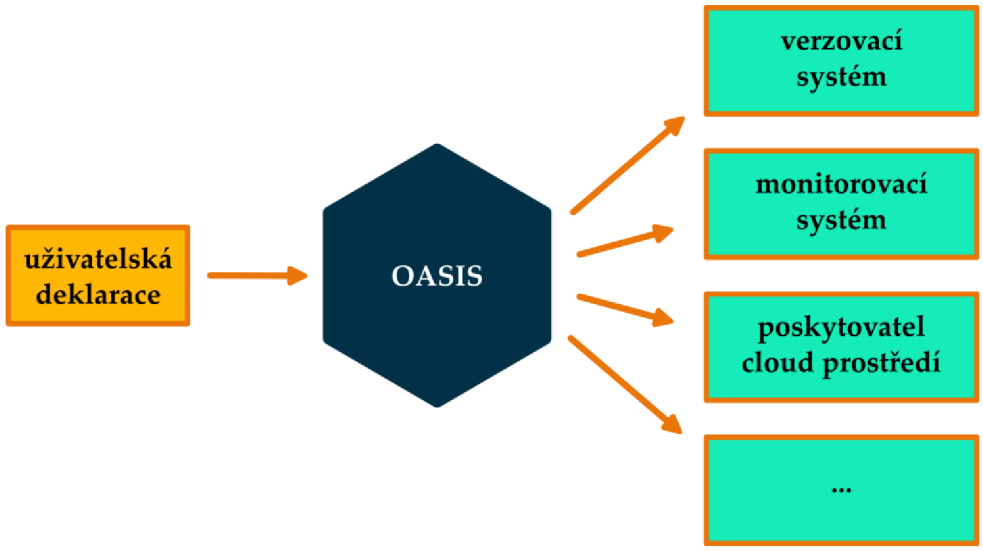
Na Massachusetts Institute of Technology vznikla licence MIT. Tato licence je krátká a velmi volná. Podmínkou je nutnost dodat text licence MIT spolu s aplikací a jménem autora. Takto licencované je například sjednocené grafické rozhraní X11 [34].

Svou práci jsem se rozhodl licencovat pomocí MIT. Tato licence nepřináší komplikace ani pro mě a mé budoucí využití, ani pro využití a případné rozšíření ostatními vývojáři a uživateli. Je jednouchá, jasná a průhledná. Smyslem mé práce je poskytnou nástroj, který bude užitečný a jsou vítány jakákoliv využití, rozšíření, převzetí a šíření. Do repositáře projektu jsem vložil soubor LICENSE, který obsahuje definici MIT licence. Jedná se o textový soubor, kde je řádek s názvem licence, řádek, kde je mé jméno jako autora projektu a následně text licence MIT. GitLab si automaticky tuto licenci vyčte a návštěvníky o této licenci informuje.

Díky vhodné volbě licence, tedy MIT, jsem umožnil šíření a využití nástroje Oasis tak, aby nástroj nebyl svázán autorskými právy a byl využitelný.

3 IMPLEMENTACE

V této kapitole popisuji postup implementace. Plynule navazuji na kapitolu č. 2, kde jsem nástroj Oasis navrhoval. Dle návrhu mým úkolem bylo implementovat nástroj Oasis. Uživatel si tento nástroj musí stáhnou jako binární soubor, který mu stačí pouze spustit a nemusí instalovat další balíčky a knihovny. Tento nástroj bude uživatel používat jako konzolovou aplikaci. Nástroj bude od uživatele přebírat deklaraci toho, co chce vyvíjet a jakými technologiemi disponuje. Deklarace budou v podobě souborů YAML. Oasis si tuto deklaraci následně musí přebrat a interně převést na strom celkové infrastruktury. Tento strom následně převádí na strom úkolů, které je třeba vykonat. Při vykonávání nástroj musí zohledni nejen pořadí, ale i možnosti paralelizace. K tomuto úkolu musí nástroj používat topologické řazení. Nástroj musí být implementován v jazyce Golang. V návrhu jsem také určil, že kód bude strukturován do Hexagonální architektury. Nástroj také musí implementovat technologie ve formě adaptérů dle kapitoly č. 2.4.



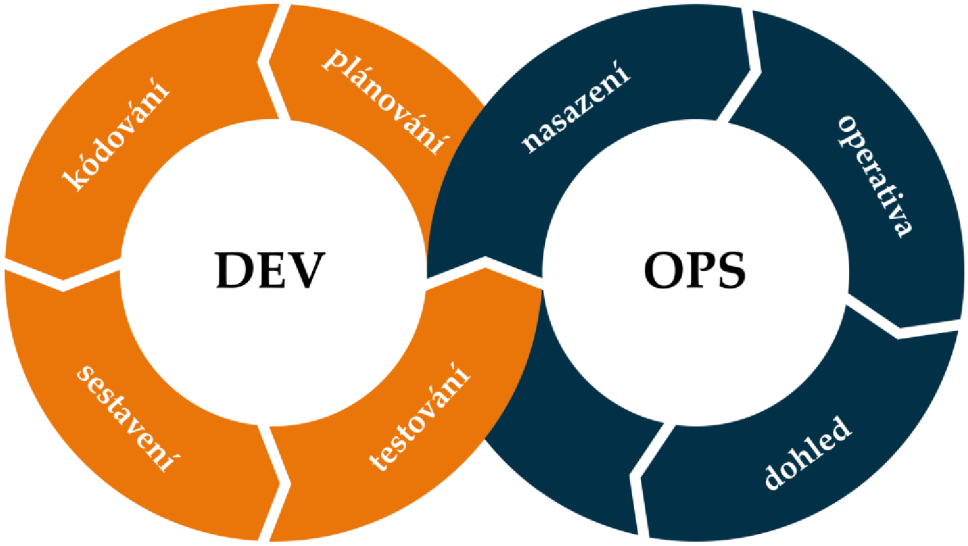
Obrázek 3.1: Obecný přehled nástroje Oasis

V následnících kapitolách nejprve popisuji implementaci DevOps metodiky pro vývoj a další podpůrné prostředky. Čtenář tak dostane možnost pochopit výhody implementace DevOps nejen z pohledu využití Oasis, ale také z pohledu vývoje takového nástroje. Dále

pak popisuji implementaci nástroje Oasis, kdy se vždy zaměřuji na dílčí část tak, jak chronologicky probíhá přijetí a vykonání deklarace.

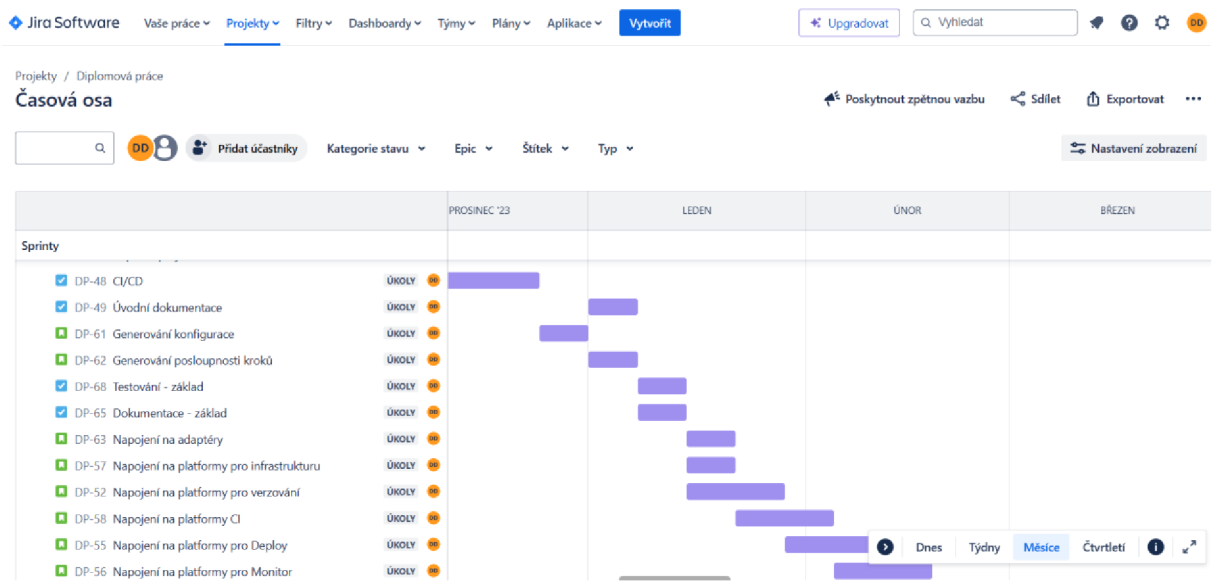
3.1 PODPŮRNÉ PROSTŘEDKY

Při vývoji SW jsem se rozhodl využít DevOps přístup. Mé potřeby nezahrnovaly všechny kroky cyklu DevOps. Oasis je nástroj, který si uživatel stáhne a používá pro své potřeby. Mou potřebou nebylo v tuto chvíli získávat analytická data. Proto jsem neimplementovat právě dva poslední kroky cyklu, kterými jsou operativa a monitoring, které se zejména starají o běh a kvalitu běhu nástroje.



Obrázek 3.2: DevOps cyklus

Rozhodl jsem se využít krok plánování. Tento krok dává sice největší smysl při práci v týmu, ale přináší benefity i jednotlivci. Plán jsem si vypracoval v praxi velmi oblíbenými nástroji Jira [29]. Systematicky jsem mohl pracovat dle takto připraveného plánu. Osobně mi dělá problém věnování příliš velkého úsilí dílčím úkolům na úkor celkového výsledku. Díky plánování jsem věděl, kolik času jakým činnostem věnovat. Při plánování jsem si také utřídil myšlenky a získal přehled nad celkovou cestou implementace. Je však důležité, že tento přístup nemusí vyhovovat každému [29].



Obrázek 3.3: Část plánu v podobě časové osy Jira

Dalším krokem, pro mě nutným, bylo kódování. Krok je zejména o procesu psaní kódu nástroje, který jsem realizoval skrze editor Visual Studio Code a DevOps platformu GitLab. V kapitole 3.1.1 detailněji rozebírám využití platformy GitLab. Pro účely testování jsem si také vytvořil vývojový kontejner, který obsahoval veškeré nutné nástroje, balíčky, důležitá nastavení a systémové proměnné. Stačilo tedy, abych pouze naklonoval kód, který jsem otevřel ve Visual Studio Code (lze i jiný editor) a uvnitř editoru jsem spustil vývojový kontejner. To mi také zaručilo čisté pracovní prostředí. Nutnou podmínkou je však v systému přítomná instalace nástroje Docker.

Implementoval jsem také kroky sestavení, testování a vydání. V kapitole 3.1.2 popisuji detailně tuto kontinuální integraci. Hlavní motivací byla automatizace kroků pro začlenění nové změny a zajištění její funkčnosti a kvality, a to ve smyslu multiplatformním.

Posledním krokem pospaným v kapitole 3.1.4 bylo nasazení. Využil jsem minimalistický přístup vystavení binárních souborů a kontejnerů volně ke stažení. S daným krokem však souvisí i podpora koncových uživatelů. Proto jsem průběžně psal také dokumentaci pro koncové uživatele.

Podstatným benefitem metodiky DevOps je i lepší organizace týmové spolupráce při vývoji a operativě. Protože jsem nástroj implementoval jako jednotlivec, tento benefit

jsem nevyužil. Pokud by se však tento open source produkt rozšířil a vývojáři by jej postupně doplňovali o jimi potřebné přírůstky, začal by se tento benefit pozitivně projevovat.

3.1.1 VERZOVACÍ SYSTÉM

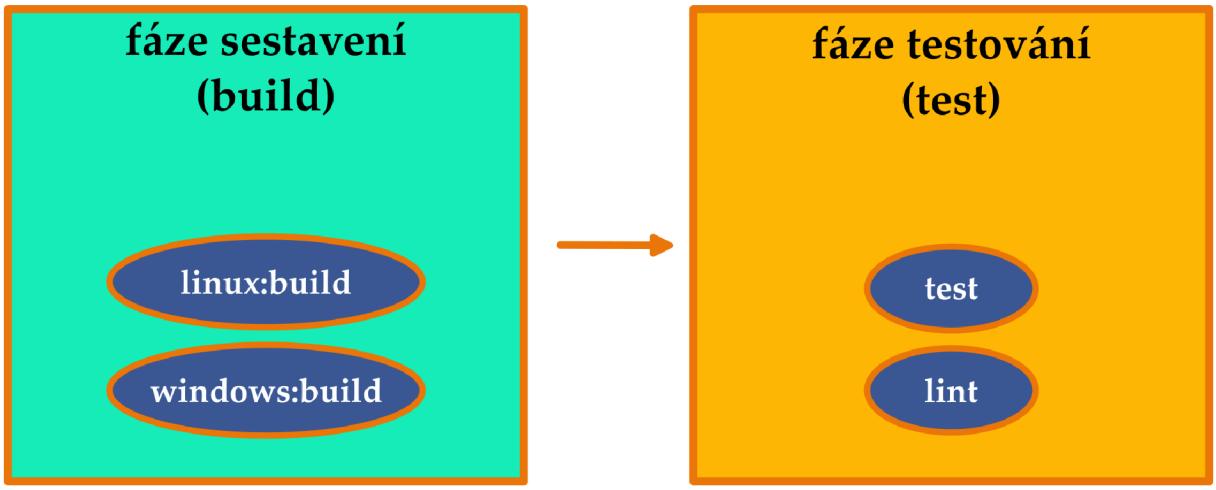
Při tvorbě nástroje jsem se rozhodl využít vhodný verzovací systém. Konkrétně jsem zvolit platformu GitLab. Tuto volbu jsem podmínil osobní preferencí. V praxi jako DevOps engineer jsem se s platformou GitLab setkal nejčastěji. Navíc jsem s platformou obeznámen nejvíce do hloubky. Platformu znám nejlépe oproti jiným. Mimo to také nabízí více zabudovaných nástrojů než alternativní GitHub.

V platformě GitLab jsem vytvořil veřejný repozitář pro nástroj Oasis s adresou https://gitlab.com/david.dlouhy1/oasis. V repozitáři se nachází kompletní kód včetně jeho verzování. GitLab také zobrazuje základní dokumentaci, kterou jsem sepsal v jazyce Markdown. To umožní příchozímu uživateli rychle se seznámit s tímto projektem. V platformě GitLab jsem vytvořil i další separátní repozitáře. Například repozitář s mou vlastní Yocto Project vrstvou, šablonové repozitáře a další. Odkazy na ně jsou dále v textu.

Git je pouze dílek skládači platformy GitLab. Rozhodl jsem se využít i další dílky, respektive funkce platformy. Díky tomu jsem se mohl následně více soustředit na samotný vývoj a neřešil jsem například manuální sestavování a nasazování aplikace. Platforma mi také umožnila lépe pracovat s odhalenými problémy při vývoji. Tyto problémy jsem mohl evidovat ve formě Issues a později se k nim vrátit.

3.1.2 CI

Abych mohl pracovat rychleji a kontinuálně se mi validovaly kousky kódu, rozhodl jsem se použít CI. Konkrétně jsem CI implementoval přímo v nástroji GitLab (GitLab CI). Tato automatizace pokrývá sestavení produktu, jeho otestování a také statickou analýzu. Všechny tyto části jsem definoval v souboru `.gitlab-ci.yml`. V úvodu souboru nastavuji důležité proměnné, které jsou později použity. Následuji definováním jednotlivých fází a poté už konkrétně specifikuji jednotlivé automatizace. Každá část automatizace má také definovaný kontejner, ve kterém se daná automatizace provádí.



Obrázek 3.4: Schéma fází procesu CI

Automatizace `linux:build` sestaví Oasis jako aplikaci určenou pro OS Linux s architekturou amd64. Používám nativní nástroj sestavení jazyka Golang, kdy jej konfiguruji pro své potřeby proměnným prostředím. Výstup je sdílen do ostatních částí automatizace pomocí artefaktů. Velmi podobně jsem řešil i sestavení pro platformu Windows. Zde jsem musel ale upravit konfiguraci a použít jiné proměnné. Zároveň jsem musel doinstalovat překladače pro křížovou kompilaci. Mohl jsem kompilovat ve Windows kontejnerech, ale linuxové kontejnery jsou univerzálnější a jedná se o doporučený postup [16].

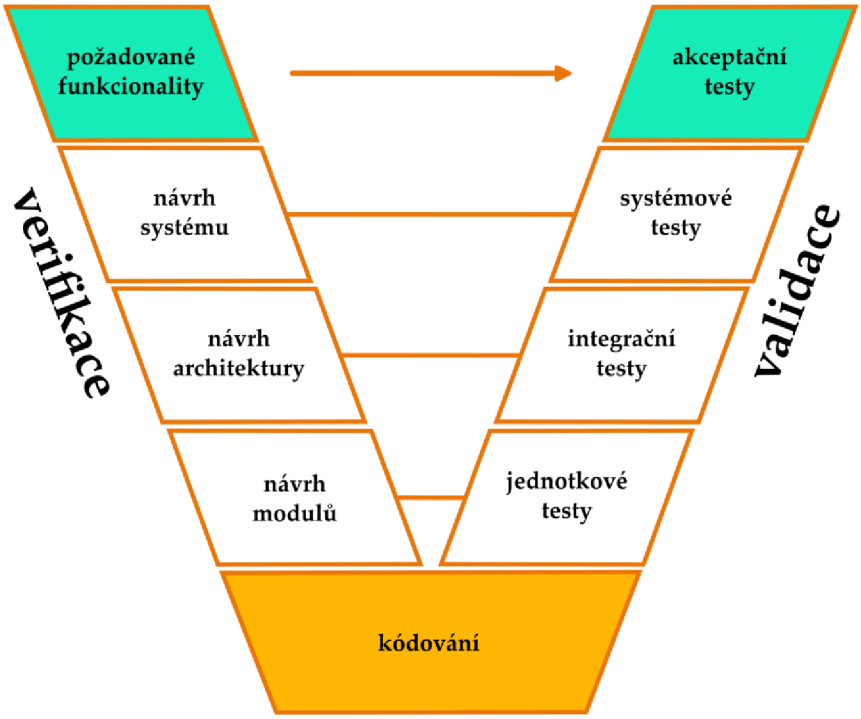
Jako další krok fáze CI jsem implementoval testování. Fáze se skládá ze dvou částí. Část `test` spustí všechny tesy projektu pomocí zabudovaného Golang testovacího nástroje. Druhá část `lint` provede statickou analýzu kódu. K tomuto používám nástroj golangcli-run, který v automatizaci do kontejneru instaluji. Nastavení nástroje držím v souboru `.golangci.yml` a jeho obsah jsem vytvořil pomocí doporučeného postupu [27]. Tuto kontrolu jsem implementoval jako nepovinnou. Tedy pokud by došlo k registraci problémů, bude pouze signalizováno, ale celý průběh CI/CD nebude zastaven. Ačkoliv linter může odhalit nedostatky, v případě Oasis tyto nedostatky nebudou kritické pro běh aplikace.

Díky CI jsem rychleji přišel na případné problémy, které se nacházely v nových přírůstcích kódu. Tyto problémy jsem vyřešil a výsledné vydání verze bylo bez těchto odhalených problémů. Automatizace mi tedy pomohla soustředit se na cíle diplomové práce. Pro-

tože jsem vytvořil nástroj s otevřeným zdrojovým kódem a kdokoliv může vytvářet nové přírůstky, tak i tomuto přispěvateli se provede CI fáze a nemusí ji sám konfigurovat.

3.1.3 TESTY

Abych zajistil kontrolu funkčnosti nástroje, rozhodl jsem se průběžně psát testy. Realizoval jsem zejména jednotkové testy a v menším množství také testy akceptační. Při výběru testů jsem vycházel z V modelu, který přiřazuje dle abstrakce jednotlivé typy testů. Při testování jsem vybral ty testy, které pro mě měly nevyšší míru přínosnosti. Ostatní bych doporučil implementovat ve chvíli, kdy je na daný úkol podstatně více času a v komerční sféře také, kde firma ovlivní vyšší rozpočet. Nejde pouze o to testy bezhlavě vytvořit. Testy by měly vývojáři poskytnout co nejvíce užitku a vývojář by měl zvážit, zda vynaložené úsilí a prostředky přinesou adekvátní benefity. Akceptační testy jsem použil na celkové otestování. Tedy zda byla splněna konkrétní funkcionalita. Jednotkové (unit) testy jsem používal na otestování funkcí v kódu. Kód jsem se rozhodl nepokrýt 100 %. Vytipoval jsem si kritická místa, jakými jsou například operace nad stromy zavilostí a úkolů. Testoval jsem také správné detekce faktů z deklarace a jejich správné vyhodnocení [3].

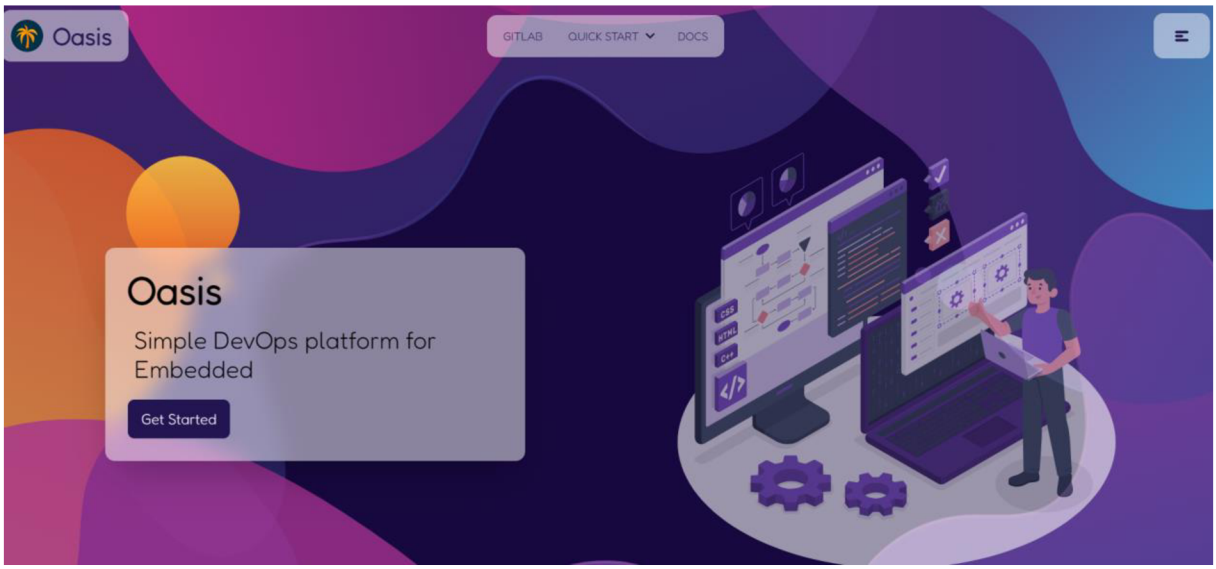


Obrázek 3.5: V model

Golang je jazyk, který nativně podporuje psaní testů, a proto má také zabudované funkcionality pro jejich vytváření. Nechal jsem si tedy nástrojem vždy vygenerovat automaticky tyto testy. Generátor pohlídá všechny náležitosti jako jsou názvy testovacích funkcí, jejich propojení z funkce, obsah testu a název souboru s testy. Následně jsem musel ověřit, zda test skutečně testuje správnou funkcionalitu. Generátor testů nepozná chybu v logice. Poté jsem připravil testovací sadu dat. Sada dat byla vždy ve formátu tabulky, kde vstupním datům náleží očekávaný výstup. Tento přístup, nativní pro Golang, se nazývá Table driven testing. Takto připravené testy jsem následně spouštěl průběžně ve svém vývojovém prostředí. Díky rozšíření Visual Studio Code jsem dostával přehledné zpětné vazby o průběhu testu. Po odeslání přírůstku kódu do repozitáře byly tyto testy spouštěny během procesu kontinuální integrace [3].

3.1.4 CD

Poté, co se provedla CI fáze, následuje automatizované nasazení (fáze CD). Rozhodl jsem se realizovat tuto fázi opět pomocí GitLab CI. Do souboru `.gitlab-ci.yml` jsem přidal novou `stage` s názvem `deploy` a připravil jsem další dvě komponenty `job`. Úkolem `contianer:deploy` je kontejnerizovat sestavenu aplikaci a uložit sestavený obraz do registru, který je přímo u projektu v platformě GitLab. Úkolem `bin:deploy` je nahrání sestavených binárních souborů (aplikace) na veřejně přístupný server, odkud si jej mohou uživatelé stáhnout. K nahrání jsem využil zabezpečený protokol SSH a nástroj SCP. K ověření využívám SSH klíč. V automatizaci musí být privátní klíč, který automatizace používá ke komunikaci se serverem, kde se nachází veřejný klíč. Aby SSH spojení bylo ještě více zabezpečeno a nedošlo k úniku tohoto klíče, musel jsem klíč uložit jako protected file v nastavení automatizace. To zajistí uvolnění souboru pouze při automatizaci a pouze pro verze, které jsou vytvořeny ověřenými přispěvateli. Proces CD se spouští vždy, pokud je vytvořena nová verze (tag) a obě části vyžadují úspěšný průběh předchozích fází (build i test). Tato část automatizace dle definice také vyžaduje sestavené soubory z předchozích fází, které jsou sdíleny mezi jednotlivými fázemi. Takto sdílené soubory se nazývají artefakty.



Obrázek 3.6: Propagační web Oasis

Jakmile jsou soubory doručeny na veřejně přístupný server, má k nim již přístup webová aplikace, kterou jsem vytvořil jako rozcestník. Adresa webu je: https://oasis.rodina-dlouha.cz/. Web jsem realizoval pomocí nástrojů Golang, Gin, React, Tailwind CSS a DaisyUI. Je rozdělen na frontend a backend. Také jsem jej kontejnerizoval a připravil mu kompletní CI/CD. Hlavním úkolem webu je přehledně a jednoduše seznámit uživatele s nástrojem a s tím, jak jej použít. Pro detailnější práci se pak web odkazuje na dokumentaci.

Obrázek 3.7: Ukázka rozhraní sloužícího ke stažení nástroje Oasis

Na webu jsem vytvořil také sekci, kde si návštěvník vybere platformu, pro kterou si chtějí nástroj stáhnout a web jim nabídne rozhraní pro stažení nejnovější verze Oasis a také možnost prohlížení a stažení starších verzí. Web také poskytuje návod, jak použít kontejnerizovanou verzi.

Obrázek 3.8: QR kód odkazující na stránku https://oasis.rodina-dlouha.cz

3.1.5 DOKUMENTACE

Jak ukázala rešerše a úvodní analýza, je třeba uživateli poskytnou přehledný návod, jak nástroj využít. Z tohoto důvodu jsem se rozhodl vytvořit dokumentaci. Zároveň smyslem dokumentace také mělo být poskytnout informace, jak postupovat, pokud by chtěl někdo produkt rozšířit.

Dokumentace je veřejně přístupná na adrese https://oasis.rodina-dlouha.cz/docs/index.html. Zvolil jsem dokumentační nástroj Sphinx. V tomto nástroji se píší jednotlivé dokumentační stránky pomocí formátu RTF s využitím jazyka Python. Poté dle definice nástroj generuje výstup, kterým je v mém případě statická webová stránka [35].

Platformu jsem zvolil na základě její znalosti. V praxi tuto platformu využívám. Oproti souborům `README` Sphinx poskytuje více možností, provázanosti, stavebních bloků a dokumentaci poskytuje jako jeden ucelený a přehledný zdroj. Soubory `README` jsem využil pro velmi rychlé popsání obsahu repozitáře a jako referenci na příslušné části dokumentace [36].

Sphinx však přináší překážky ve smyslu sestavení a nasazení. Tuto problematiku jsem opět vyřešil pomocí CI/CD, kdy se automatizace spouští, pokud je vydán tag. Jednotlivé verze označuji dle sémantického verzování. Dokumentaci uchovávám v odděleném repozitář, na který se odkazuji v `README` Oasis. Adresa je https://gitlab.com/david.dlouhy1/oasis-docs. Výhodou tohoto řešení je separace verzování dokumentace a separace verzování Oasis. Lze uchovávat obě části v jednom repozitáři, nicméně tím se zhorší přehlednost a musí se řešit různé automatizační podmínky. Prostředí pro psaní dokumentace jsem opět definoval pomocí vývojového kontejneru.

Nástrojem lze generovat různé výstupní formáty (web, PDF, Markdown aj.). Také lze odděleně držet vzhled a obsah. Využil jsem možnosti použít šablonu Read the Docs. Tato kombinace nástroje a šablony je oblíbená u linuxových vývojářů [36].

Obrázek 3.9: Dokumentace nástroje Oasis

Dokumentační platformu jsem použil také pro její schopnosti čtenáře lépe soustředit na její obsah. Hezkou ukázkou je použití bloků, jakými jsou poznámka, varování či nebezpečí. Čtenář je tak přehledně informován, a to zvyšuje jeho spokojenost s nástrojem Oasis. Uži-

tečným blokem je také vizualizace kusů kódů a obsahů souborů. Jedná se o blok, který dle nastaveného jazyka formátuje a vykresluje syntaxi. Oproti obrázku má toto obrovskou výhodu v možnosti kopírování. Dále pak platforma nabízí funkce, které jsem využil minimálně, a naopak také hojně užívané, avšak klasické bloky, jakými je nadpis, odstavec, odkaz a další. Nespornou výhodnou je také možnost systematické reference na různé části dokumentace. Platforma také poskytuje přehlednou automaticky generovanou navigaci a funkci vyhledávání skrze celou dokumentaci [36].

Obsah jsem systematiky rozdělil na sekce, dle zájmu uživatele. V první informativní části popisuji obecné informace o dokumentaci a projektu. V další sekci referuji na tuto písemnou práci. Sekce má totiž za cíl seznámit čtenáře s nástrojem, jeho návrhem, implementací a procesem jeho vzniku. Čtenář by po přečtení dané sekce měl být schopen dopsat vlastní adaptéry. Třetí sekce vysvětluje, jak nástroj Oasis použít. Po předání základních informací čtenáři mu jsou nejprve představeny požadavky na něj jako uživatele, požadavky na HW a také požadavky na SW. Předávám tak informaci o tom, co předpokládám, že by čtenář měl znát. Následuje popis deklaračních možností. Čtenář je tak již schopen sepsat vlastní deklarační soubory a ví, jaký má být jejich obsah. Závěrem sekce pak popisuji, jak deklaraci aplikovat. Do dokumentace jsem také zahrnul sekci, ve které popisuji, jak nástroj využít pro konkrétní problémy. Jako referenční příklad použití jsem zvolil OS vytvořený pomocí projektu Yocto Project.

Dokumentaci jsem psal pro potenciální koncové uživatele, kteří budou Oasis používat pro budování DevOps a mají technické vzdělání. Její obsah cílí na rychlé a jednoduché použití. Uživateli se snažím přehledně předávat jednoduché navazující kroky.

3.2 DEKLARACE POŽADAVKŮ

Nástroj Oasis požadoval jednoznačnou strukturu, kterou by poté mohl procházet a vyčítat informace. Na názvu souboru nezáleží. Vždy nástroj pracuje s obsahem souboru. Je ale třeba, aby soubor byl ve formátu YAML a měl následující strukturu.

Soubor s definicí Oasis musí začínat klíčem `oasis`, tím nástroj pozná, že se jedná o soubor s definicí Oasis. Poté uživatel musí vložit do tohoto klíče klíč `name`, který reprezentuje jméno projektu. Druhým vnořeným klíčem je `description`, který slouží pro uživatelský po-

pis. V obou případech se jedná o řetězce znaků. Tento definiční soubor musí být pouze jeden.

```yaml
---
oasis:
  name: "Example Embedded Oasis"
  description: "This is an example."
```

Zdrojový kód 3.1: Struktura souboru s definicí metadat Oasis

Soubor s definicí použitelných technologií musí začínat klíčem `technologies`. Následuje vnořený list technologií, kdy každá technologie má vždy klíče `name`, `type` a `definition`. Klíč `name` určuje název technologie. Jedná se o řetězce znaků, který nástroji pomůže určit technologický adaptér, ale také funguje jako reference pro pozdější odkazování v souboru s definicí projektu. Klíč `type` je list řetězů znaků. Položka říká, jaký port Oasis použije. Účelem je také správně propojit technologie a projekt. Klíč `definition` má individuální obsah, dle dané technologie. Některé technologie vyžadují více specifikací od uživatele, například jim nestačí pouze ověřovací token. Pro větší bezpečnost nástroj pracuje se systémovými proměnnými, tedy v klíči `definition` lze vytvářet vnořené klíče, kdy obsah nemusí být řetězec znaků přímo v souboru, ale lze použít řetěz uložený v systémové proměnné. Na tuto proměnnou se následně stačí v souboru referovat. Souborů s definicí technologií může být více. Nástroj provede spojení.

```yaml
---
technologies:

  - name: GitLab
    type:
     - vcs
     - cicd
    definition:
      token: $OASIS_GITLAB_TOKEN

  - name: Jira
    type:
     - plan
    definition:
      url: $OASIS_JIRA_URL
      mail: $OASIS_JIRA_MAIL
      token: $OASIS_JIRA_TOKEN
```

Zdrojový kód 3.2: Struktura souboru s definicí technologií

Posledním typem, je soubor, který reprezentuje projekty. Tento soubor musí začínat klíčem `projects`. Následuje vnořený list konkrétních projektů. Název projektu je reprezentován klíčem `name`. Klíč `type` říká, o jaký typ projektu jde. Hodnotu musí vybrat uživatel dle dokumentace nebo ukázkových příkladů. Nástroj bude dle tohoto typu vytvářet obsah repozitáře. Klíč `vcs` říká, jaký verzovací systéme má být využit. Jedná se o referenci na definovanou technologii. Klíč `infra` projektu přiřadí technologický adaptér pro práci s infrastrukturou, klíč `plan` projektu povolí a přidělí technologii pro DevOps fázi plánování. Klíč `monitor` provede to samé, ale pro fázi monitoring. Klíče `plan` a `monitor` jsou nepovinné, a pokud nejsou zadány, technologická funkce nebude projektu přidělena. Všechny klíče projektu jsou řetězce znaků. Souborů s projekty může být více. Nástroj je případně spojí.

```yaml
---
projects:
  - name: "Oasis example Embedded OS project"
    type: "Embedded OS (Linux)"
    vcs: GitLab
    infra: Hetzner
    plan: Jira
    monitor: Prometheus
```

Zdrojový kód 3.3: Struktura souboru s definicí projektu

3.3 JÁDRO

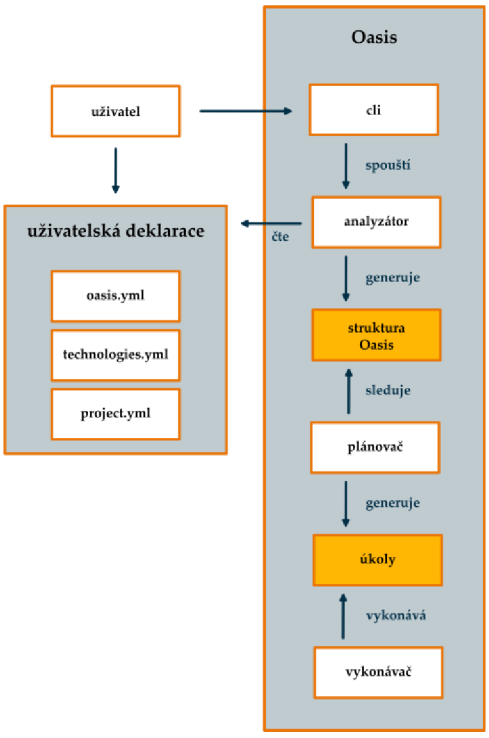
Při implementaci jádra nástroje jsem dle návrhu využil programovací jazyk Golang a Hexagonální architekturu. Snažil jsem se nástroj psát podle doporučených postupů. Implementaci jsem se snažil realizovat tak, aby byl chod snadno čitelný a jednoduché věci byly řešeny jednoduše.

Jádro jsem vytvořil tak, aby bylo správně implementováno vůči architektuře, ale zároveň aby jednotlivé komponenty jádra byly snadno nahraditelné, či aby se na ně daly napojit další systémy. Díky tomu je nástroj snadno rozšiřitelný například o grafické rozhraní.

Dle chronologického postupu požadavku uživatele jádrem jsem nejprve implementoval subsystém zodpovědný za analýzu deklarace uživatele. Uživatel komunikuje s adaptérem CLI a ten s vnitřní logikou aplikace. Vnitřní logika zajistí postupné zajištění požadovaného stavu.

Po analýze požadavků jsem implementoval navazující logiku zjištění vazeb. Kód jsem doplnil také o možnosti grafického výstupu (nakreslení strom). Následně jsem tyto informace musel zpracovat. Implementoval jsem tedy generátor konkrétních úkolů. Výstupem generátoru vznikl strom s úkoly, který bylo už jen třeba správě použít. Naimplementoval jsem tedy komponentu, která tento strom prochází, a tvořil optimalizovanou frontu s podporou paralelního spouštění. Frontu následně předávám komponentě vykonavač. Vykonavač jsem implementoval tak, aby dokázal správně sekvenčně zpracovat příchozí úkoly, ale také aby i celé vykonávání dokázal orchestrovat a pracovat s nečekanými a chybovými stavy. Při im-

plementaci jsem se inspiroval problémem dvou generálů, který se týká problémů synchronizace a spolehlivosti v distribuovaných systémech. Dva generálové, kteří chtějí společně útočit na nepřítele, musí komunikovat přes důvěryhodný kanál. Problém spočívá v tom, jak dosáhnout dohody na společném plánu útoku při absenci způsobu, jak ověřit, zda zprávy dorazily bez úprav. Jedním z mnoha řešení je vyslání několika poslů. Tím se zvýší šance na úspěch. Má to sice svá úskalí a neřeší to vše, ale u těchto distribuovaných problémů musíme vždy počítat s určitou pravděpodobností ztráty spojení a uživatele tak informovat o neúspěchu a chybě síťového spojení [37]. Podobné řešení využívá také Ansible [19].



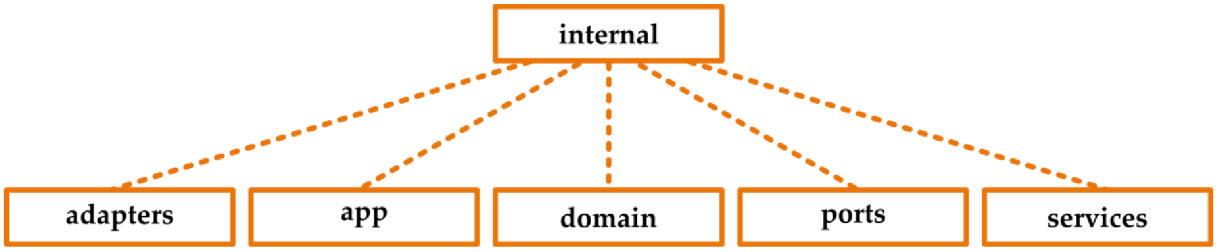
Obrázek 3.10: Schéma průběhu vykonání požadované deklarace v nástroji Oasis

3.3.1 HEXAGONÁLNÍ ARCHITEKTURA

Při implantaci nástroje jsem nejprve dle kapitoly č. 2.3 použil plochou strukturu. Vytvořil jsem první prototyp a postupně s novými přírůstky jsem architekturu modifikoval. Postupně jsem tímto rektorováním s implementací vytvořil aplikaci s Hexagonální architekturou. Tento postup jsem zvolil na základě doporučeného postupu [27].

Výsledná architektura obsahuje v kořenovém adresáři soubor `main.go`. Soubor inicializuje celou aplikaci a pracuje s dalšími komponentami. Dále se zde nacházejí soubory pro definici použitých balíčků a podpůrné soubory. Obsahem jsou také podpůrné adresáře, které se architektury netýkají. Architektury se týká pouze adresář `internal`. Jedná se o doporučený postup, jak držet systematicky jednotlivé části architektury.

V `internal` jsem připravil adresář `app`. Ten obsahuje strukturu, která reprezentuje Oasis. Tím jsem zvýšil přehlednost aplikace. Navíc tento balíček `app` je možné importovat i v jiných projektech.

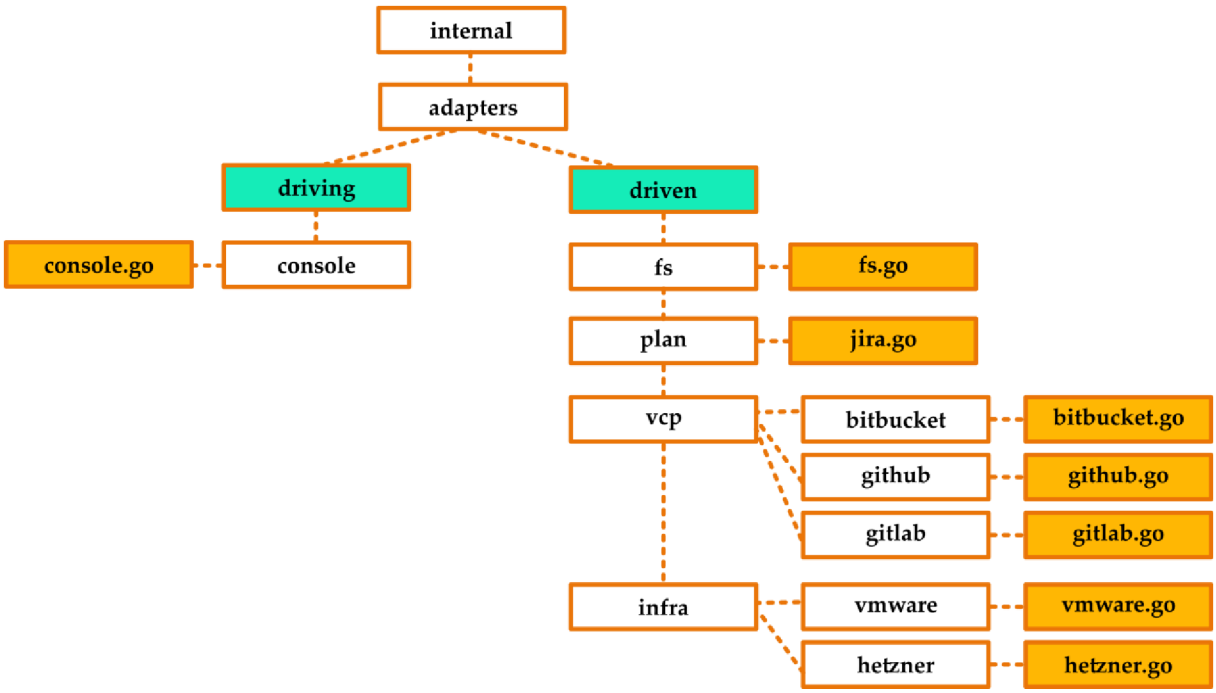


Obrázek 3.11: Adresářová struktura Hexagonální architektury

Porty jsem definoval pomocí rozhraní. Pro lepší organizaci mám tři různé soubory pro řídicí a řízené adaptéry a také rozhraní pro služby. Vše držím v balíčku `ports`. Později v kódu tato rozhraní využívám k dependenci injection. Tedy během běhu aplikace spojuji konkrétní technologicky závislé adaptéry s obecným předpisem (porty). Vše se spojuje ve strukturách služeb, adaptérů a definice aplikace. Jednotlivé součásti na sebe musejí držet reference.

V `adapters`, kde jsou jednotlivé adaptéry, mám od sebe oddělené řídicí a řízené adaptéry pro lepší čitelnost. Také mám v podadresářích i adresáře seskupující adaptéry dle portů. Tedy adaptéry pro verzovací systémy, pro virtualizační platformy a další. Každý adaptér tvoří vlastní balíček a realizoval jsem jej jako strukturu, která má své funkce a kontraktor. Konstruktor dle doporučeného postupu vrací ukazatel na vytvořenou strukturu [27]. Adaptéry mají portem definované funkce. Tyto funkce jsem implementoval jako veřejné (velké písmeno). Pomocné funkce jsem neexportoval a realizoval je jako interní funkce (začínají

malým písmenem). Adaptéry komunikují se zbytek aplikace výhradně pomocí doménových objektů. Musejí si také držet reference na jednotlivé služby. Vzhledem k tomu, že jsem jako CLI framework použil nástroj Cobra, provedl jsem vlastní implementaci napojení na Cobra tak, aby toto napojení bylo realizováno jako adaptér. Standardně tomu tak není. Hlavní motivací, proč použít framework Cobra, byla potřeba funkce autocomplete pro různé konzole a požití vyladěného CLI namísto vlastní implementace.



Obrázek 3.12: Adresářová struktura adaptérů

Doménovou vrstvu (balíček `domain`) jsem implementoval v adresáři `domain`. Zde jsem ještě dle kontextu rozdělil jednotlivé části domény na separátní soubory. Nadefinoval jsem dle Domain Driven Design [27] jednotlivé objekty a jejich chování. Opět jsem využil práci s ukazateli a tvorbu struktur skrze konstruktor. Implementaci jsem ale nedržel striktně dle definice a optimalizoval jsem ji pro snadnější použití. Konkrétně jsem u primitivních datových typů netvořil jejich ekvivalent v podobě struktury. To by mělo smyslu u větších projektů a je možné, že pokud se nástroj bude rozšiřovat, pak bude nutný refaktoring [27]. V doméně jsem také nadefinoval důležité konstantní hodnoty v podobě konstant a také jsem

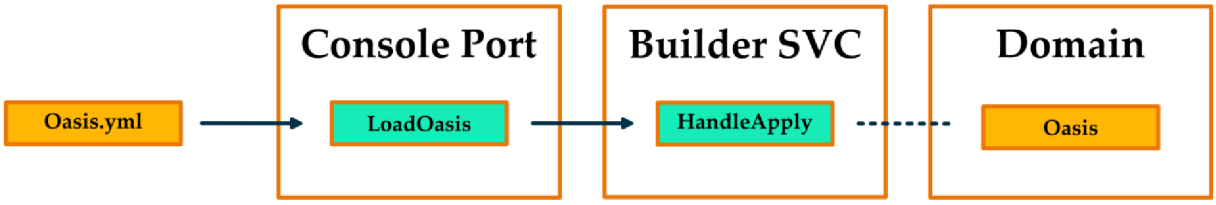
zpřehlednil nějaké hodnoty pomocí reprezentace výčtovými hodnotami. Jednotlivým částem struktur jsem také přidružil jejich klíče pro soubor formátu YAML.

Služby jsem umístil do adresáře service, kde každá služba má vlastní adresář. Například služba zodpovědná za sestavení infrastruktury a za naplánování a spuštění jednotlivých úkolů `BuildService` má svůj adresář `builder` a soubor `service.go`, kde je jednak struktura s konstruktorem reprezentující `BuilderService`, ale také tu jsou zde rozhraním definované funkce. Služba si drží referenci na adaptéry pravých stran a svůj výstup vrací typicky adaptérům levých stran, které jej předávají volajícímu aplikace.

Uživatel tedy zavolá nástroj Oasis a skrze parametry navolí požadovanou činnost a předá referenci na cestu k soubor s názvem `Oasis.yml`. Aplikace se po zavolání spustí a sestaví za běhu hexagon. Tedy služby, potřebné adaptéry atd. Požadavek uživatele je přijat adaptérem CLI. Adaptér volá službu pro načtení a vykonání deklarace. Služba se postará o logiku a pracuje s objekty definovanými v doméně. Pokud potřebuje služba komunikovat s externími systémy, využívá k tomu adaptéry pravých stran. Referenci na adaptéry jsem umístil do struktury služby. Takto běží požadavek zleva doprava a pak nazpět a uživatel je informován o průběhu.

3.3.2 SKENER SOUBORŮ YAML

Poté, co aplikace registruje požadavek na sestavení infrastruktury a přijme soubor `Oasis.yml`, zavolá se z adaptéru služba zodpovědná za načtení dat ze souborů YAML. Konktrétně se tato činnost děje v obslužné funkci `HandleApply` obsažené v adaptéru s názvem `console`. Definice adaptéru, tedy jeho port, se jmenuje `ConsolePort`.



Obrázek 3.13: Schéma komunikace komponent v Hexagonální architektuře při skenování souborů YAML

Dále jsem implementoval komunikaci konzolového adaptéru se službami, dle principů Hexagonální architektury. Tedy konzolový adaptér volá funkci `LoadOasis`. Tato funkce je součástí služby `BuilderService`. V této službě jsem implementoval dle návrhu doménovou logiku. Tedy práci s adaptérem systému souborů a nalezení všech souborů s definicemi a převedení dat z těchto souborů do paměti aplikace v podobě doménových struktur (kolekce položek) [20].

Struktury jsem definoval v doménové vrstvě v souboru `oasis.go`. Pro jednoduché použití jsem dodělal o funkci, která se stará o vytvoření struktury, tedy "konstruktor". Strukturu také rovnou předávám tímto "kontraktorem" jako ukazatel na danou struktur. Tím jsem zajistil efektivnější práci s pamětí po vzoru C/C++ [20]. Každé struktuře jsem přidělil její položky včetně příslušných datových typů. Každá struktura má také přiřazené klíče, podle kterých se později provádí převod souboru YAML na strukturu v paměti aplikace.

Typy struktur jsem implementoval dle principů Domain Driven Design s modifikací [3]. Neimplementoval jsem pro každou položku struktury vlastní datový typ, ale využil jsem zabudované datové typy jako je `string`, `int` a další. Implementoval jsem pouze tehdy, pokud se nejednalo o zbytečné komplikování. Pokud by se později tato potřeba projevila, díky Hexagonální architektuře by refaktoring byl snadný a rychlý. Zároveň by tento úkol byl také vhodný i pro umělou inteligenci [27].

Službu jsem implementoval tak, aby konkurentně spustila dvě podúlohy. Tímto přístupem jsem optimalizoval rychlost sestavení struktury Oasis [20]. Tyto dvě podúlohy bylo třeba synchronizovat. Tedy funkce služby musela počkat na jejich kompletní dokončení. To jsem zajistil pomocí `WaitGroup`, což je synchronizační primitivum jazyka Golang. Jedná se o jednoduché počitadlo, které se dá inkrementovat, deklarovat a zjišťovat, zda je nastaveno na hodnotu 0 [27]. Protože jsem spouštěl podúlohy dvě, inkrementoval jsem `WaitGoup` o 2 a následně jsem spustil dvě podúlohy, kdy každá podúloha dostala referenci na `WaitGroup` a pomocí konceptu `defer` toto počítalo o 1 dekretovala při ukončení své činnosti. Konstrukce `defer` je koncept Golang, kdy lze nastavit v libovolném místě v kódu funkce příkaz, který se vykoná těsně před ukončením funkce [20]. Jakmile tedy obě funkce dokončí svou činnost, bude `WaitGroup` nastaven na 0. Čekání na to, až bude `WaitGroup` nastaven na 0, jsem zajistil zavoláním funkce `Wait` na daném `WaitGroup`. Poté je čekání ukončeno a pokračuje se ve vy-

konávání kódu, respektive ukončení funkce `LoadOasis` a vrací se reference na strukturu obsahující popis Oasis.

Obrázek 3.14: Schéma komunikace dvou podúloh skrze kanál

Obě podúlohy spolu musí komunikovat a předávat si data. K tomu jsem využil kanál. V tomto kanálu průběžně posílá podúloha, zodpovědná za nalezení Oasis souborů, cesty k souborům. Jakmile je její činnost ukončena, uzavře kanál a tím signalizuje, že již žádná další data nebude třeba zpracovávat. Funkce zodpovědná za načtení včetně převodu dat na struktury naslouchá kanálu, a pokud se objeví nová cesta, provede svou činnost, pokud se objeví signál, který říká, že byl kanál uzavřen, pak ukončí svou činnost.

Během vykonávání může také dojít k chybám. Pro správné zpracování těchto chyb jsem vytvořil opět kanál, do kterého mohou obě podúlohy zasílat chyby, ke kterým došlo. Tyto chyby bylo také nutné zpracovat. Proto jsem ke dvěma původním podúlohám spustil třetí, která naslouchá na kanálu, dokud je otevřena, a pokud se objeví chyba, provede obslužnou akci. Činnost této podúlohy je ukončena uzavřením kanálu. Toto uzavření jsem implementoval v podúlohách. Nebylo třeba, abych zde pracoval s `WaitGroup`. Z principu není problém, aby podúloha "přežila" svou rodičovskou úlohu. Navíc je vždy ukončena uzavřením kanálu.

Všimneme si, že doteď jsme neřešili implementaci podúloh. Správná implementace Hexagonální architekty totiž musí odstínit logiku od konkrétní technologické implementace. Samotná práce se systémem souborů, jako je průchod adresářů a práce se souborem, je v kompetenci adaptéru pro systém souborů.

Podúloha zodpovědná za načtení cest k souborům volá funkci adaptéru s názvem `GetOasisFiles`. Tato funkce adaptéru parametrem přebírá cestu zadanou uživatelem

z konzole. Pokud je cesta neplatná, dojde k předání chyby do chybového kanálu a činnost je ukončena. Pokud je cesta platná, pak se projde aktuální adresář a získají se všechny soubory a adresáře. Zde bylo třeba, abych rozlišil chování funkce, pokud při průchodu narazí na soubor či na adresář. Pokud funkce narazí na soubor, předá cestu k souboru do kanálu a svou činnost ukončí. Pokud funkce dostane vstupním parametrem cestu, která vede do adresáře, provede se prohledání tohoto adresáře a rekurzivně se spustí pro každou položku opět tato funkce. Toto rekurzivní volání jsem navíc realizoval jako podúlohu (gorutina). Pro celkovou synchronizaci jsem musel navyšovat vždy pro každou podúlohu výše zmíněnou instanci `WaitGroup` o 1.

Nyní, když je jasné, jakým způsobem se plní fronta úloh, vysvětlím poslání funkce adaptéru systému souborů s názvem `LoadOasisFiles`. Její zodpovědností je zpracovat soubory a načíst struktury do paměti (parsing) programu. Tato funkce si drží mapu zpracovaných cest, aby nedošlo ke kolizi. Tu využívá při naslouchání kanálu, kde zpracovává příchozí cesty. Zpracované cesty následně v mapě označí jako zpracované. To je důležité v případech, kdy uživatel navíc využívá symbolické cesty a jiné referenční pomůcky.

Pokud cesta ještě nebyla zpracována, provede se privátní funkce `loadOasisFile`. Tato funkce již přímo zpracovává soubory YAML. Při implementaci jsem využil balík `github.com/go-yaml/yaml` a konkrétně funkci `Unmarshal`, která provede konverzi souboru YAML na strukturu domény dle definovaných klíčů, které jsem implantoval přímo v definici struktury dle doporučeného postupu [20]. Soubory jsou tříděny podle svého obsahu a identifikačního popisu dle kapitoly 3.2. Pokud se nejedná o definiční soubor, či soubor YAML, je uživatel informován o detekci vadného souboru. Rozhodl jsem se toto řešit jako propustnou chybu, která nemá vliv na pokračování vykonávání sestavení platformy. Uživatel ví, že se soubor nenačet, ale může mít v adresáři i jiné soubory než ty pro Oasis. Protože soubory mohou být pro různé struktury (`Oasis`, `Technology`, `Project` apod.), implementoval jsem konstrukci `switch`, která podle kořenového klíče souboru YAML říká, do jaké struktury se má provést `Unmarshal`. Jamile je vytvořena, struktura je přidána do výchozí struktury Oasis. Musí existovat právě jeden soubor YAML s rodičovským klíčem `oasis`. Pokud toto splněno není, dojde k ukončení nástroje a uživatel je o tomto faktu informován. U každého převodu souboru YAML na strukturu jsou kontrolovány správné datové typy formou nepropustné

chyby. Kontrolu povinných prvků YAML jsem implementoval na úrovni definice struktury. Pokud struktura neobsahuje povinný prvek, jedná se o nepropustnou chybu. Pokud naopak soubor YAML osahuje nějaký klíč, který je navíc, a nástroj s ním nepracuje, je tento fakt ignorován a není s ním dále pracováno. Na této úrovni jsem se rozhodl neimplementovat kontrolu povolených hodnot daných klíčů. V tuto chvíli se nejedná o zásadní kritérium. Pokud se později narazí na nepodporovanou hodnotu, je uživatel upozorněn, aplikace ukončí svůj běh a uživatel tak má informaci, kde si svou hodnotu v kódu může doimplementovat. Tím uživatele motivuji pro jeho doplnění kódu do platformy. Pokud by se ukázalo, že je třeba implementovat tuto kontrolu, implementoval bych opět přímo v definici struktury (doménová část), kde bych nadefinoval konkrétní omezení pro dané položky včetně případné gramatiky.

Oasis takto postupně konstruuje strukturu `Oasis`, která popisuje deklarovanou platformu. Struktura se skládá z uživatelem definovaného jména instance platformy Oasis (`Name`) a popisku (`Description`). Součástí je také struktura, která obsahuje pole technologií (`Technologies`), a struktura, která obsahuje pole projektů (`Projects`). Pole technologií jsem opět dle Domain Driven Design implementoval jako strukturu, která obsahuje pole struktur `Technology`. Obdobně jsem implementoval také projekty.

Struktura `Technology` se pak skládá z textového řetězce reprezentujícího název technologie (`Name`). Díky názvu jsem později mohl implementovat vazby. To samé platí i o poli řetězců reprezentujících typy dané technologie (`Definition`). Poslední položkou je pak mapa, kde je popsána definice dané technologie. Mapu jsem využil z důvodu rozmanitého množství volitelných položek pro danou technologii. Každá technologie má své specifické parametry a podobně.

Struktura `Project` se podobně skládá z názvu, kterým je řetězec znaků a jeho posláním jsou referenční možnosti. Pomocí řetězce `VCS` je projektu přiřazena technologie pro verzování. Infra definuje použitou technologii pro infrastrukturu a pomocí `Plan` jsem definoval použitou technologii pro plánování. Součástí struktury je také `Type`. Jedná se o definici typu projektu, zda se jedná o projekt pro Yocto Project a jiné. Tuto položku jsem implementoval odlišně. Posupoval jsem zde striktně dle Domain Driven Design [27] a vytvořil struktur `ProjectType`, která daný typ reprezentuje. `ProjectType` je struktura vycházející přímo z řetězce znaků. Smyslem zde bylo udělat větší přehled a propojit s konstantními hodnotami.

V hodnotách jsou například reference na projekty, názvy klíčů a jiné. Smyslem takové implementace bylo zlepšit přehlednost a udělat kód dlouhodobě udržitelnější [3]. Z těchto hodnot jsem vytvořil `enum` (výčtový typ) pro přehlednou práci a komparace zejména v konstrukcích `switch` [3].

Díky této implementaci jsem mimo jiné získal možnost uložit struktury do binárního souboru a později je snadno načíst bez nutnosti provádět převod soubor YAML. Také je tato reprezentace vhodná pro strojové zpracování, ale lze ji předat nástroji, který by platformě realizoval GUI.

3.3.3 GENERÁTOR STROMU ZÁVISLOSTÍ

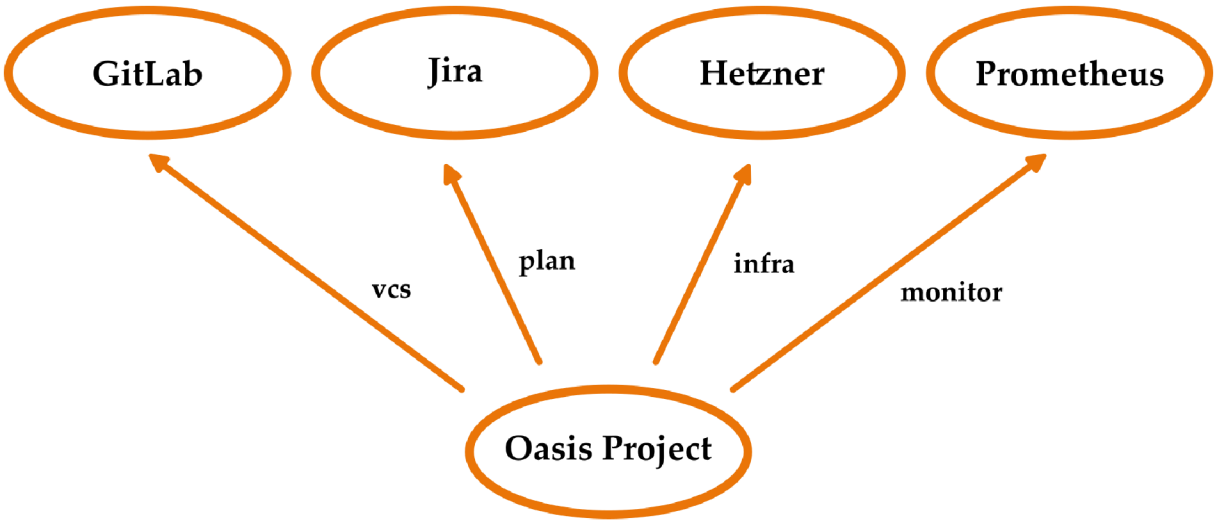
Poté, co dojde k načtení struktury `Oasis`, konzolový adaptér zavolá funkci `GenerateDependencyGraph`. Jedná se opět o funkci služby `BuildService`. Tento krok jsem neparalelizoval s načtením ani s pozdějším generováním stromu úkolů. Všechny tři kroky musí pracovat vždy se všemi informacemi získanými předchozích fází. Pokud bych paralelizoval, mohlo by dojít k porušení konzistence dat.

Funkce `GenerateDependencyGraph` vytvoří nejprve orientovaný acyklický graf jako prázdnou doménovou strukturu s názvem `domain.DAG`. Respektive pro lepší práci jsem vytvořit kontraktor, který mi vrací ukazatel na nově vytvořenou strukturu v paměti a s touto referencí se dále pracuje. Funkce sekvenčně detekuje uzly grafu pomocí interní funkce `detectVertexes`. Té se v kódu předává ukazatel na `Oasis` a na graf (strom zavilostí). Pokud tedy funkce provede úpravy techno struktur, bude referovaná struktura upravena a další funkce budou pracovat s těmito úpravami.

Detekování uzlů grafu projde technologie a projekty a pro každý výskyt vytvoří nový uzel, který je do grafu přidán. Aby uzel mohl reprezentovat jak projekty, tak i technologii, musel jsem uzel reprezentovat jako strukturu `DAGVertex`, který má položku `Value` typu rozhraní (`interface{}`). Později lze datový typ zajišťovat pomocí konstrukce switch. Jedná se o standardní přístup. Golang není navržen pro práci s OOP [20].

Poté interní funkce `detectEdges` projde projekty obdobně jako předchozí funkce a detekuje orientované hrany mezi uzly. K tomuto účelu využívám vnoření cyklů. Při průchodu

projektů procházím jednotlivé technologie a páruji typ technologie a typ určité vlastnosti dané technologie. Tedy pokud `projects.VCS` obsahuje řetězec `GitLab` a `technology.Name`, je také `GitLab`, dojde ke spárování. Porovnávám řetězce převedené na pouze malá písmena. Pro porovnávání využívám konstrukce `switch`. Pokud daná vlastnost nebyla ještě implementována, aplikace ukončí svou činnost a uživatel je o této informaci zpraven. K tomu využívám mechanismus `panic`.



Obrázek 3.15: Ukázka stromu závislostí

Takto vytvořený strom závislostí se předává zpět konzolovému adaptéru. Ten jej předává k dalšímu zpracování. Strom je také uživateli vygenerován ve formátu SVG. Tuto funkcionalitu jsem implementoval v adaptéru systému souborů, kde využívám knihovnu `graphviz`, která takový graf dokáže vizualizovat. V adaptéru tedy provádím konverzi struktury `DAG` na struktury dané knihovny. Tvorba souboru je spuštěna konkurentně při vykonávání stromu úkolů ve vlastním vlákně (gorutina).

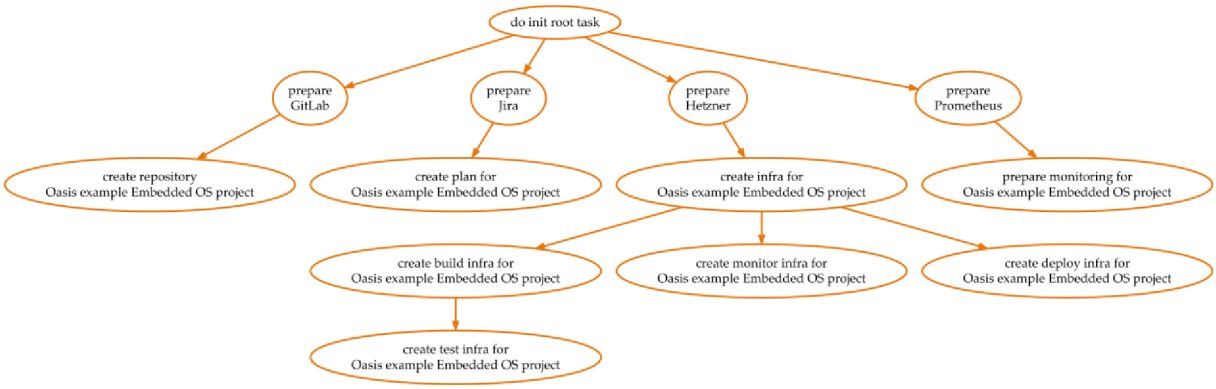
3.3.4 GENERÁTOR STROMU ÚKOLŮ

Konzolový adaptér dále volá funkci `GenerateTaskTree` služby `BuilderService` a předává jí obdobně jako u předchozího kroku ukazatele na strukturu `Oasis` a strom zavilostí. Funkce nejprve pomocí konstruktoru vytvoří ukazatel na strukturu `TaskTree` z doménové vrstvy.

Dále najde kořenové uzly. Tedy uzly, do kterých nevedou žádné hrany. Pro efektivity využívám strukturu mapa (`map`).

Následně je volána interní funkce `bfs`, která vyžaduje parametry předané reference na kořenové uzly, referenci na strom závislostí a referenci na strom úkolů. Funkce prochází strom zavilostí a strukturu `Oasis`. Nejprve funkce vytvoří pole struktur `QueueVertex`. Tento uzel ve frontě má referenci na uzel, ke kterému se vztahuje. Struktura dále obsahuje referenci na rodičovský uzel. Dále struktura nese referenci na hranu mezi aktuálním vrcholem a rodičem. Poslední položkou je reference na úkol ve stromu úkolů, na který tento úkol navazuje.

Funkce do výsledného stromu přidá inicializační úkol. Tento úkol nemá žádný výkonný smysl. Při zpracování stromu z něj však vychází vykonavač úkolů. Funkci si přebere předané reference na uzly grafu a přidá je do fronty. Protože se jedná o uzly bez rodičů, a tedy i bez příchozích hran, jsou tyto vlastnosti nastaveny na `nil`. Rodičovským úkolem ze stromu úkolů je výše zmíněný inicializační úkol.



Obrázek 3.16: Ukázka stromu úkolů, který byl vygenerován nástrojem Oasis

Následuje cyklus, který běží, dokud jsou položky ve frontě. Cyklus odebere první uzel z fronty a pomocí konstrukcí `switch` vygeneruje konkrétní kroky. Tyto kroky následně funkce přidá do výsledného stromu úkolů, a to včetně hran přechodu. Dětské uzly tohoto uzlu jsou následně předány do fronty. Zde jsou však k dispozici již informace o rodičovském uzlu a typu hrany. Také rodičovský úkol v tomto případě již není inicializační úkol, ale předešlý úkol. Celý cyklus se následně opakuje.

Vnořené konstrukce switch detekují typ aktuálního uzlu stromu zavilostí. Používá se k tomu hodnota, která říká datový typ dané položky. V Golang lze tuto informaci zjistit snadno pomocí `item.(type)` a návratovou hodnotu porovnávat konstrukcí switch, kde se testují různé datové typy. Implementoval jsem takto dle doporučeného postupu [20].

Pokud se jedná o projekt, použije konstrukci `switch` k detekci typu rodiče. Povolená je v tuto chvíli pouze kombinace technologie jako rodiče a potomka jako projektu. Dále se pomocí další vnořené konstrukce `switch` ověří typ hrany. Pokud se jedná o hranu typu "projekt má verzovací systém", typu "projekt má infrastrukturu v ", typu "projekt má monitoring" a další. Jakmile je jasné o taký typ vazby jde, je potřeba ještě vybrat konkrétní technologii. K tomu opět využívám vnořenou konstrukci `switch`. Nepodporované technologie vyvolají `panic`. V této nejvíce vnořené části se připraví technologický adaptér a vezmou se informace z definiční mapy. Vytváří se takto vzor vkládání zavilostí (dependency injection) a krátí se tak zápis kódu. Neopakují se ty samé kousky kódu. V konstrukci je vytvořena vždy alespoň jedna, ale i více doménových struktur `Task`. Tyto struktury jsou následně přidány do stromu úkolů (struktura `TaskTree`). Také se přidá hrana rodičovského úkolu a aktuálně vytvořeného. Struktura Task má svůj název, který je při vykonávání uživateli vypsán, a také v sobě nese anonymní funkci, kterou uživatel vloží pomocí dependency injection. Tato funkce je konkrétní sada volání služeb a adapterů, které vedou k požadovanému úkolu. Například vytvoření nového projektu v technologii GitLab. Takto je zabalen úkol a připraven na vykonání.

Pokud se jedná o technologii, použije funkce konstrukci `switch` k detekci konkrétní technologie a zjistí technologické předpoklady, které jde splnit před použitím technologie. Pokud takové předpoklady existují, vygenerují se obdobně jednotlivé kroky ke splnění a všechny se následně ve správném pořadí přidají do stromu úkolů.

Pokud typ není ani projekt, ani technologie konstrukce panic, ukončí běh aplikace a upozorní na tuto skutečnost vývojáře. Aby mohlo dojít k výskytu nového typu, musí právě dojít k zásahu vývojáře.

U stromu úkolů, jeho operacích a úkolech jako takových jsem postupoval dle doporučení Domain Driven Design [27]. Mám tedy strukturu složenou z dílčích doménových struktur, které mají nad sebou definované funkce a také konstruktor, který vrací referenci na nově vytvořenou strukturu.

Výsledný strom úkolů je opět orientovaný acyklický graf, kdy se vždy vychází z inicializačního kořenového uzlu. Inicializační uzel se větví na dílčí úkoly a takto se postupuje dále až na nejvíce závislé úrovně. Lze takto snadno reprezentovat paralelizovatelné úkoly. Lze také reprezentovat nutné posloupnosti vykonávání a vykonání úkolů předpokládaných jiným úkolem.

3.3.5 VYKONAVAČ ÚKOLŮ

Nyní je jasně definováno, jaké úkoly je třeba udělat a máme jejich obslužné funkce. Z adaptéru konzole jsem implementoval volání funkce `BuildOasis`, která je součástí služby (struktury) `BuildService`.

Tato funkce přijímá jako vstupní parametr ukazatel na výše zmíněný strom úkolů. Ukazatel jsem použil z důvodu efektivní práce s pamětí a zrychlení procesu. Ve funkci se volají konkurentně dva podúlohy. Data si mezi sebou posílají přes Golang kanál. Synchronizaci jsem realizoval pomocí uzavření kanálu a Golang synchronizačního primitiva `WaitGroup`.

Obrázek 3.17: Komunikace dvou podúloh úlohy vykonávání úkolů

První konkurentně spuštěnou podúlohou (gorutina) je topologické řazení nad stromem úkolů. Funkce se jmenuje `topSort`. Jakmile je topologické řazení hotovo, funkce uzavírá kanál a tím říká i druhé podúloze o konci své činnosti. K uzávěru kanálu jsem využil Golang konstrukci `defer`, která je doporučeným způsobem [20]. Topologické řazení si nejprve zjistí, jaké uzly jsou bez rodičů. Konkrétně si zjistí počet závislostí, které jsou toho charakteru. Pokud jich je 0, pak se tato úloha posílá do kanálu ke zpracování. Při odeslání je také úloha ohodnocena úrovní. Tato informace je zásadní pro později konkurentní vykonávání úloh druhou podúlohou. Důležitým krokem je nejprve odebrat všechny vazby pro tento uzel,

tedy všechny vazby, kdy rodičem je právě daný uzel. Poté v kódu odebírám tento aktuální uzel z celého stromu a zvyšuji číslo úrovně. Tím vznikají opět uzly bez rodičů. Proto tuto činnost opakuji do doby, kdy je počet uzlů roven 0.

Druhou konkurentně spuštěnou podúlohou (gorutina) je vykonávání úkolů z fronty s názvem `doTasks`. Funkce bere sekvenčně jeden úkol z fronty za druhým, a pokud je úkol ze stejné úrovně jako ta předtím, pak se spustí podúloha (gorutina), která se stará o vykonání konkrétního úkolu, a také se nastaví interní `WaitGroup` pro synchronizaci. Výchozí úrovní je úroveň 0. Pokud se narazí na úlohu, která není ze stejné úrovně, počká se na dokončení všech podúloh spuštěných touto podúlohou. Následně se nastaví nová úroveň dle této a takto se postup dále opakuje. Svou činnost podúloha končí ve chvíli, kdy je fronta uzavřena první podúlohou a zároveň došlo k dokončení všech mechanismů `WaitGroup`. Tedy provedly se všechny gorutiny, které byly zodpovědné za provedení úkolu z fronty.

Výše zmíněná podúloha zodpovědná za provedení konkrétního úkolu je `doTask`. Funkce je velmi jednoduchá a pouze volá obslužnou funkci daného úkolu. Obslužná funkce je součástí struktury a nastavuje se během generování stromu úkolů (kapitola 3.3.4). Podúloha signalizuje dokončení své činnosti pomocí `defer` a `WaitGroup`.

Díky Hexagonální architektuře jsem neřešil na této úrovni vůbec implementaci komunikace s externími systémy a pouze jsem dostával informace o (ne)úspěšném provedení dané úlohy.

Při implementaci jsem také ověřil, že sekvenční zpracování bez konkurentního spouštění bylo výrazně pomalejší. Většina úloh byla totiž komunikace s externími distribuovanými systémy. Vzhledem k tomu, že systémy mají odezvu a nedělí se přímo o výkon mezi sebou, bylo by neefektivní čekat na dokončení jednoho úkolu a poté pokračovat jiným. Je ale velmi důležité hlídat splnění všech předchozích úloh = nelze spustit vše paralelně, protože úlohy mají mezi sebou vazby.

4 REFERENČNÍ POUŽITÍ

Posledním cílem práce bylo ověření nástroje na vývoji embedded OS za pomocí Yocto Project. Proto v následující části popisuji toto využití. Nejprve detailněji popíši modelovu situaci. Poté popíši jednotlivé kroky nástroje. Rozhodl jsem se také popsat i možné dlouhodobé využití. Na závěr pak nástroj a celkové řešení porovnávám s jinými. Cílem této části je předání uceleného postupu čtenáři pro splnění konkrétní situace včetně popisu chování nástroje.

4.1 DEFINICE MODELOVÉ SITUACE

Zadání práce vyžaduje, aby nástroj podporoval distribuci OS Linux vyvíjenou s podporou Yocto Project a DevOps. Sestavení a konfigurace výsledné infrastruktury musí být také částečně automatizovány. Pokryty musí být systémy pro řízení vývojového kódu a jeho verzí, dále systém určený ke kompilaci a sestavení produktu, a nakonec systém určený k testování výsledného SW.

Modelovou situaci jsem nad rámec zadání rozšířil o monitoring nasazeného zařízení a sestavení infrastruktury včetně grafických dashboards. Dalším rozšířením je i staging nasazení a kontrola zranitelností. Posledním rozšířením je vytvoření prostoru pro řízení projektu a plánování. Modelová situace je také vybrána tak, aby u ní bylo možné provést plnou automatizaci (nikoliv semi-automatizaci). V zadání není specifikována self-hosted nebo cloud infrastruktura.

V následující ukázce demonstruji využití poskytovatele cloudových služeb Hetzner. Pro plánování bude využit nástroj Jira. Pro automatizaci, CI/CD, bezpečnost a repozitáře projektu bude využit GitLab. Testování bude realizováno pomocí nástrojů ptest a statické analýzy CVE. Proces nasazení pak bude pokryt nasazením do QEMU, které bude přes specifický port vystaveno pro externí přístup. Monitoring zařízení bude realizován pomocí sady nástrojů Prometheus, Node Exporter a Grafana.

Nástroje Jira a GitLab budou použity jako oficiální služby (nikoliv self-hosted). Nástroje QEMU, GitLab Runner, Prometheus, Node Exporter a Grafana budou na ekonomicky výhodném typu virtuálního serveru se sdíleným procesorem. Během zkušebních provozů jsem

toto vyhodnotil jako nejlepší kompromis. Navíc je mé řešení do budoucna škálovatelné, a to jak horizontálně, tak i vertikálně.

Od uživatele se přepokládá základní znalost správy OS Linux, princip cloud služeb, pracovní stanice s připojením do internetu. Stanice musí mít OS Linux nebo OS Windows. Počítá se s architekturou amd64. Také se předpokládá vytvořený účet u oficiálních služeb GitLab a Jira včetně možnosti tvorby tokenů a klíčů. Také pokud by se chtěl uživatel přímo připojit na jednotlivé servery, je nutné, aby disponoval SSH klientem, stejně tak aktuálním webovým prohlížečem. Pro nástroj Oasis toto však není podmínkou.

4.2 APLIKACE NÁSTROJE

Nejprve je nutné, aby si uživatel stáhl nástroj. To provede intuitivně pomocí propagačních stránek, které jsem vytvořil jako součást práce. Nachází se na adrese: https://oasis.rodina-dlouha.cz/. Zde si uživatel vybere svůj OS a verzi 1.0.0. Doporučuji uživateli průběžně nahlížet do oficiální dokumentace na adrese: https://oasis.rodina-dlouha.cz/docs/index.html.

Po stažení je nutné vytvořit deklaraci. Pro tuto modelovou situaci jsem připravil již hotovu deklaraci ve složce `demo/embedded-os` v oficiálním repozitáři nástroje Oasis. Uživatel si tedy stáhne tuto složku do svého zařízení.

Z hlediska bezpečnosti je nutné, aby uživatel nastavil systémové prostředí. Tento krok lze obejít nahrazením proměnných v YAML deklaracích. Je nutné, aby uživatel důkladně zvážil bezpečnostní rizika. Konkrétně musí mít uživatel nastaven přístupový token GitLab (proměnná `OASIS_GITLAB_TOKEN`), přístupový token Hetzner (`OASIS_HETZNER_TOKEN`), svoji adresu cloud Jira (`OASIS_JIRA_URL`), mailovou adresu přiřazenou Jira (`OASIS_JIRA_MAIL`) a přístupový token Jira (`OASIS_JIRA_TOKEN`).

Je nutné je taky vytvořit složku `data`. Tuto složku je nutné vytvořit v umístění nástroje Oasis.

Nyní je možné nástroj spustit pomocí příkazu `oasis apply --file=demo/embedded-os/oasis.yml`. Oasis začne načítat interní strukturu. Sestaví strom závislostí a strom úkolů. Aplikace komunikuje výhradně s uživatelem skrze konzolový adaptér.

Oasis nejprve připraví technologické adaptéry GitLab, Jira, Hetzner a Prometheus. Jedná se o 4 konkurentně vykonávané úkoly. Poté se přechází na další úroveň, kdy se vytváří konkrétně zdroje a konfigurace nad těmito technologiemi. Jmenovitě se vytvoří repozitář v GitLab. Využívám k tomuto princip fork. Mám přípravné projektové šablony, které se uživateli nakopírují do jeho jmenného prostoru. Pokud by v šabloně došlo k nějaké změně, může si uživatel tuto změnu také přebrat. Druhou úlohou je vytvoření plánu na platformě Jira. Jedná se o Kanban projekt.

Dalším úkolem je vytvoření infrastruktury v cloud prostředí Hetzner. Nejprve se vygeneruje nový pár SSH klíčů. Tyto klíče se nahrají do cloud prostředí. Za asistence GitLab adaptéru se vytvoří GitLab Runner pro automatizaci build, test a deploy fáze. V platformě Hetzner adaptér vytvoří nový server typu `cpx41` s OS Rocky Linux 9.3. Jedná se o server se sdíleným CPU s architekturou `x86` a konfigurací 8 CPU, 16 GB RAM, 240 GB disk. Cena je cca 30 € za měsíc provozu. K serveru je přiřazena veřejná IPv4 adresa. Automatizace si tuto informaci zjistí sama. Uživatel IP adresu zjistí v platformě Hetzner. Poté, co dojde k vytvoření serveru, připojí se nástroj pomocí SSH k danému serveru a využije k tomu vygenerovaného klíče. Poté se spustí sada připravených příkazů.

V tuto chvíli se může objevit chybový výpis, který obsahuje podřetězec `connect: connection refused`. Pokud se objeví, stačí nástroj opětovně spustit a chyba se vyřeší.

Nejprve se manažerem `dnf` aktualizují balíčky a systém. Poté se nainstaluje technologie Docker s kontejnerizační platformou `contianerd`. Nainstalují se také základní nástroje pro správu serveru (`wget`, `htop`, `rsync`…). Automatizace také vytvoří chybějící složky pro nasazení `staging` a `test`. Dojde také k detekci volných portů a jsou navrženy volné porty pro použití. Připraví se také adresáře pro nasazení GitLab Runner a pro potřeby Yocto Project se nastaví Runner tak, aby pracoval s perzistentně uloženými daty typu `DL_DIR` a `SSTATE_CACHE`. Následně se spustí registrační kontejner a provede registraci za pomocí dat vytvořených z předešlé úlohy. Po registraci se Runner nasadí ne jako registrační, ale výkonný. Je označen jedinečným identifikátorem a každý projekt má tak svůj vlastní Runner paralelně na serveru.

Pro potřeby monitoringu se opět pomocí SSH zavolají příkazy pro nasazení Grafana stack. Nejprve se vytvoří potřebné adresáře pro perzistentní data, dále se vytvoří konfigu-

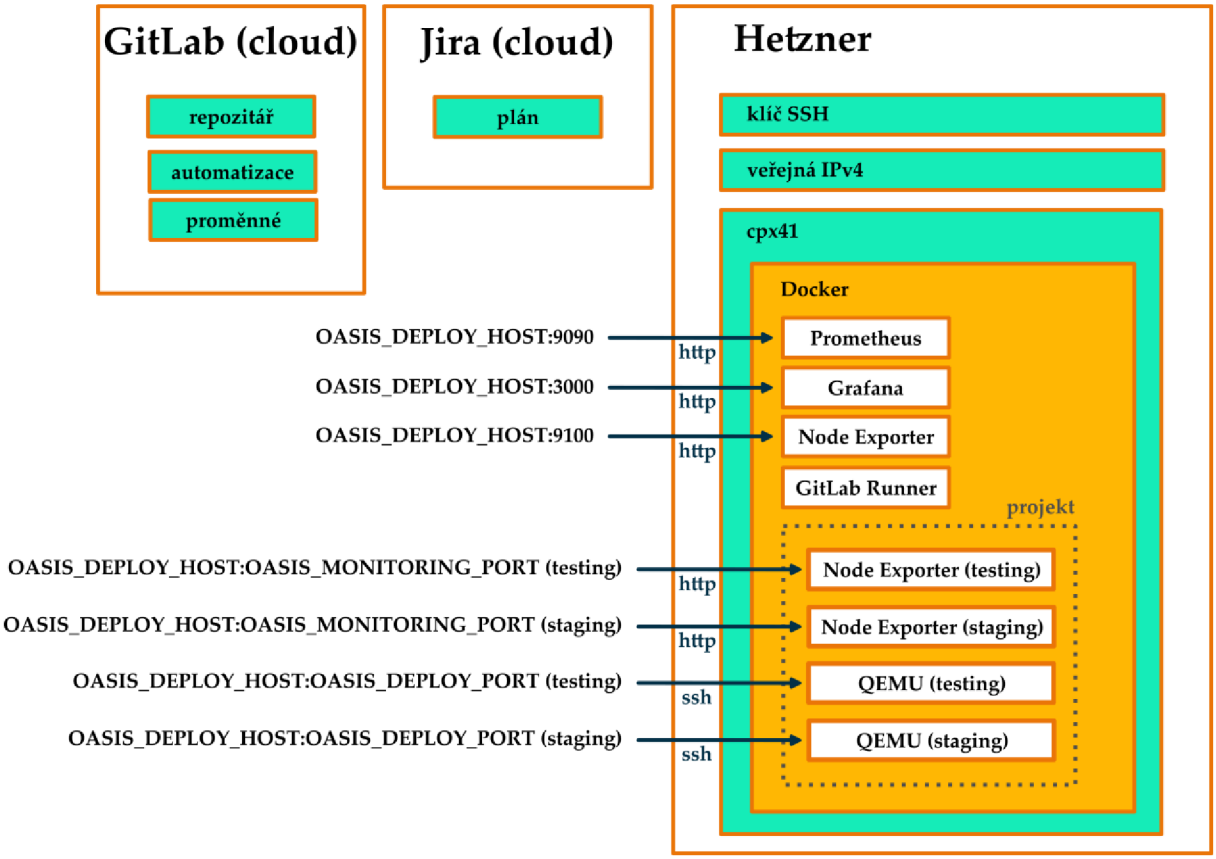
rační soubory a naplní se příslušnými daty. Poté se nasadí kontejnery s Prometheus, Node Exporter a Grafana. Uživatel se musí přihlásit na portu `3000` do Grafana. Při prvním přihlášení využije kombinaci admin a admin. Po přihlášení údaje doporučuji změnit a dále si zde udělat případně konektor a dashboard na technologii Prometheus. Poté již uživatel uvidí všechna svá data. Metriky však může sledovat i bez nástroje Grafana, a to na portu `9100` a `9090` a portech pro metriky.

Data projektu (porty projektu, SSH klíče, složky nasazení a další) jsou držena na bezpečném místě, a to v nastavení v GitLab v sekci CI/CD Variables. Uživatel si zde data může prohlédnout. Nástroj je na toto místo nahraje, aby je mohlo využívat CI/CD, potažmo Runner.

Jakmile je vše dokončeno, ukončí aplikace svou činnost kódem `0` a uživatel může infrastrukturu plně využívat. Klasická činnost by tedy byla naklonování kódu, spuštění vývojového kontejneru a vývoj samotný. Kód také obsahuje napojení na vrstvu `meta-oasis`, kterou jsem vytvořil pro potřeby CI/CD, a je třeba ji mít zahrnutou. Tato vrstva je držena v separátním repozitáři (https://gitlab.com/david.dlouhy1/meta-oasis) a je automaticky klonována do vývojového kontejneru. Poté, co uživatel provede změny, nahraje kód do repozitáře (`commit` & `push`). Aby uživatel spustil CI/CD, musí vytvořit v repozitáři `tag`. Pro přehlednost by uživatel měl využít sémantické verzování.

Po spuštění automatizace se nejprve vykoná fáze sestavení. Tuto činnost vykonává Runner v infrastruktuře v platformě Hetzner. Automatizace probíhá v kontejneru stejném, který je použit jako vývojový kontejner. Provedou se příkazy sestavení pro OS vyvíjený za podpory Yocto Project. Následný výstup je uložen jako artefakt, který je možné stahovat a nasazovat. Automatizace provede testování CVE a ptest. K tomu musí udělat automatizace testovací nasazení na server v platformě Hetzner. Pro jednoduchost vše běží na jednom serveru a vyžívá se nasazení do kontejneru, který zprostředkovává QEMU. Ke spojení automatizace využívá vygenerovaný privátní klíč a SSH. K lepší orientaci a pro možné uživatelské úpravy jsem v šablonovém repozitáři vytvořil soubor `docker-compose.yml`, pomocí kterého je nasazení do QEMU definováno. Takto se nasazuje i staging. Uživatel se pak na tento kontejner může připojit pomocí SSH a vygenerovaného privátního klíče. Metriky jsou rovněž

vystaveny na příslušných portech z Variables. Lze je také vyčítat v nástroji Grafana. Zde automatizace propojí pouze staging. Testovací nasazení nikoliv.



Obrázek 4.1: Přehled infrastruktury vytvořené nástrojem

4.3 DLOUHODOBÉ VYUŽITÍ

Nástroj Oasis lze využívat i dlouhodobě. Nástroj bude zajišťovat deklarovaný stav a aplikovat jej. U možných budoucích rozšíření je třeba, aby uživatel i vývojář na tuto zpětnou kompatibilitu myslel a pokud hrozí nějaké problematické stavy, musí být tato informace v dokumentaci.

Lze přidávat nové projekty a technologie. Stačí tedy deklarovat nový projekt a následně spustit nástroj Oasis. Každé spuštění také provádí průběžné aktualizace komponent. Kontejnery Docker se neaktualizují.

Editace existujících částí infrastruktury jsem neimplementoval. Nelze provozovat činnosti jako je změna názvu existujícího projektu. Oasis tuto úpravu pochopí jako nový projekt. To samé platí i pro technologie, ale i jednotlivé atributy.

4.4 KOMPARACE S JINÝMI NÁSTROJI

Oproti jiným nástrojům je Oasis zaměřen na rychlost zajištění požadovaného stavu. Uživatel také ocení jednoduchost v použití. Uživatel silně deklarativně popíše svou potřebu a nástroj zajistí vše potřebné, bez nutnosti specifikace. Tyto dvě vlastnosti byly Achillovou patou zmíněných nástrojů uvedených v kapitole č. 1.3 a kapitole č. 2.1.

Výhodou nástroje oproti jiným je jeho architektura, která je dle kapitoly č. 2.3 optimální pro open source vývoj takového nástroje. Díky vhodné volbě programovacího jazyka je nástroj nativně multiplatformní a lze jej sestavit pro různé OS a architektury. Stačí pokud danou kombinaci podporuje kompilátor jazyka Golang. Nástroj jsem také kontejnerizoval a uživatel může použít tuto oficiálně kontejnerizovanou verzi.

Nástroj je reprezentován jedním binárním souborem, který není závislý na knihovnách a aplikacích systému. Uživatel komunikuje s cloud infrastrukturou právě přes tento binární soubor. Oproti jiným nástrojům je tak instalace nástroje triviální a stačí pouze stáhnout jeden soubor. Nutným krokem je však také vytvoření deklaračních souborů a případné nastavení systémových proměnných.

Nástroji jsem také připravil podpůrné DevOps nástroje a nastavil příslušné automatizace tak, aby bylo možné snadno nástroj udržovat a rozšiřovat, a to i včetně DevOps cyklu pro dokumentaci.

Mé řešení je ale také potřeba ještě rozšířit o další adaptéry a jejich napojení na proces sestavení a konfigurace infrastruktury. Nástroj také potřebuje doladit do podoby, kterou si představují koncoví uživatelé a musí vyspět. Nelze jej tedy aktuálně nasadit plně do praxe a počítat s maximální spolehlivostí. Jedná se o funkční řešení, které je pokryto testy, ale bylo použito na příliš malém vzorku projektů a koncových uživatelích. Je tedy třeba do vývoje a použití nástroje zapojit komunitu. Nástroj tak může být postupně rozšiřován a laděn. Díky

architektuře a vhodnému rozdělení na komponenty a paralelizaci lze dodělat celému nástroji grafické webové rozhraní.

ZÁVĚR

Všechny cíle vytyčené v úvodu této práce se mi podařilo úspěšně naplnit a nástroj jsem úspěšně otestoval na požadované modelové situaci.

Vytvořil jsem přehled existujících nástrojů a řešení a popsal jsem jejich strukturu. Rešerši jsem doplnil o analýzu potřeb koncových uživatelů a analýzu problematiky vývoje embedded OS za podpory DevOps.

Vytvořil jsem obecný návrh nástroje, který své činnosti sestavení a konfigurace infrastruktury pro vývoj SW produktu separuje na komponenty a využívá efektivní práce s pamětí a paralelizaci činností. Na základně rešerše a komparace jsem navrhl použití jazyka Golang, použití Hexagonální architektury, definoval jsem funkcionality nástroje a zvolil jsem vhodnou open-source licenci. Navrhl jsem také základní technologické adaptéry, které pokryly všechny fáze DevOps cyklu. Zadání vyžadovalo pouze fáze kódování, sestavení a testování. Navrhl jsem také využití existujících grafových algoritmů a topologického řazení. Věnoval jsem se i návrhu komunikace uživatele s nástrojem.

Dle návrhu a principů SW inženýrství jsem nástroj implementoval. Mé řešení zohledňuje bezpečnost komunikace, a také adekvátně přistupuje k heslům tokenům a dalším tajným údajům. Kód jsem doplnil o kódové testy a vytvořil jsem základní dokumentaci ve Sphinx. Také jsem při implementaci vytvořil CI/CD pro vývoj nástroje a propagační web, který uživateli pomůže s rychlým startem s nástrojem.

Na závěr jsem úspěšně ověřil použití nástroje při vývoji embedded OS za podpory Yocto Project. Nástroj vytvořil kompletní DevOps infrastrukturu, a to plně automatizovaně. Výjimku tvoří vytvoření definičních souborů uživatelem a manuální nastavení v grafickém prostředí Grafana z fáze monitoring. Věnoval jsem se také porovnání svého řešení s ostatními a popsal jsem dlouhodobé využití nástroje.

Tato práce je navržena tak, aby byla rozšiřitelná komunitou. Rozšíření spatřuji v doplnění dalších typů projektů, v implementaci dalších technologických adaptérů a ve funkci dynamického vertikálního a horizontálního škálování. Také web by bylo možné rozšířit tak, aby poskytoval GUI pro nástroj.

POUŽITÁ LITERATURA

[1] SIMMONDS, Chris. *Mastering Embedded Linux Programming*. Second Edition. Packt, 2017. ISBN 9781787283282.

[2] SAKOVICH, Natallia. SAM SOLUTIONS. 4 Steps of the Embedded (or IoT) Product Development Life Cycle. SAM SOLUTIONS. *SaM Solutions* [online]. 2024 [cit. 2024-04-15]. Dostupné z: https://www.sam-solutions.com/blog/embedded-product-development-life-cycle/

[3] ANAGNOSTOPOULOS, Achilleas. *Hands-On Software Engineering with Golang*. Packt, 2020. ISBN 9781838554491.

[4] MCCARTHY, Matthew A., Lorraine M. HERGER, Shakil M. KHAN a Brian M. BELGODERE. Composable DevOps: Automated Ontology Based DevOps Maturity Analysis. *2015 IEEE International Conference on Services Computing* [online]. IEEE, 2015, 600-607 [cit. 2024-04-22]. ISBN 978-1-4673-7281-7. Dostupné z: doi:10.1109/SCC.2015.87

[5] *ECS SRIA* [online]. 2024 [cit. 2024-04-10]. Dostupné z: https://ecssria.eu/

[6] THE LINUX FOUNDATION. *Embedded Open Source Summit 2023* [online]. 2023 [cit. 2024-04-13]. Dostupné z: https://events.linuxfoundation.org/embedded-open-source-summit/

[7] THE LINUX FOUNDATION. *The Yocto Project* [online]. 2023 [cit. 2024-04-13]. Dostupné z: https://www.yoctoproject.org/

[8] HARON, Ahmad Safwan, Mohamad Sofian Abu TALIP, Anis Salwa Mohd KHAIRUDDIN a Tengku Faiz Tengku Mohmed Noor IZAM. Internet of Things Platform on ARM/FPGA Using Embedded Linux. *2017 International Conference on Advanced Computing and Applications (ACOMP)* [online]. IEEE, 2017, 99-104 [cit. 2024-04-22]. ISBN 978-1-5386-0607-0. Dostupné z: doi:10.1109/ACOMP.2017.26

[9] OZCELIKORS, Mustafa a Akin GUMUSKAVAK. Platform-independent Infotainment and Digital Cluster Development using Yocto Project. *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)* [online]. IEEE, 2020, 1-5 [cit. 2024-04-22]. ISBN 978-1-7281-7116-6. Dostupné z: doi:10.1109/ICECCE49384.2020.9179288

[10] OFFERMAN, Tyron, Robert BLINDE, Christoph Johann STETTINA a Joost VISSER. *A Study of Adoption and Effects of DevOps Practices* [online]. IEEE, 2022, 2022-6-19, 1-9 [cit. 2024-04-22]. ISBN 978-1-6654-8817-4. Dostupné z: doi:10.1109/ICE/ITMC-IAMOT55089.2022.10033313

[11] NORTHERN.TECH. *Mender* [online]. 2024 [cit. 2024-04-12]. Dostupné z: https://mender.io/

[12] MEMFAULT INC. *Memfault* [online]. 2024 [cit. 2024-04-12]. Dostupné z: https://memfault.com/

[13] AMAZON WEB SERVICES, INC. Ingesting, analyzing, and visualizing metrics with DevOps Monitoring Dashboard on AWS. AMAZON WEB SERVICES, INC. *Amazon Web Services* [online]. 2024 [cit. 2024-04-12]. Dostupné z: https://docs.aws.amazon.com/solutions/latest/devops-monitoring-dashboard-on-aws/solution-overview.html

[14] AMAZON WEB SERVICES. DevOps Monitoring Dashboard on AWS. AMAZON WEB SERVICES. *Amazon Web Services* [online]. 2024 [cit. 2024-04-12]. Dostupné z: https://aws.amazon.com/solutions/implementations/devops-monitoring-dashboard-on-aws/?did=sl_card&trk=sl_card

[15] ARORA, Tarun a Utkarsh SHIGIHALLI. *Azure DevOps Server 2019 Cookbook*. Second Edition. Packt, 2019. ISBN 9781788839259.

[16] COWELL, Christopher, Nicholas LOTZ a Chris TIMBERLAKE. *Automating DevOps with GitLab CI/CD Pipelines*. Packt, 2023. ISBN 9781803233000.

[17] ZUUL PROJECT CONTRIBUTORS. About Zuul. ZUUL PROJECT CONTRIBUTORS. *Zuul - A Project Gating System* [online]. 2024 [cit. 2024-04-13]. Dostupné z: https://zuul-ci.org/docs/zuul/latest/about.html

[18] Wind River Studio: A Tour of the Capabilities. WIND RIVER SYSTEMS, INC. *Wind River Software* [online]. 2024 [cit. 2024-04-12]. Dostupné z: https://www.windriver.com/studio/tour

[19] FREEMAN, James a Jesse KEATING. *Mastering Ansible*. 4th Edition. Packt, 2021. ISBN 9781801818780.

[20] DOAK, John a David JUSTICE. *Go for DevOps*. Packt, 2022. ISBN 9781801818896.

[21] KRIEF, Mikael. *Terraform Cookbook*. Packt, 2020. ISBN 9781800207554.

[22] LEHMAN, Noah. THE LINUX FOUNDATION. Linux Foundation Launches OpenTofu: A New Open Source Alternative to Terraform. THE LINUX FOUNDATION. *Linux Foundation* [online]. 2024 [cit. 2024-04-13]. Dostupné z: https://www.linuxfoundation.org/press/announcing-opentofu

[23] JOSHI, Mayank. *Mastering Chef*. Packt, 2015. ISBN 9781783981564.

[24] SANDILANDS, David. *Puppet 8 for DevOps Engineers*. Packt, 2023. ISBN 9781803231709.

[25] PULUMI. *Pulumi Docs* [online]. 2024 [cit. 2024-04-13]. Dostupné z: https://www.pulumi.com/docs/

[26] BRAUNTON, Alex. *Hands-On DevOps with Vagrant*. Packt, 2018. ISBN 9781789138054.

[27] LASZCAK, Robert a Miłozs SMÓŁKA. *Go with the Domain* [online]. 20210903. Three Dots Labs, 2021 [cit. 2024-04-13]. Dostupné z: https://threedots.tech/go-with-the-domain/

[28] VIEIRA, Davi. *Designing Hexagonal Architecture with Java*. Second Edition. Packt, 2023. ISBN 9781837635115.

[29] LI, Patrick. *Jira 8 Essentials*. Sixth Edition. Packt, 2022. ISBN 9781803232652.

[30] BASTOS, Joel a Pedro ARAÚJO. *Hands-On Infrastructure Monitoring with Prometheus*. Packt, 2019. ISBN 9781789612349.

[31] KNUTH, Donald Ervin. *The art of computer programming*. 3rd ed. Upper Saddle River, NJ: Addison-Wesley, c1997. ISBN 978-020-1896-831.

[32] AHMAD, Imran. *50 Algorithms Every Programmer Should Know*. Second Edition. Packt, 2023. ISBN 9781803247762.

[33] BRESNAHAN, Christine. *Linux Essentials*. Second Edition. Wiley, 2015. ISBN 9781119092063.

[34] MYŠKA, Matěj, Radim POLČÁK, Jaromír ŠAVELKA, Libor KYNCL a Iveta SVIRÁKOVÁ. *Veřejné licence v České republice* [online]. 2.0. Masarykova univerzita, 2014 [cit. 2024-04-15]. ISBN 978-80-210-7193-3. Dostupné z: https://www.fi.muni.cz/studies/verejne-licence.pdf

[35] JACKSON, Cody. *Learn Programming in Python with Cody Jackson*. Packt, 2018. ISBN 9781789531947.

[36] CHINCHILLA, Chris. *Technical Writing for Software Developers*. Packt, 2024. ISBN 9781835080405.

[37] RAITURKAR, Jyotiswarup. *Hands-On Software Architecture with Golang*. Packt, 2018. ISBN 9781788622592.

PŘÍLOHY

Tato práce neobsahuje žádné přílohy. Veškeré zdrojové kódy a součásti jsou ve veřejných repozitářích uvedených v textu práce.