VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA PODNIKATELSKÁ

FACULTY OF BUSINESS AND MANAGEMENT

ÚSTAV INFORMATIKY

INSTITUTE OF INFORMATICS

NÁVRH A IMPLEMENTACE DASHBOARDŮ VE FIREMNÍM PROSTŘEDÍ

DESIGN AND IMPLEMENTATION OF DASHBOARDS IN A CORPORATE ENVIRONMENT

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — **Daria Dmitriievich**

AUTHOR

VEDOUCÍ PRÁCE — **doc. András Rontó**

SUPERVISOR

BRNO 2025

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA PODNIKATELSKÁ

Zadání bakalářské práce

Ústav:	Ústav informatiky
Studentka:	**Daria Dmitriievich**
Vedoucí práce:	**doc. András Rontó**
Akademický rok:	2024/25
Studijní program:	Manažerská informatika

Garant studijního programu Vám v souladu se zákonem č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů a se Studijním a zkušebním řádem VUT v Brně zadává bakalářskou práci s názvem:

Návrh a implementace dashboardů ve firemním prostředí

Charakteristika problematiky úkolu:

Úvod
Cíle práce, metody a postupy zpracování
Teoretická východiska práce
Analýza současného stavu
Vlastní návrhy řešení
Závěr
Seznam použité literatury
Přílohy

Cíle, kterých má být dosaženo:

Cílem bakalářské práce je navrhnout a implementovat dashboardy pro tým ve společnosti, které umožní vizualizaci klíčových metrik a dat spojených s vývojem firmwarové komponenty MultiComm.

Základní literární prameny:

ATHICK, Asjad. Getting Started with Elastic Stack 8.0: Run powerful and scalable data platforms to search, observe, and secure your organization. Birmingham: Packt Publishing, 2022. ISBN 9781800569492.

GORMLEY, Clinton and TONG, Zachary. Elasticsearch: The Definitive Guide. Sebastopol: O'Reilly Media, 2015. ISBN 9781449358549.

MIELL, Ian and SAYERS, Aidan Hobson. Docker in Practice. 2nd Edition. Shelter Island: Manning Publications, 2019. ISBN 9781617294808.

SRIVASTAVA, Anurag and AZARMI, Bahaaldine. Learning Kibana 7: Build Powerful Elastic Dashboards with Kibana's Data Visualization Capabilities. 2nd Edition. Birmingham: Packt Publishing, 2019. ISBN 9781838556556.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2024/25

V Brně dne 9.2.2025

L. S.

Ing. Jiří Kříž, Ph.D. garant	prof. Ing. et Ing. Stanislav Škapa, Ph.D. děkan

Abstrakt

Tato bakalářská práce se zaměřuje na návrh a implementaci dashboardů pro tým společnosti Siemens, který se zabývá vývojem firmwarové komponenty MultiComm. V práci je analyzován současný stav vývoje a jsou identifikovány jeho hlavní problémy. Na základě těchto nedostatků bylo navrženo řešení zahrnující automatizovaný sběr, zpracování a vizualizaci dat. Výsledkem práce jsou dashboardy zobrazující data týkající se kvality vývoje komponenty MultiComm.

Klíčová slova

Elasticsearch, Azure DevOps, pipeline, dashboard, Kibana, Python, vizualizace dat, automatizace

Abstract

This bachelor's thesis focuses on the design and implementation of dashboards for a team at Siemens engaged in the development of the MultiComm firmware component. The thesis analyzes the current state of development and identifies the main issues. Based on these findings, a solution was proposed that includes automated data collection, processing, and visualization. The result of the work is a set of dashboards that display data related to the quality of the development of the MultiComm component.

Keywords

Elasticsearch, Azure DevOps, pipeline, dashboard, Kibana, Python, data visualization, automation

Bibliografická citace

DMITRIIEVICH, Daria. *Návrh a implementace dashboardů ve firemním prostředí* [online]. Brno, 2025 [cit. 2025-05-19]. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/168595. Bakalářská práce. Vysoké učení technické v Brně, Fakulta podnikatelská, Ústav informatiky. Vedoucí práce doc. András Rontó.

Čestné prohlášení

Prohlašuji, že předložená bakalářská práce je původní a zpracovala jsem ji samostatně. Prohlašuji, že citace použitých pramenů je úplná, že jsem ve své práci neporušila autorská práva (ve smyslu zákona č. 121/2000 Sb., o právu autorském a o právech souvisejících s právem autorským).

V Brně dne 19. 5. 2025

Daria Dmitriievich
autor

Poděkování

Ráda bych poděkovala panu Ing. Martinu Švédovi a panu Ing. Tomáši Floriánovi za možnost vypracovat tuto práci na zadané téma, za jejich odborné vedení a cenné poznatky, které mi poskytli v průběhu celé doby jejího zpracování. V neposlední řadě bych ráda poděkovala také svým blízkým za jejich podporu během celého studia.

OBSAH

ÚVOD **11**

CÍLE PRÁCE, METODY A POSTUPY ZPRACOVÁNÍ **12**

1 TEORETICKÁ VÝCHODISKA PRÁCE **13**

1.1 Firmware ve vestavěných systémech 13
1.1.1 Definice a charakteristika firmware 13
1.1.2 Použití firmwaru 13
1.1.3 Vývoj a nasazení firmwaru 14
1.2 Data, datové systémy a jejich struktury 14
1.2.1 Kvalitativní a kvantitativní data 14
1.2.2 Úvod do datových systémů 15
1.2.3 Typy datových systémů a jejich struktura 15
1.2.4 Dokumentově orientovaná NoSQL databáze Elasticsearch . . . 16
1.3 Role dashboardů a práce s daty 17
1.3.1 Příprava a zpracování dat 17
1.3.2 Vizualizace dat 17
1.4 Kontejnerizace a rozdíly oproti virtualizaci 17
1.4.1 Co je Docker a jak zajišťuje kontejnerizaci 18
1.4.2 Architektura Dockeru 19
1.5 Nasazování automatizace pomocí Azure DevOps 20
1.5.1 Problémy manuálního vývoje softwaru 20
1.5.2 Automatizovaný vývoj softwaru pomocí CI/CD pipeline 21
1.5.3 Využití pipeline pro automatizaci v Azure DevOps 21
1.6 APIs a jejich využití pro přístup k datům 22
1.6.1 Úvod do REST API 23
1.6.2 Elasticsearch REST API 23
1.6.3 Azure DevOps REST API 24

2 ANALÝZA SOUČASNÉHO STAVU **25**

2.1 Představení společnosti Siemens 25
2.2 Organizační struktura 25
2.3 Současné postupy monitorování 26
2.3.1 Analýza dat uložených v JFrog Artifactory 26

2.3.2 Získávání a vizualizace dat z běhových logů Azure DevOps . . 27
2.4 Nedostatky současného řešení 28
2.4.1 Proč není Azure DevOps dashboard dostačující pro požadavky projektu 29
2.5 Požadavky na nový systém 30

3 VLASTNÍ NÁVRH A IMPLEMENTACE ŘEŠENÍ 32
3.1 Hodnocení a výběr technologií 32
3.1.1 Porovnání Elasticsearch, Kibana a Splunk 32
3.1.2 Porovnání Grafana a Kibana 33
3.1.3 Porovnání Azure DevOps a GitLab 33
3.1.4 Porovnání Docker a Podman 34
3.2 Návrh architektury 34
3.3 Nasazení Docker kontejnerů 35
3.3.1 Vytvoření Docker kontejnerů 35
3.3.2 Použití Docker Compose 36
3.4 Vytvoření lokálně provozovaného agenta na linuxovém serveru 39
3.4.1 Nastavení Git a připojení repozitáře 39
3.4.2 Instalace a konfigurace lokálně provozovaného agenta 39
3.4.3 Postup práce Python skriptů při zpracování dat 40
3.4.4 Popis Python funkcí 41
3.4.5 Struktura Azure DevOps pipeline definované pomocí YAML souboru 47
3.5 Implementace dashboardu v Kibaně 52
3.5.1 Vizualizace stavu pipeline 52
3.5.2 Vizualizace výsledků Unit Testů 54
3.5.3 Vizualizace statické analýzy kódu (Lint) 55
3.5.4 Vizualizace Open Bugs 56
3.5.5 Vizualizace CI/CD testů 57
3.5.6 Vizualizace jitteru a latence 58
3.5.7 Přehled jednotlivých dashboardů a globální filtrování 60
3.6 Zhodnocení implementace 63
3.7 Návrhy na zlepšení 64

ZÁVĚR 65

LITERATURA 66

Seznam obrázků

1.1 Architektura Dockeru 20

3.1 Architektura systému pro automatizovaný sběr a zpracování dat v lokálním prostředí 35
3.2 Konfigurace Azure DevOps pipeline 49
3.3 Pipfile.lock s definovanými závislostmi 50
3.4 Zobrazení výsledků jednotlivých fází pipeline 53
3.5 Tabulkové zobrazení výsledků unit testů podle komponent a buildů . 54
3.6 Grafické zobrazení chyb, varování a informací ze statické analýzy kódu 55
3.7 Analýza řízení životního cyklu chyb 56
3.8 Vizualizace výstupů testovacích plánů 58
3.9 Grafické zobrazení latence a jitteru pro různé komunikační protokoly 59
3.10 Navigace a globální filtrování v dashboardu Kibana 60
3.11 Ukázka Multicomm Dashboardu 61
3.12 Ukázka Tests from CI/CD Dashboardu 62
3.13 Ukázka Jitter Dashboardu 63

ÚVOD

Pro zvýšení kvality softwaru je důležité mít během jeho vývoje přehled o průběhu a dosažených výsledcích. Ať už se jedná o vlastní projekt, nebo o projekt v rámci firmy, často vzniká potřeba rychle zjistit, jaké dopady měla konkrétní změna softwaru. Je potřeba analyzovat, zda nedošlo k novým problémům vyžadujícím zásah, nebo zda nová verze přináší zlepšení oproti té předchozí. Jedním z klíčových aspektů úspěšného vývoje je monitorování dat, která během něj vznikají, aby bylo možné situaci rychle porozumět a včas odhalit případné problémy.

V této oblasti nacházejí uplatnění dashboardy, které umožňují přehledně zobrazovat a analyzovat důležitá data na jednom místě. Dashboard by měl obsahovat jednu nebo více vizualizací zobrazujících klíčová data potřebná pro analýzu vývoje softwaru. Díky přehledným grafům a tabulkám obsaženým v dashboardu je možné rychleji reagovat na změny, lépe sledovat trendy a efektivněji plánovat další kroky vývoje.

Bakalářská práce analyzuje současné řešení a identifikuje nedostatky v týmu společnosti Siemens, který se zabývá vývojem firmwarové komponenty MultiComm. Na základě požadavků je následně představen návrh a implementace nového řešení pro monitorování klíčových parametrů prostřednictvím dashboardů. Cílem těchto dashboardů je zobrazovat data, která týmu pomáhají sledovat kvalitu vývoje dané komponenty.

CÍLE PRÁCE, METODY A POSTUPY ZPRACOVÁNÍ

Cílem bakalářské práce je navrhnout a implementovat dashboardy pro tým ve společnosti, které umožní vizualizaci klíčových metrik a dat spojených s vývojem firmwarové komponenty MultiComm. Výsledný nástroj má centralizovat přístup k těmto datům, usnadnit jejich analýzu a podpořit sledování trendů a změn. Dashboard má zjednodušit přístup k těmto datům, která jsou v současnosti uchovávána na různých místech a často neumožňují efektivní sledování trendů a změn. Tato práce je rozdělena do tří částí:

První část shrnuje teoretická východiska a technologie, které jsou důležité pro pochopení celého řešení. Jsou zde vysvětleny pojmy jako firmware, datové struktury, principy kontejnerizace pomocí Dockeru, automatizace prostřednictvím pipeline v Azure DevOps a využití REST API pro komunikaci mezi systémy.

Druhá část se věnuje analýze současného stavu a problémů, které má dashboard řešit. Jsou zde popsány nedostatky stávajícího způsobu ukládání a správy dat, chybějící centralizace, obtížnost sledování trendů a nutnost manuálního přístupu k několika nezávislým systémům. Na základě této analýzy jsou definovány požadavky na nový systém.

Třetí část se zabývá návrhem a implementací vlastního řešení. Nejprve je provedeno hodnocení a výběr vhodných technologií, poté následuje návrh architektury systému. Následně je detailně popsán postup nasazení Docker kontejnerů, konfigurace pipeline v Azure DevOps, zpracování dat pomocí skriptů v jazyce Python a finální implementace dashboardů v prostředí Kibana. Součástí jsou rovněž ukázky výsledných vizualizací, zhodnocení implementace a návrhy na možná budoucí vylepšení celého řešení.

1 TEORETICKÁ VÝCHODISKA PRÁCE

1.1 Firmware ve vestavěných systémech

V moderním světě technologií hraje firmware klíčovou roli v každodenním fungování zařízení. S firmwarem se lze setkat od domácí elektroniky až po průmyslové řídicí systémy. Firmware představuje neviditelný most mezi hardwarovými komponentami a softwarem, který umožňuje jejich efektivní a spolehlivou spolupráci.

1.1.1 Definice a charakteristika firmware

Firmware je typ softwaru, který je úzce navázán na určité hardwarové zařízení, nejčastěji na specifický vestavěný systém [2]. Nemůže tedy fungovat očekávaným způsobem, pokud by byl nainstalován na jiné zařízení, než pro které byl navržen nebo optimalizován.

Obecně se instaluje do trvalé paměti zařízení (*non-volatile*), jako je ROM[1], což zajišťuje jeho trvalost. Tuto instalaci provádí výrobce zařízení během výroby [2]. V některých systémech je však firmware uložen v přepisovatelné paměti, což umožňuje jeho aktualizaci například z důvodu implementace bezpečnostní záplaty nebo přidání nové funkce.

Na rozdíl od běžně známého softwaru, který je navržen pro uživatelskou interakci, je role firmwaru skryta pod povrchem softwarového prostředí. Slouží jako komunikační rozhraní mezi vyššími vrstvami softwaru, například operačním systémem Windows, a hardwarem daného zařízení.

1.1.2 Použití firmwaru

Firmware lze nalézt téměř v každém počítačovém zařízení, například:

- v počítačích ve formě BIOSu[2] nebo UEFI[3], ale také na grafické kartě,
- v počítačovém příslušenství, jako jsou tiskárny, monitory nebo kamery,
- v síťových zařízeních, jako jsou firewally, směrovače a přepínače,
- v řídicích systémech provozní technologie, například v PLC[4] od firmy Siemens.

[1] ROM (Read-Only Memory)

[2] BIOS (Basic Input/Output System) - forma firmwaru, která instruuje operační systém počítače o správném způsobu fungování.

[3] UEFI (Unified Extensible Firmware Interface) - specifikace pro softwarový program, který zajišťuje propojení firmwaru počítače s jeho operačním systémem.

[4] PLC (Programmable Logic Controller) - průmyslový počítač, který je odolný a přizpůsobený pro řízení výrobních procesů, jako jsou montážní linky.

Firmware u těchto zařízení zajišťuje spuštění a inicializaci hardwarových komponent a také jejich správnou funkci [2].

1.1.3 Vývoj a nasazení firmwaru

Vývoj, ladění a nasazování firmwaru je na rozdíl od softwaru mnohem náročnější. Systémy, na kterých firmware běží, jsou totiž navrženy pro určitý účel. Pro vývojáře to znamená omezené zdroje, jako je paměť, výkon či požadavky na energetickou efektivitu. Vývojáři musí navíc zohlednit i výslednou výrobní cenu celého zařízení, která bývá při masové produkci klíčovým faktorem. Vzhledem k omezenému přístupu k běžným ladicím nástrojům a prostředkům může být identifikace a odstraňování chyb časově velmi náročné [17].

Firmware proto musí být co nejvíce optimalizován pro daný hardware ještě před zahájením výroby. Zároveň by měl poskytovat určitou míru flexibility pro případné aktualizace. Vzhledem k možným budoucím zranitelnostem a požadavkům na nové funkce je důležité, aby firmware podporoval bezpečné a spolehlivé provedení aktualizace, a to i v provozním prostředí.

1.2 Data, datové systémy a jejich struktury

Během vývoje softwaru vzniká velké množství logů či výstupů z ladicího procesu, které mohou vývojářům pomoci při hledání chyb. Díky těmto datům mohou vývojáři také lépe sledovat průběh projektu. Z tohoto důvodu je nutné data ukládat, ale zároveň je důležité rozlišovat mezi důležitými a méně podstatnými daty. Ne všechna data jsou totiž pro analýzu užitečná.

1.2.1 Kvalitativní a kvantitativní data

Kvalitativní údaje jsou data, která nelze snadno vyjádřit číselně, a tudíž nad nimi nelze provádět aritmetické operace. Tato data se primárně vztahují k vlastnostem objektů a slouží k jejich vzájemnému rozlišení. Zaměřují se proto víc na samotný vzhled objektů, přičemž k jejich odlišení slouží vlastnosti jako barva nebo název [15]. Do této kategorie lze zahrnout například i metadata souborů, jako je aktuální verze nebo identifikátor buildu[5], které umožňují jednoznačnou identifikaci dat.

Naopak kvantitativní údaje představují data, která mohou nebo musí být sečtena, aby poskytla požadované informace. Jedná se hlavně o čísla, počty nebo výsledky různých funkcí, také sem patří číselné vlastnosti objektů. S těmito daty je následná vizualizace a vyhodnocení výrazně jednodušší [15]. Příkladem je počet úspěšných a neúspěšných testů nebo jiná měřitelná data, často označovaná termínem metriky.

[5] Build - proces sestavení kódu.

V praxi je potřeba tyto údaje spolu kombinovat, protože jenom společně umožňují přesně identifikovat silné stránky nebo nedostatky objektu. Kvalitativní data ukazují, kolik unikátních objektů existuje pro analýzu a kvantitativní data určují frekvenci nebo počet existujících problémů [15].

1.2.2 Úvod do datových systémů

Data je nutné někam uchovávat. Právě k tomu slouží systémy pro správu databází, které navíc poskytují správu a vyhledávání dat pomocí softwaru DBMS[6]. Tento software se skládá z úložné a správní části. Úložná část zahrnuje samotná data a jejich popis. Správní část umožňuje provádět dotazy, manipulovat s daty a řídit přístup. Pro ukládání dat se nejčastěji používají dva typy databází: relační databáze SQL[7] a nerelační databáze NoSQL[8] [11].

1.2.3 Typy datových systémů a jejich struktura

Relační databáze ukládají data do tabulek s pevně definovanou strukturou a využívají SQL jako dotazovací jazyk. Vyznačují se vysokou konzistencí dat a jsou vhodné například pro ERP[9] systémy, bankovnictví a e-commerce platformy [11].

NoSQL databáze nejsou založeny na relačním modelu. Používají flexibilní datové struktury, například klíč-hodnota, dokumenty, sloupce nebo grafy. Podporují horizontální škálování, mají volnější požadavky na schéma a často preferují výkon před konzistencí. Jsou vhodné pro webové aplikace, sociální sítě, Big Data[10] [11].

Jak již bylo uvedeno, NoSQL databáze používají flexibilní datové struktury a existují čtyři hlavní kategorie NoSQL databází [11]:

- Klíč-hodnota databáze (Key-Value Stores)
- Sloupcově orientované databáze (Column-Family Stores)
- Dokumentově orientované databáze (Document Stores)
- Grafové databáze (Graph Databases)

[6] DBMS (Database Management System)
[7] SQL (Structured query language)
[8] NoSQL (Not Only SQL)
[9] ERP (Enterprise Resource Planning) - informační systém, který integruje různé podnikové procesy a funkce.
[10] Big Data - označuje extrémně velké objemy dat.

1.2.4 Dokumentově orientovaná NoSQL databáze Elasticsearch

Vzhledem k tomu, že je v této bakalářské práci použita dokumentově orientovaná databáze Elasticsearch, budou v následující části vysvětleny klíčové pojmy právě tohoto typu databází [1]:

- **Elasticsearch** je distribuovaný vyhledávací a analytický nástroj, který umožňuje efektivní vyhledávání a agregaci velkých objemů dat. Slouží jako NoSQL databáze určená pro ukládání JSON dokumentů, podporuje fulltextové vyhledávání a výpočty agregací. Nabízí REST API[11] ve formátu JSON[12], což usnadňuje integraci s různými technologiemi.

- **Uzel** představuje běžící instanci Elasticsearch. Každý uzel může uchovávat data a zpracovávat požadavky. Jeden z uzlů v rámci clusteru je zvolen jako hlavní a zajišťuje správu změn, například přidání nového indexu nebo dalšího uzlu.

- **Cluster** v Elasticsearch je skupina jednoho nebo více uzlů, které spolupracují na ukládání a vyhledávání dat. Každý cluster má jedinečné jméno a všechny jeho uzly sdílejí uložená data i výpočetní zátěž.

- **Dokument** v systému Elasticsearch je JSON objekt uložený pod jedinečným identifikátorem (ID). Obsahuje dvojice klíč–hodnota, kde hodnoty mohou být například texty, čísla, logické hodnoty, geolokace nebo vnořené objekty.

- **Index** v Elasticsearch lze chápat jako obdobu databáze v relačních databázích. Slouží ke sdružování souvisejících dokumentů.

- **Indexování** znamená uložení dokumentu do indexu, podobně jako bychom v relačních databázích prováděli operaci INSERT[13]. Pokud již dokument se stejným identifikátorem existuje, nahradí se novým dokumentem.

Dalším klíčovým konceptem Elasticsearch je použití **inverzního indexu**, což je základní datová struktura umožňující rychlé vyhledávání v textu. Inverzní index funguje tak, že namísto lineárního prohledávání dokumentů vytváří mapu slov na seznamy dokumentů, ve kterých se dané slovo vyskytuje. Díky tomu je možné velmi efektivně vyhledat všechny dokumenty obsahující určité slovo nebo frázi [6].

[11] REST API (Representational State Transfer Application Programming Interface)
[12] JSON (JavaScript Object Notation) - formát pro strukturované ukládání dat.
[13] INSERT - příkaz v jazyce SQL, který slouží k vložení nového záznamu do tabulky v databázi.

1.3 Role dashboardů a práce s daty

Téměř každé digitální zařízení odesílá a ukládá data. Z těchto uložených dat lze získat informace, které mohou být dále využity k analýze a vizualizaci. Před samotnou analýzou je však nutné porozumět údajům, které se zpracovávají. Bez základní znalosti struktury a obsahu těchto dat může být provádění optimalizačních kroků nebo přijímání zásadních rozhodnutí zavádějící. Proto je potřeba správně informovat odpovědné osoby, které budou s daty pracovat. Také je vhodné vytvořit snadno pochopitelnou vizualizaci pro co nejlepší získání přehledu o zkoumané problematice [16].

1.3.1 Příprava a zpracování dat

Příprava dat k analýze není zcela jednoduchá. Hlavními problémy bývají jejich komplexnost, decentralizace a objem. Na zjištění příčiny určitého jevu je často potřeba sledovat všechny jeho části nebo hledat korelace mezi různými systémy, které situaci ovlivňují. Stavy, logy nebo výsledky z různých systémů, které využívají odlišný formát, je potřeba správně uložit a také zajistit jejich rychlé prohledávání [16].

Naštěstí je právě k tomu možné využít například databázi Elasticsearch, která je popsaná v předchozí sekci 1.2.4.

1.3.2 Vizualizace dat

Dashboardy tvoří vizualizační vrstvu datové analýzy a právě s touto vrstvou se během analýzy setkává nejvíce uživatelů. Skládá se z tabulek nebo grafů a soustředí se na poskytnutí jasného a srozumitelného přehledu o klíčových ukazatelích výkonnosti (KPI[14]) [16].

Aby byly dashboardy srozumitelné, musí být rychle čitelné pro každého uživatele a musí obsahovat pouze nezbytná data potřebná k dané analýze. Například použitím červeného nebo zeleného zabarvení pro zvýraznění důležitých trendů. Dále musí být propojeny s jinými daty, která zobrazují stejný jev, čímž se zvýší přehled o řešené problematice. Dashboardy by měly být snadno rozšiřitelné za účelem analýzy nových parametrů a zároveň dostupné i mimo aplikace, ve kterých jsou provozovány [16]. Nejdůležitější vlastností dashboardů je však schopnost analýzy dat v reálném čase. Tato schopnost je nezbytná pro včasnou identifikaci problému a jeho rychlé vyřešení.

1.4 Kontejnerizace a rozdíly oproti virtualizaci

Virtualizace je proces, který umožňuje jednomu fyzickému zařízení sdílet své výpočetní, síťové a úložné prostředky mezi více virtuálními stroji. Pomocí hypervizorového

[14]KPI (Key Performance Indicator)

softwaru lze na jednom fyzickém hardwaru spouštět více operačních systémů nezávisle na sobě [10].

Kontejnerizace spočívá v zabalení aplikace spolu se všemi jejími závislostmi do izolovaného, lehkého kontejneru. Kontejnery sdílejí jádro operačního systému, jsou přenosné, flexibilní a efektivní z hlediska provozu [10].

Vzhledem k tomu, že Docker je v současnosti nejpoužívanější a nejrozšířenější platformou pro práci s kontejnery, je vhodné zaměřit následující porovnání právě na něj a na virtuální stroje.

Hlavní rozdíly mezi virtuálními stroji a kontejnery Dockeru jsou následující [13]:

- **Zaměření**:
  - Kontejnery Dockeru jsou orientovány na běh samostatné aplikace.
  - Virtuální stroje jsou zaměřeny na běh celého operačního systému.
- **Operační systém**:
  - Docker kontejnery sdílejí jádro hostitelského systému.
  - Každý virtuální stroj má vlastní operační systém spravovaný hypervizorem.
- **Procesy**:
  - Docker kontejnery jsou navrženy pro běh jednoho hlavního procesu.
  - Virtuální stroje typicky spravují více procesů současně.

1.4.1 Co je Docker a jak zajišťuje kontejnerizaci

Pro pochopení principů kontejnerizace je nutné se seznámit s nástrojem Docker, jeho základními pojmy a architekturou [13]:

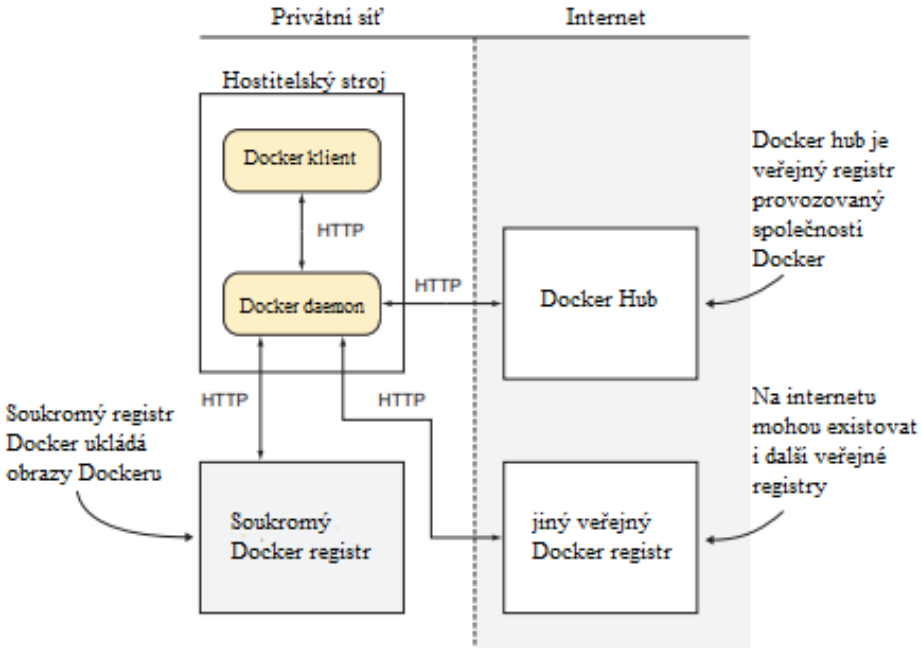
- **Docker** je nástroj, který zjednodušuje nasazení aplikací pomocí kontejnerů. Tyto kontejnery sdružují aplikaci spolu s jejími závislostmi do lehké, přenosné jednotky. Díky tomu je zajištěn konzistentní běh aplikace napříč různými prostředími, čímž se snižuje složitost správy konfigurací a režie spojená s virtuálními stroji.
- **Obraz (image)** představuje šablonu obsahující vše potřebné k vytvoření a spuštění kontejneru, například aplikační soubory, knihovny a nastavení.

- **Kontejner** je běžící instance obrazu. Z jednoho obrazu lze spustit více kontejnerů současně.
- **Registr** slouží jako centrální úložiště pro Docker obrazy, ze kterého je možné obrazy nahrávat a stahovat.
- **Dockerfile** je textový soubor obsahující sekvenci instrukcí, které se provádějí ve stanoveném pořadí a určují, jak má být Docker obraz vytvořen a jak má vypadat výsledný obraz.
- **Docker Compose** je nástroj, který umožňuje definovat a spouštět více kontejnerů jako jednu aplikaci. Konfigurace aplikace se popisuje v souboru **docker-compose.yml**, kde jsou uvedeny všechny potřebné služby. Místo ručního spouštění jednotlivých kontejnerů umožňuje Docker Compose jejich společné spuštění jediným příkazem.

1.4.2 Architektura Dockeru

Docker je postaven na architektuře klient-server a skládá se ze dvou hlavních částí: Docker klienta a Docker daemonu[15]. Klient odesílá příkazy prostřednictvím REST API, které daemon přijímá a zpracovává – spravuje kontejnery, obrazy, sítě i úložiště. Daemon zároveň komunikuje s Docker registry, odkud může obrazy stahovat nebo je nahrávat. Docker podporuje veřejné i privátní registry, přičemž nejznámějším veřejným registrem je Docker Hub, zatímco organizace si mohou provozovat vlastní privátní registry pro interní použití [13].

[15] Daemon - počítačový program běžící na pozadí operačního systému a vykonává specifické úkoly.



Obrázek 1.1: Architektura Dockeru

Obrázek znázorňuje, jak Docker klient běžící na hostitelském stroji komunikuje s Docker daemonem. Docker daemon následně pracuje s Docker obrazy, buď v registru umístěném ve stejné privátní síti jako host nebo na veřejných cloudových registrech. Všechna komunikace mezi klientem a daemonem, stejně jako komunikace mezi daemonem a registry, probíhá prostřednictvím HTTP[16] požadavků.

1.5 Nasazování automatizace pomocí Azure DevOps

Každé manuální nasazení nebo vydání jakéhokoli softwaru s sebou nese různé komplikace. Prostředí, ve kterém se aplikace instaluje a spouští, je pokaždé jiné. Navíc se střídají vývojáři, takže ten, kdo aplikaci původně implementoval, již nemusí být k dispozici. V důsledku těchto i dalších problémů se proces vývoje postupně začal zkvalitňovat a automatizovat.

1.5.1 Problémy manuálního vývoje softwaru

Vývoj a nasazení většiny aplikací vyžaduje značné úsilí a funkčnost několika samostatných částí, za jejichž implementaci je zodpovědný celý tým nebo jeden člen. Během procesu vývoje je potřeba vykonávat rozhodnutí, což může vést k lidským chybám. Dokonce i drobné rozdíly při skládání těchto částí mohou vést k odlišným výsledkům při opětovném nasazování [8].

[16]HTTP (Hypertext Transfer Protocol)

Automatizace vývoje softwaru je nezbytná pro zajištění efektivity, konzistence a škálovatelnosti. Zavedením dvou různých prostředí, jedno pro testování a druhé pro produkci, se snižuje riziko lidských chyb. Rozšiřování a ladění probíhá v testovacím prostředí, kde nejsou ovlivňováni koncoví uživatelé. Teprve finální stabilní verze aplikace je nasazena do produkčního prostředí [8].

1.5.2 Automatizovaný vývoj softwaru pomocí CI/CD pipeline

Zavedení automatizace při vývoji softwaru lze realizovat různými způsoby, ale jedním z nejpoužívanějších je využití principu CI/CD pipeline[17], tedy *Continuous Integration / Continuous Deployment*.

Kontinuální Integrace označuje princip nepřetržitého testování nových verzí kódu. Týmy, které tuto metodu při vývoji svého softwaru využívají, mohou tvrdit, že jejich software je vždy po nahrání nové verze funkční, pokud všechny přidělené testy v CI pipeline proběhnou bez chyb. V případě, že nějaký test selže, tým je okamžitě upozorněn a může chybu opravit. Tímto jsou chyby zachyceny ještě před procesem vydání finální verze softwaru, kdy je náprava chyb mnohem rychlejší a levnější [8].

Kontinuální Nasazení zastřešuje automatizaci posledního kroku při nasazování softwaru do produkčního prostředí. Po úspěšném dokončení všech přiřazených testů se aplikace aktualizuje na novou verzi. Samozřejmě návrh těchto testů není vůbec jednoduché. Z tohoto důvodu bývá tento přístup kombinován s dalšími metodami nasazování nového softwaru, které pomáhají minimalizovat rizika spojená s instalací nové verze [8].

1.5.3 Využití pipeline pro automatizaci v Azure DevOps

Azure Pipelines je součástí platformy Azure DevOps od společnosti Microsoft. Slouží k automatizaci sestavování, testování a nasazování různých projektů, od softwarových aplikací až po infrastrukturu. Využívá principy kontinuální integrace, testování a nasazování a umožňuje spolehlivé a opakovatelné nasazování do libovolného prostředí.

Pro použití Azure Pipelines musí být zdrojový kód uložen v systému pro správu verzí (*Version Control System*). Podporovány jsou například Azure Repos Git[18], Git[19], GitHub[20] nebo TFVC[21] [12].

[17] Pipeline - automatizovaný proces, který zahrnuje sestavení, testování a nasazení aplikace při každé změně v kódu.
[18] Azure Repos Git - distribuovaný systém pro správu verzí integrovaný v Azure DevOps.
[19] Git - rozšířený distribuovaný systém pro správu verzí.
[20] GitHub - platforma pro hostování Git repozitářů s podporou spolupráce, pull requestů a CI/CD.
[21] TFVC (Team Foundation Version Control) - systém pro správu verzí od Microsoftu.

Pipeline lze definovat pomocí YAML[22] souboru, ve kterém jsou jednotlivé kroky automatizace přesně specifikovány. V následující části bude popsána struktura YAML souboru a základní prvky, které tento soubor obvykle obsahuje [12]:

- **Úloha (job)** představuje sadu kroků, které se vykonávají na jednom agentovi. Každá pipeline obsahuje alespoň jednu úlohu. Jednotlivé úlohy mohou běžet paralelně nebo na sobě mohou záviset podle definovaného pořadí.
- **Pool** je skupina agentů, která usnadňuje správu a organizaci prostředí pro spouštění pipeline. Každý agent je registrován do jednoho poolu. Při konfiguraci pipeline je potřeba určit, ve kterém poolu bude pipeline spouštěna.
- **Agent** je výpočetní prostředek, například server nebo virtuální stroj, s nainstalovaným softwarem, který umožňuje spouštění jednotlivých úloh v rámci pipeline. Pro sestavení kódu nebo nasazení softwaru pomocí Azure Pipelines je vyžadován alespoň jeden agent. Každý agent je schopen v daném okamžiku vykonávat právě jednu úlohu. Azure Pipelines nabízí různé typy agentů, například agenty hostované společností Microsoft (*Microsoft-hosted agents*) nebo agenty spravované samotným uživatelem (*Self-hosted agents*).
- **Kroky (steps)** tvoří jednotlivé operace, které jsou vykonávány postupně a dohromady tvoří obsah jedné úlohy. Každý krok představuje akci, například spuštění skriptu nebo úkolu.
- **Úkol (task)** je předdefinovaná akce, která vykonává úlohu v rámci pipeline. Příkladem je **JFrogGenericArtifacts@1**, což je úkol určený ke stahování artefaktů z platformy JFrog Artifactory. Úkoly mohou být poskytovány přímo společností Microsoft, dále třetími stranami, jako je například JFrog, nebo si je vývojáři mohou vytvořit sami.
- **Skript** slouží ke spuštění vlastních příkazů. Na operačních systémech Windows se příkazy vykonávají prostřednictvím **cmd.exe**, zatímco na ostatních platformách, jako je Linux, se používá **shell bash**.

1.6 APIs a jejich využití pro přístup k datům

V dnešní době je většina internetových aplikací řízena pomocí API, díky kterým je uživatelská zkušenost mnohem interaktivnější a rychlejší. Namísto toho, aby se

[22]YAML (YAML Ain't Markup Language) - přehledný formát pro zápis konfiguračních souborů.

webové stránky načetly okamžitě, se při otevření stáhnou pouze důležitá data. Zbytek stránky se pak aktualizuje na pozadí, zatímco uživatel interaguje s obsahem [5].

1.6.1 Úvod do REST API

REST API je jedním z typů aplikačních rozhraní, který běží na webové části aplikace a poskytuje standardizovaný přístup k datům dostupným na serveru. Klientské zařízení toto serverové rozhraní využívá ke čtení, odeslání, aktualizaci nebo mazání dat prostřednictvím metod GET, POST, PUT a DELETE [5].

Tyto HTTP metody REST API jsou dostupné prostřednictvím URL[23] adresy. Po odeslání požadavku na server se nevrací klasická webová stránka, ale proběhne zpracování požadavku na práci s daty. Skriptovací jazyk, pomocí něhož je API implementováno, tento požadavek zpracuje a odešle zpět odpověď – ve formě statusového kódu *200 OK* nebo *404 Not Found*, hlavičky HTTP a případně i strukturovaných dat specifických pro danou operaci [5].

Vzhledem k tomu, že REST API přijímá požadavky prostřednictvím protokolu HTTP, musí být zdroje, strukturovaná data, odeslány ve formátu JSON nebo XML[24]. Tyto zdroje mohou reprezentovat libovolné informace či data uložená v databázi [5].

1.6.2 Elasticsearch REST API

Jako praktický příklad využití REST API lze uvést jednoduchý požadavek na zjištění počtu dokumentů uložených v databázi Elasticsearch. Pomocí HTTP metody GET a nástroje **curl**[25] lze snadno ověřit, kolik dokumentů se nachází ve zvoleném indexu:

```
curl -X GET "http://localhost:9200/quality_lint/_count"
```

Tento příkaz odešle požadavek na získání počtu dokumentů uložených v indexu s názvem **quality_lint**. Dotaz je směrován na lokální instanci Elasticsearch dostupnou na adrese **localhost:9200**.

Odpověď serveru je ve formátu JSON a obsahuje informaci o počtu nalezených dokumentů, například:

[23] URL (Uniform Resource Locator)
[24] XML (Extensible Markup Language)
[25] curl - příkazový nástroj pro přenos dat přes různé protokoly.

```
{
 "count":12,
 "_shards":
  {
   "total":1,
   "successful":1,
   "skipped":0,
   "failed":0
  }
}
```

Klíčovým prvkem odpovědi je hodnota **count**, která udává celkový počet dokumentů uložených v daném indexu. Tento dotaz lze využít například ke kontrole, zda daný index obsahuje nějaká data.

1.6.3 Azure DevOps REST API

Azure DevOps REST API lze využít například k přidávání tagů[26], jako jsou *Release* nebo *Nightly Build*, k jednotlivým buildům. Následující ukázka využívá nástroj **curl** pro přidání tagu **release-v3.0** k buildu s **id 34246**.

```
curl -X PUT
  -u :<PAT_TOKEN>
  https://dev.azure.com/my-organisation/my-project/_apis/build/
  builds/34246/tags/release-v3.0?api-version=7.2-preview.3
```

Autentizace je zajištěna pomocí PAT[27], který má oprávnění k zápisu tagů k jednotlivým buildům. Token lze vygenerovat přímo v uživatelském rozhraní Azure DevOps.

[26] Tag a tagování - slouží k lepší správě a identifikaci buildů.
[27] PAT (Personal Access Token)

2 ANALÝZA SOUČASNÉHO STAVU

Tato kapitola se zabývá analýzou současného stavu ve společnosti Siemens, konkrétně v souvislosti se sledováním a vyhodnocováním klíčových metrik vývoje firmwarové komponenty MultiComm. V současné době nejsou metriky centralizovány do jednoho systému, což vede k řadě problémů při jejich správě a analýze. Tento nedostatek je hlavním motivem pro návrh a implementaci dashboardů, které by měly tyto problémy řešit.

2.1 Představení společnosti Siemens

Siemens je globální technologická společnost zaměřená na digitální transformaci průmyslu, infrastruktury, mobility a zdravotnictví. Její hlavní poslání spočívá v podpoře klientů při odhalování jejich potenciálu a efektivní implementaci inovací, které zvyšují produktivitu a konkurenceschopnost v digitálním věku.

Společnost Siemens má více než 89 zastoupení po celém světě a působí ve více než 200 zemích. V České republice disponuje třemi hlavními vývojovými centry, a to v Praze, Plzni a Brně. Zaměstnává zde více než 450 vývojářů a techniků.

Siemens poskytuje komplexní služby od poradenství přes návrh a vývoj prototypů až po realizaci a implementaci finálních řešení. Zaměřuje se na digitální transformaci v široké škále odvětví, jako je průmyslová výroba, energetika, zdravotnictví, automobilový průmysl, mobilita a logistika. Díky hlubokým znalostem a zkušenostem s informačními (IT) a provozními technologiemi (OT) nabízí Siemens řešení, která umožňují realizaci digitální budoucnosti.

2.2 Organizační struktura

Vývoj probíhá v rámci oddělení Siemens Foundational Technologies, které poskytuje služby výzkumu a vývoje různým obchodním jednotkám společnosti Siemens. Každý tým se skládá z několika klíčových rolí:

- **Product Owner (produktový vlastník)**: Tato role je zodpovědná za definování požadavků na produkt, stanovování priorit a komunikaci se zúčastněnými stranami. Product Owner zajišťuje, že tým vyvíjí produkt, který odpovídá potřebám zákazníka.

- **Project Leader (vedoucí projektu)**: Řídí projekt od jeho zahájení až po dokončení. Koordinuje práci týmu, sleduje termíny a dohlíží na dodání projektu včas a v požadované kvalitě.
- **Developer (vývojář)**: Zodpovídá za technickou realizaci projektu. Píše kód, implementuje požadované funkce a řeší technické problémy, které se mohou během vývoje objevit.
- **Tester**: Ověřuje, zda firmware funguje správně a odpovídá stanoveným požadavkům. Zaměřuje se na odhalování chyb a zajištění celkové kvality produktu.

2.3 Současné postupy monitorování

V současné době v týmu Siemens Foundational Technologies neexistuje centralizovaný systém pro monitorování a vizualizaci klíčových metrik firmwarové komponenty MultiComm. Výsledky testů a další důležité údaje jsou uchovávány na různých místech a nejsou jednotně zpracovávány, což ztěžuje jejich efektivní analýzu i sledování.

Metriky jsou často uloženy v artefaktech vzniklých po dokončení pipeline v systému JFrog Artifactory, a to ve formátu JSON nebo jako logovací soubory. V některých případech se metriky vůbec neukládají a jsou dostupné pouze prostřednictvím logů z pipeline.

Tento neefektivní způsob ukládání a zpracování dat komplikuje nejen monitorování kvality firmwaru, ale také znemožňuje sledování trendů. Navíc chybí možnost filtrování metrik podle specifických kritérií, což dále omezuje schopnost rychle identifikovat problémy a sledovat klíčové ukazatele v reálném čase.

2.3.1 Analýza dat uložených v JFrog Artifactory

Některá data z buildu jsou ukládána do JFrog Artifactory. Je to univerzální repozitář umožňující správu artefaktů[1] v průběhu celého životního cyklu vývoje softwaru [9].

Kvůli absenci přímé vizualizace je nutné ručně vstupovat do rozhraní JFrog Artifactory, vyhledat ZIP[2] soubor, ve kterém jsou uloženy výstupy vytvořené během buildu, stáhnout jej a nalézt uvnitř požadovaný log nebo JSON soubor. Tento soubor je pak potřeba analyzovat nebo porovnat s jiným. Celý proces je manuální, časově náročný a málo efektivní.

Například v případě souboru kontroly kvality kódu (lint), který obsahuje seznam varování, chyb a dalších informací, může být soubor velmi rozsáhlý. Z tohoto dů-

[1] Artefakt – výstup build procesu, například binární soubor, knihovna, archiv, log nebo report.
[2] ZIP - formát pro kompresi a archivaci souborů do jednoho souboru.

vodu je často nutné vytvořit skript, který spočítá celkový počet varování, chyb a informačních hlášení. Samotný počet těchto položek však nestačí.

Je potřeba tato data ukládat na místo, kde se bude uchovávat historie všech lint souborů, a následně je vizualizovat. To umožní porovnávat výsledky mezi jednotlivými buildy a sledovat zlepšení nebo zhoršení kvality kódu.

Například:

- Pokud se na grafu objeví výrazně vyšší počet chyb, členové týmu si toho budou moci rychle všimnout a začít řešit problém.
- Pokud lint kontrola z nějakého důvodu selže a nová data se na grafu neobjeví, tým na to bude upozorněn absencí hodnot a může vyšetřit příčinu.

Kromě lint souborů se do JFrog Artifactory ukládají také soubory s výsledky měření síťových charakteristik, jako je jitter, tedy odchylky od předpokládané hodnoty periodicity zpráv. Bez grafického znázornění je obtížné rychle pochopit, jak jsou hodnoty jitteru nebo latence rozloženy a identifikovat případné odchylky nebo změny v kvalitě přenosu mezi jednotlivými buildy.

Proto samotné ukládání těchto dat nestačí – je nezbytné je vizualizovat. Díky grafickému zobrazení by tým mohl:

- Okamžitě odhalit anomálie, například neobvykle vysoký jitter, nebo vysokou latenci po příslušném buildu.
- Porovnávat výsledky mezi jednotlivými buildy a tím vyhodnocovat dopad změn v konfiguraci nebo softwarových úprav.

2.3.2 Získávání a vizualizace dat z běhových logů Azure DevOps

Po dokončení každého buildu jsou na Azure DevOps dostupné výstupní logy, které lze získat prostřednictvím API, například:

```
https://dev.azure.com/{organization}/{project}/_apis/build/builds/
125/logs/14
```

Tyto logy však nejsou trvale uchovávány, zůstávají dostupné pouze po omezenou dobu, dokud není daný build automaticky odstraněn.

Ve výstupních logech se často nacházejí textová data, která jsou pro tým cenná z hlediska vizualizace a vyhodnocování, například výsledky unit testů, rozdělené podle jednotlivých komponent. V současnosti však chybí centrální úložiště, kam by se tyto

informace systematicky ukládaly. Neexistuje automatizovaný proces pro jejich zpracování a vizualizaci.

Navrhované řešení spočívá ve vytvoření nástroje, který po každém buildu automaticky stáhne příslušné logy, extrahuje z nich požadované informace, například počet úspěšných a neúspěšných testů pro každou komponentu a uloží je do databáze. Na základě těchto dat bude následně možné vytvářet přehledné vizualizace, které budou:

- zobrazovat přehled úspěšných a neúspěšných testů pro každou komponentu.
- obsahovat identifikátor buildu (build number), aby bylo zřejmé, ze kterého buildu výsledky pocházejí.

Zavedení procesu pro automatizované získávání, zpracování a ukládání dat s následnou vizualizací by týmu přineslo řadu přínosů:

- Tým by získal okamžitý přístup k analytickým výstupům bez nutnosti manuální práce s logy nebo přihlašování do Azure DevOps.
- Grafické zobrazení výsledků umožní sledovat vývoj kvality v čase, například jestli se stabilita konkrétní komponenty zlepšuje, nebo naopak zhoršuje.

2.4 Nedostatky současného řešení

Hlavním nedostatkem současného řešení je skutečnost, že všechna data jsou uložena v různých systémech. Logy a JSON soubory jsou uloženy v JFrog Artifactory, zatímco logy z buildů jsou přístupné pouze z prostředí Azure DevOps.

V současnosti není k dispozici žádný nástroj, který by umožnil sledovat a ukládat všechna důležitá data na jednom centralizovaném místě v reálném čase. Vývojáři tak musí přistupovat k několika různým systémům, manuálně stahovat soubory nebo vyhledávat potřebné informace v logech z Azure DevOps, což způsobuje zpoždění při odhalování chyb a zpomaluje celý vývojový proces.

Dalším nedostatkem je samotný formát uložených dat. Logy často obsahují stovky až tisíce řádků, což ztěžuje jejich ruční procházení a analýzu. JSON soubory jsou vhodné pro zpracování pomocí skriptů a nástrojů, nicméně pro uživatele bez předchozích zkušeností bývají často nepřehledné a obtížně čitelné. Ponechat tyto soubory bez vizualizace tedy není vhodné řešení.

Navíc chybí automatizovaný proces pro zpracování těchto datových souborů a jejich strukturované ukládání do jedné databáze. Data, která by zpřístupnila všechny relevantní informace pro hodnocení kvality softwaru, včetně výsledků testů, síťových

metrik nebo výstupů z lint nástrojů, nejsou v současné podobě možné jednoduše vizualizovat.

Neexistuje efektivní způsob porovnávání dat v čase ani mezi různými verzemi firmwaru. Izolovanost jednotlivých systémů komplikuje sledování trendů a brání včasnému vyhodnocení dopadů změn. Není snadné zjistit, zda nová verze firmwaru přinesla zlepšení, nebo naopak způsobila nové problémy.

Stávající řešení rovněž neumožňuje filtrovat data podle specifických parametrů, například čísla buildu nebo přiděleného tagu.

2.4.1 Proč není Azure DevOps dashboard dostačující pro požadavky projektu

Azure DevOps obsahuje možnost tvoření dashboardů, avšak toto řešení není dostatečně flexibilní pro potřeby vizualizace komplexních metrik požadovaných v tomto projektu.

Primárním omezením je nemožnost zobrazení specifických dat, jako jsou například hodnoty jitteru nebo výsledky statistické analýzy zdrojového kódu (lint). Zdrojem těchto dat jsou různé typy souborů – textové logy obsahující stovky až tisíce řádků a JSON soubory zaměřené na cílené metriky. Azure DevOps dashboard však není navržen pro efektivní zpracování ani vizualizaci takto strukturovaných dat.

Dalším zásadním omezením je absence podpory pro komplexní práci s historickými daty. K dispozici jsou pouze předdefinované nástroje s omezenými možnostmi přizpůsobení, které umožňují zobrazit stav aktuálního buildu nebo několika posledních buildů. Neumožňují však podrobnější analýzu dlouhodobých trendů.

Není možné přepínat mezi daty podle build tagu, čísla buildu nebo jiných parametrů, což výrazně omezuje možnosti filtrování a porovnávání dat mezi jednotlivými verzemi.

Výsledky testů jsou navíc v Azure DevOps standardně vázány na build a po jeho odstranění jsou nenávratně ztraceny. To komplikuje snahu o dlouhodobé sledování kvality vývoje a znemožňuje zpětnou analýzu.

Celkově lze říci, že dashboard v Azure DevOps je vhodný pro základní přehled o aktuálním stavu buildů, nikoli však pro detailní, historickou a specializovanou analýzu metrik potřebnou v tomto projektu. Z těchto důvodů bylo rozhodnuto nasadit nástroje Elasticsearch a Kibana, kde Elasticsearch slouží k ukládání a vyhledávání dat a Kibana poskytuje pokročilé možnosti jejich vizualizace, filtrování a analýzy v reálném čase.

2.5 Požadavky na nový systém

S ohledem na výše popsané nedostatky současného řešení je nutné navrhnout nový systém, který zajistí centralizované a efektivní sledování, automatizované zpracování, analýzu a porovnávání všech klíčových metrik souvisejících s vývojem a testováním.

Nový systém by měl splňovat následující požadavky:

- **Jednotné úložiště dat**: Zavedení centralizované databáze pro všechny relevantní metriky, například využitím systému Elasticsearch. Všechna data budou po zpracování ukládána ve vhodném formátu, který týmu usnadní přístup k potřebným údajům pro analýzu a vizualizaci.
- **Dynamická vizualizace**: Poskytnutí aktuálních a interaktivních vizualizačních nástrojů pro rychlé a intuitivní vyhodnocování metrik. Dashboardy v systému Kibana nabídnou přehled o všech klíčových údajích, které tým potřebuje analyzovat a okamžitě zobrazit.
- **Automatizace procesů**: Nasazení automatických mechanismů pro sběr, zpracování a analýzu dat, čímž se sníží manuální náročnost a riziko vzniku chyb. Automatizace bude zajištěna prostřednictvím Azure DevOps pipeline, která spustí Python skripty, jež data zpracují a odešlou do databáze.
- **Historická analýza**: Umožnění sledování a porovnávání metrik v čase s cílem identifikovat dlouhodobé trendy a změny ve výkonnosti firmwaru. Na Kibana dashboardech bude možné zobrazit nejen aktuální hodnoty, ale i historické výsledky z různých buildů a porovnávat je mezi sebou.
- **Personalizace a filtrování**: Poskytnutí možnosti přizpůsobit si zobrazení dat podle individuálních potřeb uživatelů a filtrovat metriky podle zvolených parametrů, jako je číslo buildu, tag nebo časové období. Díky tomu bude možné například zobrazit všechna data z buildů označených tagem *Nightly* nebo porovnat hodnoty z různých časových úseků.
- **Nasazení služeb v kontejnerech**: Služby Elasticsearch a Kibana budou nasazeny v oddělených Docker kontejnerech. Toto řešení umožní rychlé nasazování nových verzí, snadnou správu, flexibilní škálování podle aktuálních potřeb a jednoduchou migraci na jiný server. Kontejnerizace zároveň zvýší izolaci jednotlivých služeb, čímž přispěje k vyšší bezpečnosti, stabilitě a udržovatelnosti systému.

Implementace takového systému by zásadně zlepšila schopnost efektivně monitorovat klíčové metriky, urychlila identifikaci a řešení problémů a celkově přispěla k vyšší kvalitě vyvíjeného firmwaru.

3 VLASTNÍ NÁVRH A IMPLEMENTACE ŘEŠENÍ

Tato část bakalářské práce se zaměřuje na praktickou realizaci návrhu, konkrétně na implementaci dashboardů ve firemním prostředí pro tým v společnosti Siemens. Hlavním cílem této kapitoly je popsat jednotlivé kroky implementace, představit automatizaci celého procesu a ukázat výsledné grafy, tabulky a další vizualizace, které tvoří dashboardy.

Před samotným popisem praktické realizace bude provedeno srovnání dostupných technologií a vysvětlení důvodů, proč byly pro tento projekt zvoleny odpovídající nástroje. Poté se kapitola zaměří na vytvoření kontejnerů pro systémy Kibana a Elasticsearch, následně na konfiguraci Azure DevOps agenta a tvorbu pipeline. Dále bude popsáno zpracování dat pomocí Python skriptů, se zaměřením na proces získávání, přípravy a ukládání dat do systému Elasticsearch. Na závěr se kapitola věnuje tvorbě jednotlivých dashboardů v prostředí Kibana.

3.1 Hodnocení a výběr technologií

Výběr technologií pro implementaci dashboardů byl založen na porovnání alternativních nástrojů a jejich klíčových vlastností. Následuje shrnutí srovnání vybraných nástrojů a vysvětlení důvodů jejich volby:

3.1.1 Porovnání Elasticsearch, Kibana a Splunk

Pro účely tohoto projektu bylo nezbytné zvolit vhodnou technologii pro ukládání, analýzu a vizualizaci dat. Před samotným výběrem technologie proběhlo porovnání *open-source* řešení v podobě Elasticsearch a Kibana s komerční platformou Splunk. Cílem je určit, která z těchto variant nejlépe odpovídá potřebám týmu.

Kibana je vizualizační nástroj pro práci s daty uloženými v Elasticsearch, jehož struktura byla popsána v sekci 1.2.4. Kromě vizualizace dat podporuje upozorňování, hlášení, monitorování komponent Elastic Stacku a správu Elasticsearch clusteru. Nabízí uživatelské rozhraní s aplikacemi, jako je Discover, Visualize a Dashboard, pro analýzu a vizualizaci dat [1].

Na druhé straně **Splunk** představuje výkonnou platformu pro vyhledávání, analýzu a vizualizaci strojově generovaných dat. Data jsou shromážděna z různých zdrojů, jako jsou webové stránky, aplikace, senzory nebo zařízení tvořících součást IT infrastruktury. Umožňuje rychlé a efektivní získávání odpovědí na klíčové otázky týkající se optimalizace výkonnosti serverů nebo zlepšení zákaznické zkušenosti [4].

Při výběru technologií byla nakonec upřednostněna kombinace Elasticsearch a Kibana. Hlavním důvodem této volby je jejich *open-source* charakter, který umožňuje jejich provozování na vlastním serveru bez nutnosti placené licence. Naproti tomu Splunk představuje komerční řešení s vyššími náklady, přičemž jeho bezplatná verze je navíc omezena limitem 500 MB indexovaných dat denně. Tento limit činí Splunk nevhodným pro potřeby projektu, zatímco Elasticsearch a Kibana nabízejí veškeré požadované funkce a z hlediska nákladů představují výrazně výhodnější variantu.

3.1.2 Porovnání Grafana a Kibana

Při návrhu řešení pro vizualizaci dat byla zvažována také alternativa v podobě nástroje **Grafana**. Tento nástroj podporuje správu a vizualizaci dat z různých zdrojů, jako jsou relační databáze, časové řady, NoSQL databáze, logy nebo soubory ve formátu CSV[1]. Umožňuje provádět analytické zpracování i v případě omezeného přístupu k datům. Od verze 8.x navíc podporuje živé streamování dat z IoT[2] prostřednictvím protokolů MQTT[3] nebo HTTP.

Grafana umožňuje vytvářet přizpůsobitelné dashboardy s širokou škálou vizualizací, jako jsou časové řady, statistické grafy, topologická nebo geografická zobrazení. Tyto dashboardy lze snadno upravovat, nebo sdílet s týmem. Kromě toho nabízí i pokročilé funkce, jako jsou datové transformace, centralizovaná správa upozornění a anotace dat přímo v grafech, což usnadňuje interpretaci dat a spolupráci mezi uživateli [7].

Oba nástroje, Grafana i Kibana, jsou určené pro vizualizaci dat. Pro tento projekt byla nakonec zvolena Kibana díky své nativní integraci s Elasticsearch. Právě kvůli této kombinaci je možné snadno vyhledávat, analyzovat a vizualizovat data bez nutnosti složité konfigurace nebo připojování dalších externích nástrojů.

3.1.3 Porovnání Azure DevOps a GitLab

V rámci návrhu projektu bylo také nutné rozhodnout, která platforma bude použita pro implementaci CI/CD pipeline. Zvažována byla dvě rozšířená řešení – Azure DevOps a GitLab. Azure DevOps byl podrobněji rozebrán v sekci 1.5.3.

GitLab je *open-source* platforma postavená na systému Git, která v sobě integruje nástroje pro správu repozitářů, plánování projektů, CI/CD, kontrolu kódu i sledování úkolů v rámci jednotného rozhraní. Obě platformy, GitLab i Azure DevOps, poskytují nástroje pro správu repozitářů, CI/CD pipeline a projektový management [3].

[1] CSV (Comma-Separated Values)
[2] IoT (Internet of Things)
[3] MQTT (Message Queuing Telemetry Transport)

Azure DevOps byl pro tento projekt zvolen z důvodu, že jej tým již využívá pro správu repozitářů a pipeline. Zavádění nového systému pouze pro tuto implementaci by bylo neefektivní z hlediska času, nákladů i potřeby školení uživatelů.

3.1.4 Porovnání Docker a Podman

Docker a Podman jsou nástroje pro práci s kontejnery, které umožňují vytvářet, spouštět a spravovat kontejnerizované aplikace. Docker byl detailněji popsán v sekci 1.4.2.

Podman ke svému provozu nevyžaduje administrátorská oprávnění, a z hlediska bezpečnosti tak představuje výhodnější variantu. V rámci tohoto projektu byl však zvolen nástroj **Docker**, a to především z důvodu předchozích zkušeností týmu s tímto nástrojem.

Problém s kontejnery vyžadujícími administrátorská oprávnění byl identifikován a následně úspěšně vyřešen. Podrobnosti k technickému řešení jsou uvedeny v následujících částech této práce.

Volba těchto technologií vycházela z potřeb týmu a z již zavedených standardů. Díky těmto nástrojům bylo možné navrhnout a implementovat dashboardy s minimálním dopadem na existující infrastrukturu a procesy.

3.2 Návrh architektury

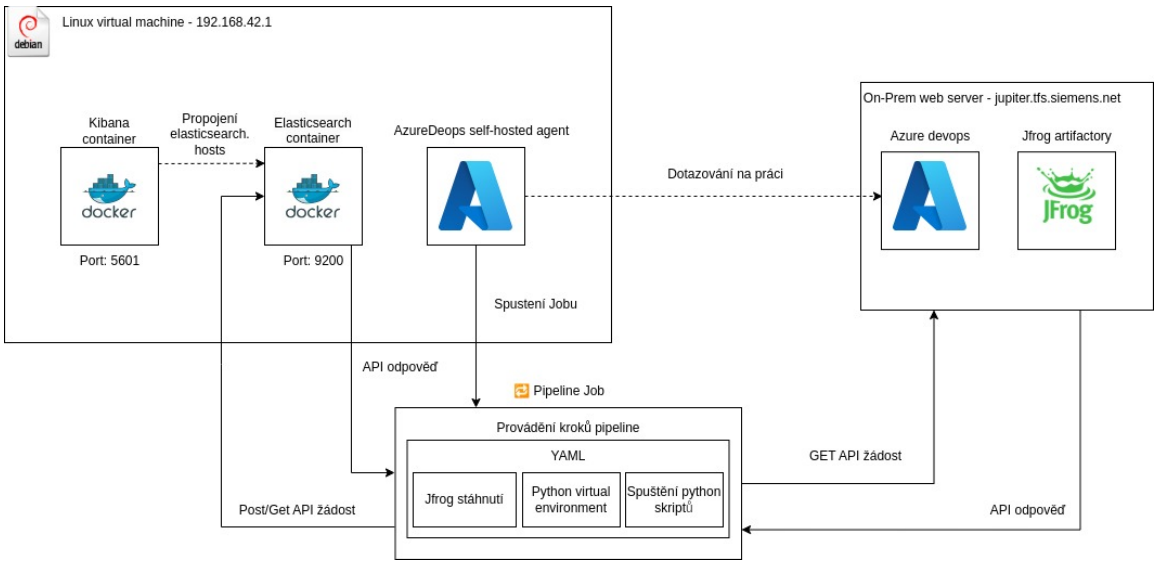
Navržená architektura je postavena na virtuálním serveru s operačním systémem Debian, na kterém jsou spuštěny Docker kontejnery obsahující služby Kibana a Elasticsearch. Klíčové metriky, které jsou monitorovány, jsou ukládány do databáze Elasticsearch, jež je propojena s nástrojem Kibana sloužícím k vizualizaci těchto dat.

Monitorovaná data jsou do Elasticsearch nahrávána prostřednictvím skriptů v jazyce Python, které jsou spouštěny v rámci pipeline Azure DevOps, provozované v lokálním prostředí (*on-premises*).

Tato pipeline běží na lokálně provozovaném agentovi (*self-hosted agent*), který je umístěn na stejném serveru. Agent pravidelně ověřuje dostupnost úloh v Azure DevOps. V případě spuštění pipeline vykonává jednotlivé kroky definované v konfiguračním souboru ve formátu YAML.

Tento soubor obsahuje úkol pro stažení JSON souborů a logů ze systému JFrog Artifactory. Získaná data jsou následně zpracována do podoby monitorovacích metrik pomocí skriptů navázaných na poslední provedený build.

Níže je znázorněn diagram architektury celého řešení:



Obrázek 3.1: Architektura systému pro automatizovaný sběr a zpracování dat v lokálním prostředí

3.3 Nasazení Docker kontejnerů

Prvním krokem implementace dashboardů je nasazení Docker kontejnerů služeb Elasticsearch a Kibana. Nejprve je nutné nainstalovat nástroj Docker na linuxový systém.

Aby bylo možné tyto kontejnery spustit, je nutné pro každou službu vytvořit vlastní Docker obraz pomocí **Dockerfile**. Tento soubor definuje způsob sestavení kontejnerového obrazu, konfiguraci služby a nastavení oprávnění.

Pro správu a orchestraci[4] více kontejnerů současně se využívá **Docker Compose**. Tento nástroj umožňuje jejich definici a správu pomocí konfiguračního souboru ve formátu YAML a jejich snadné spuštění jediným příkazem:

```
docker compose up
```

3.3.1 Vytvoření Docker kontejnerů

Před použitím nástroje Docker Compose je důležité porozumět procesu kontejnerizace jednotlivých služeb. Pro každou službu, tedy Elasticsearch a Kibanu, se vytváří samostatný soubor Dockerfile, který popisuje způsob sestavení příslušného obrazu.

Níže je uveden příklad souboru Dockerfile pro službu Elasticsearch:

[4]Orchestrace - znamená automatizované řízení a propojení více služeb při jejich nasazení a správě.

```
FROM docker.elastic.co/elasticsearch/elasticsearch:8.17.2

RUN chown -R elasticsearch:elasticsearch /usr/share/elasticsearch

USER elasticsearch

CMD ["/usr/share/elasticsearch/bin/elasticsearch"]
```

Vysvětlení příkazu FROM:

- **FROM** — Určuje, ze kterého základního obrazu bude kontejner vytvořen.
- **docker.elastic.co** — Doménové jméno registru, z něhož je obraz stahován.
- **elasticsearch** — Název repozitáře v Docker registru.
- **elasticsearch:8.17.2** — Název obrazu spolu s tagem označujícím jeho verzi.

Nastavení oprávnění — Pomocí příkazu **chown** se změní vlastník příslušných adresářů na definovaného uživatele elasticsearch. Toto nastavení slouží ke zvýšení úrovně bezpečnosti, protože výchozí Docker obraz pro Elasticsearch standardně běží s administrátorskými právy uživatele root, což představuje potenciální bezpečnostní riziko.

Přepnutí uživatele — Příkaz **USER elasticsearch** zajišťuje, že daná služba se nespouští pod uživatelem root, ale pod neprivilegovaným účtem elasticsearch, což zvyšuje bezpečnost celé implementace.

Spuštění služby — Příkaz **CMD** definuje hlavní spustitelný proces kontejneru. V tomto případě se jedná o binární soubor služby Elasticsearch, nacházející se ve standardním adresáři */usr/share/elasticsearch/bin/elasticsearch*.

3.3.2 Použití Docker Compose

Docker Compose byl zvolen pro efektivní správu více kontejnerů, které mezi sebou komunikují. Namísto ručního spouštění jednotlivých kontejnerů lze všechny služby definovat v konfiguračním souboru **docker-compose.yml** a následně je spustit jediným příkazem ve stanoveném pořadí pomocí klíče **depends_on**.

Ukázka konfiguračního souboru docker-compose.yml pro služby Elasticsearch a Kibana:

```
services:
  Elasticsearch:
    container_name: es-container
    build:
      context: .
      dockerfile: Dockerfile.Elasticsearch
    restart: always
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
      - path.repo=/usr/share/Elasticsearch/backup
      - cluster.name=docker-cluster
    networks:
      - es-net
    ports:
      - 9200:9200
    volumes:
      - ./Elasticsearch-data:/usr/share/Elasticsearch/data
      - ./backup:/usr/share/Elasticsearch/backup

  Kibana:
    container_name: kb-container
    build:
      context: .
      dockerfile: Dockerfile.Kibana
    restart: always
    networks:
      - es-net
    depends_on:
      - Elasticsearch
    ports:
      - 5601:5601
    volumes:
      - ./kibana.yml:/usr/share/kibana/config/kibana.yml

networks:
  es-net:
```

Služba Elasticsearch sestavuje Docker obraz ze souboru **Dockerfile.Elasticsearch** a konfiguruje Elasticsearch pro běh v režimu jednoho uzlu (single-node). Služba je zpřístupněna na portu 9200, který umožňuje přístup k REST API.

Trvale uložená data a snapshoty[5] jsou mapovány do adresářů mimo samotný kontejner pomocí klíče **volumes**. To zajišťuje jejich zachování i po restartu kontejneru.

Služba Kibana je nakonfigurována tak, aby se spouštěla až po inicializaci služby Elasticsearch, což je zajištěno pomocí klíče depends_on. Obraz služby Kibana je sestaven ze samostatného souboru **Dockerfile.Kibana**, stejně jako v případě Elasticsearch. Služba naslouchá na portu 5601, což je výchozí port webového rozhraní Kibany. Konfigurace je načítána z vlastního konfiguračního souboru **kibana.yml**, který je připojen z hostitelského systému.

Obě služby sdílejí síť s názvem **es-net**, což umožňuje jejich vzájemnou komunikaci přes definované názvy kontejnerů.

Popis konfiguračního souboru kibana.yml

Konfigurační soubor **kibana.yml** slouží k nastavení základních parametrů služby Kibana. Definuje způsob přístupu k webovému rozhraní Kibana a specifikuje připojení k instanci Elasticsearch.

Ukázka obsahu souboru **kibana.yml**:

```
# kibana.yml
server.host: "0.0.0.0"
server.publicBaseUrl: "http://192.168.42.1:5601"
elasticsearch.hosts: ["http://es-container:9200"]
```

Přehled klíčových parametrů

- **server.host: "0.0.0.0"**
  Umožňuje službě Kibana běžet na všech síťových rozhraních hostitelského systému, díky čemuž je možné přistupovat k rozhraní i z jiných zařízení v síti.

- **server.publicBaseUrl: "http://192.168.42.1:5601"**
  Definuje veřejnou URL adresu, pod kterou je služba Kibana dostupná. IP adresa **192.168.42.1** odpovídá síťovému rozhraní serveru a port **5601** představuje výchozí port webového rozhraní.

- **elasticsearch.hosts:** [**"http://es-container:9200"**]
  Nastavuje připojení ke službě Elasticsearch běžící v kontejneru **es-container** na portu **9200**. Kibana tímto způsobem získává data přímo z databáze Elasticsearch pro následnou vizualizaci.

[5]Snapshot - v systému Elasticsearch představuje uloženou zálohu indexů a stavu clusteru k určitému časovému okamžiku pro účely obnovení dat.

3.4 Vytvoření lokálně provozovaného agenta na linuxovém serveru

Pro zajištění automatizovaného spouštění úloh definovaných v rámci pipeline nástroje Azure DevOps je třeba nakonfigurovat lokálně provozovaného agenta na linuxovém serveru. Tento takzvaný *self-hosted agent* umožňuje spouštění pipeline přímo na interním serveru, čímž se eliminuje nutnost využívat cloudové agenty poskytované společností Microsoft. Hlavní výhodou tohoto přístupu je úspora nákladů, protože provoz nevyžaduje využití zpoplatněných cloudových agentů.

3.4.1 Nastavení Git a připojení repozitáře

Pro bezpečné a pohodlné připojení k repozitáři Azure DevOps prostřednictvím protokolu SSH[6] je nejprve třeba vygenerovat pár SSH klíčů – soukromý a veřejný. Veřejný klíč se následně vloží do Azure DevOps, zatímco soukromý zůstává uložen na lokálním linuxovém serveru a slouží k autentizaci připojení.

Aby mohla komunikace mezi serverem a Azure DevOps probíhat automaticky, bez nutnosti opakovaného zadávání přihlašovacích údajů, je nutné upravit konfiguraci SSH klienta v souboru **/.ssh/config**. V tomto souboru se specifikují parametry připojení, jako je uživatelské jméno a cesta k privátnímu klíči, což umožňuje bezheslové spojení se vzdáleným repozitářem.

3.4.2 Instalace a konfigurace lokálně provozovaného agenta

Tato část popisuje postup instalace a konfigurace lokálně provozovaného agenta služby Azure DevOps na linuxovém serveru. Postup vychází z oficiální dokumentace Azure DevOps [12] a slouží k nastavení a propojení serveru s touto platformou.

Stažení a příprava instalačního balíčku

Nejprve je nutné vytvořit pracovní adresář, do kterého bude agent nainstalován. V terminálu se postupně provedou následující kroky:

- Vytvoření adresáře:

  mkdir /azure-agent

- Přesunutí do nově vytvořeného adresáře:

  cd /azure-agent

[6] SSH (Secure Shell) - zabezpečený síťový protokol sloužící k šifrované komunikaci mezi klientem a serverem, typicky používaný pro vzdálený přístup a správu systémů.

Následně je třeba stáhnout instalační balíček agenta z lokální instance Azure DevOps používané ve společnosti Siemens. Balíček je dostupný po přihlášení do webového rozhraní platformy. Po jeho stažení se soubor přesune na server do připraveného adresáře a rozbalí se pomocí příkazu:

tar -zxvf vsts-agent.tar.gz

Konfigurace agenta

Po rozbalení instalačního balíčku se spustí konfigurační skript:

./config.sh

Tento skript provede základní nastavení agenta a jeho propojení s platformou Azure DevOps. Během interaktivní konfigurace je vyžadováno zadání adresy organizace v Azure DevOps, autentizačního tokenu (PAT) a názvu agenta.

Spuštění agenta jako systemd služby

Aby se agent spouštěl automaticky po restartu systému a běžel na pozadí jako systémová služba, je třeba jej zaregistrovat jako službu **systemd**[7]. To se provede pomocí následujících příkazů:

- Instalace agenta jako systemd služby:

  sudo ./svc.sh install

- Okamžité spuštění služby:

  sudo ./svc.sh start

Tímto způsobem bude agent automaticky spuštěn při každém restartu linuxového serveru. Po úspěšné konfiguraci se agent zobrazí v rozhraní Azure DevOps se stavem *online* a bude připraven k použití v rámci pipeline.

3.4.3 Postup práce Python skriptů při zpracování dat

Důležitá vývojová data jsou do systému Elasticsearch odesílána prostřednictvím skriptů v jazyce Python. Každý graf, tabulka či jiná vizualizace v prostředí Kibana odpovídá jednomu skriptu. Skripty jsou navrženy tak, aby systematicky vykonávaly předem definované kroky zpracování dat.

Každý Python skript se skládá z několika funkcí, jejichž implementace je podrobně popsána v následující části. Níže je uveden přehled základního postupu, který tyto skripty vykonávají.

[7] systemd – nástroj pro správu služeb a procesů v systému Linux.

Získávání dat z posledního buildu pipeline

Data z aktuálně dokončeného buildu pipeline jsou získávána dvěma způsoby:

- **Prostřednictvím Azure DevOps REST API**: Python skripty využívají rozhraní REST API platformy Azure DevOps, které umožňuje získávat podrobné informace o průběhu jednotlivých buildů pipeline.
- **Pomocí úkolu v Azure DevOps pipeline**: Pro automatizované stahování souborů ze systému JFrog Artifactory se využívá specializovaný úkol (task), definovaný přímo v YAML konfiguraci Azure DevOps pipeline. Tento úkol zajišťuje přenos datových souborů na cílový linuxový server, kde je nasazen lokálně provozovaný agent.

Kontrola duplicity dat

Před samotným zpracováním dat každý Python skript ověřuje, zda již záznam s daným číslem buildu nebyl do systému Elasticsearch uložen. Tato kontrola je nezbytná pro zachování konzistence dat a prevenci vzniku duplicitních záznamů v databázi.

Zpracování datových souborů a JSON odpovědí z Azure DevOps

Po úspěšné kontrole duplicity skripty načítají soubory, které byly staženy pomocí specializovaného úkolu v pipeline služby Azure DevOps. Tyto soubory se nacházejí v předdefinované cestě na linuxovém serveru.

Skripty zpracovávají také odpovědi získané z rozhraní Azure DevOps REST API, které jsou ve formátu JSON. Z těchto odpovědí jsou následně extrahovány klíčové informace nezbytné pro další analýzu a vizualizaci v nástroji Kibana.

Odesílání dat do Elasticsearch

Na základě získaných a zpracovaných informací jsou vytvořeny strukturované slovníky obsahující všechna potřebná data. Tato data jsou následně odeslána do systému Elasticsearch, kde probíhá jejich indexace. Po úspěšném indexování jsou data připravena k detailní analýze a vizualizaci v nástroji Kibana.

3.4.4 Popis Python funkcí

Ke každému kroku postupu práce skriptů, jak byl popsán výše, je přiřazena klíčová funkce, která daný krok implementuje. Každá z těchto funkcí je doplněna o popis vstupních parametrů, výstupních hodnot a její role v celkovém procesu zpracování dat.

Funkce pro získávání dat z posledního buildu pipeline

Účelem funkce **find_build_id** je získat identifikátor posledního provedeného buildu vybrané pipeline v prostředí Azure DevOps. K tomuto účelu se využívá metoda GET z rozhraní Azure DevOps REST API, konkrétně volání *Builds - List*, které vrací seznam buildů.

Tato metoda umožňuje filtrování výsledků podle různých parametrů. V tomto případě je seznam buildů omezen parametrem **branchName**, který určuje větev, k níž se buildy vztahují.

Ukázka požadavku může vypadat následovně:

```
GET {organization_url}/{project}/_apis/build/builds?api-version=7.0&
branchName={branch}
```

Tímto způsobem funkce získá pouze buildy, které se vztahují ke zvolené větvi.

Ukázka implementace

```
def find_build_id(organization_url, project, auth, pipeline_name, branch):
    """
    Finds the build ID for a specified pipeline in Azure DevOps.
    """
    url = (
        f"{organization_url}/{project}/_apis/build/builds?"
        f"api-version=7.0&branchName={branch}"
    )
    response = requests.get(url, auth=auth, timeout=10)
    response.raise_for_status()

    for build in response.json().get("value", []):
        if build["definition"]["name"] == pipeline_name:
            return build["id"], build["buildNumber"], build["finishTime"]

    return None, None, None
```

Parametry funkce

- **organization_url** – Základní URL adresa organizace v prostředí Azure DevOps.
- **project** – Název projektu v rámci dané organizace.
- **auth** – Autentizační údaje (token), které umožňují přístup k Azure DevOps REST API.
- **pipeline_name** – Název pipeline, jejíž buildy mají být vyhledány.

- **branch** – Název větve, podle které mají být buildy filtrovány.

Průběh zpracování

Funkce nejprve sestaví URL adresu API požadavku. Tato adresa obsahuje základní URL organizace, název projektu, verzi API a název větve, podle kterého jsou buildy filtrovány.

Po sestavení URL je pomocí knihovny **requests**[8] odeslán HTTP požadavek typu GET. Tento požadavek obsahuje potřebné autentizační údaje a je nastaven s časovým limitem 10 sekund pomocí parametru *timeout=10*. Pokud během komunikace s API dojde k chybě, například z důvodu neplatné odpovědi nebo překročení časového limitu, funkce vyvolá výjimku pomocí metody **response.raise_for_status()**.

V případě úspěšného provedení požadavku je vrácena odpověď ve formátu JSON, ze které funkce získá seznam buildů z klíče *value*. Následně funkce prochází seznam a pro každý build ověřuje, zda splňuje následující podmínku:

```
build["definition"]["name"] == pipeline_name
```

Položka **definition** je slovník obsahující metadata buildu. Klíč **name** specifikuje název pipeline, ze které daný build pochází. Podmínka tedy ověřuje, zda nalezený build odpovídá názvu pipeline zadanému jako vstupní parametr.

Pokud je nalezen build, jehož název pipeline odpovídá požadovanému názvu, funkce vrací trojici informací:

- **build["id"]** – Unikátní číselné ID buildu přidělené službou Azure DevOps.
- **build["buildNumber"]** – Identifikátor buildu v rámci dané pipeline.
- **build["finishTime"]** – Časové razítko označující okamžik dokončení buildu.

Pokud odpovídající build není nalezen nebo dojde k chybě, funkce vrací trojici prázdných hodnot *None*, čímž indikuje neúspěšné získání požadovaných dat.

Funkce pro kontrolu duplicity dat

Účelem funkce **check_build_number_in_Elasticsearch** je ověřit, zda v systému Elasticsearch již existují data spojená s daným číslem buildu.

[8] requests - knihovna v jazyce Python určená pro odesílání HTTP požadavků.

Ukázka implementace

```
def check_build_number_in_Elasticsearch(client, index_name, build_number):
    """Checks if a specific build number exists in Elasticsearch."""
    query = {"query": {"match": {"build_number": build_number}}, "size": 1}
    res = client.search(index=index_name, body=query)
    return bool(res["hits"]["hits"])
```

Parametry funkce

- **client** – Instance klienta Elasticsearch, prostřednictvím které funkce komunikuje s databází.
- **index_name** – Název indexu v Elasticsearch, ve kterém se provádí vyhledávání.
- **build_number** – Jedinečný identifikátor buildu pipeline, jehož přítomnost je nutné ověřit z důvodu možné duplicity.

Průběh zpracování

Funkce vytváří dotaz do systému Elasticsearch, který vyhledává dokumenty obsahující zadané číslo buildu. Tento dotaz je odeslán metodou **client.search**, jež zajišťuje komunikaci s databází. Následně funkce vyhodnotí odpověď a vrací *True*, pokud byl nalezen alespoň jeden odpovídající dokument, nebo *False*, pokud žádný dokument s daným číslem buildu nebyl nalezen.

Funkce pro zpracování datových souborů

Účelem funkce **extract_logfile_content** je získat obsah logovacího souboru uloženého v archivu ve formátu ZIP. Extrakce probíhá na základě porovnání názvů souborů podle zadané části názvu, což umožňuje univerzální použití této funkce pro různé typy logů, u nichž není přesný název souboru předem znám.

Ukázka implementace

```
def find_partial_sentence_line(sentences, partial_sentence):
    """
    Searches for a partial sentence within a list and returns the first match.
    """
    for sentence in sentences:
        if partial_sentence in sentence:
            return sentence
    return None

def extract_logfile_content(zip_path, target_log_file_path):
    """
    Extracts the content of the log file from the zip archive.
```

```
    Args:
        zip_path: The path to the zip file.
        target_log_file_path: The substring used to identify the log file within the archive.

    Returns:
        The log file content as bytes.
    """
    try:
        with ZipFile(zip_path, "r") as zip_ref:
            names = zip_ref.namelist()
            log_file_path = find_partial_sentence_line(names, target_log_file_path)
            if log_file_path:
                with zip_ref.open(log_file_path, "r") as logfile:
                    return logfile.read()
            else:
                raise FileNotFoundError(
                    f"No log file containing '{target_log_file_path}' found in the archive."
                )
    except BadZipFile as e:
        raise BadZipFile(
            f"The file at {zip_path} is not a valid zip file or is corrupted."
        ) from e
```

Parametry funkce

- **zip_path** – Cesta k archivu ve formátu ZIP, ze kterého má být log soubor extrahován.
- **target_log_file_path** – Částečný název cílového log souboru, podle kterého se provádí vyhledávání v archivu.

Průběh zpracování

Funkce nejprve otevře archiv ve formátu ZIP a získá seznam všech souborů, které jsou v něm obsaženy. Pro nalezení požadovaného logovacího souboru uvnitř archivu využívá pomocnou funkci s názvem **find_partial_sentence_line**. Tato funkce prochází seznam názvů souborů a hledá první název obsahující zadaný podřetězec ze vstupního parametru **target_log_file_path**.

Pokud je odpovídající soubor nalezen, otevře se a jeho obsah se načte do paměti ve formě bajtového řetězce. V případě, že žádný soubor s odpovídajícím názvem v archivu neexistuje, funkce vyvolá výjimku *FileNotFoundError*. Pokud je archiv ve formátu ZIP poškozený nebo neplatný, funkce vyvolá výjimku *BadZipFile*.

Funkce pro odesílání dat do Elasticsearch

Hlavním cílem funkce **send_unittest_results_to_Elasticsearch** je uložit výsledky unit testů do databáze Elasticsearch ve formátu vhodném pro následnou vizualizaci v nástroji Kibana.

Ukázka implementace

```
def send_unittest_results_to_Elasticsearch(
    client, index_name, build_number, test_results, build_tag
):
    """
    Sends the unit test results to the Elasticsearch index.
    """
    if not test_results:
        _log.info("No unit test results available. Exiting script.")
        sys.exit(1)

    for component, stats in test_results.items():
        document = {
            "component": component,
            "passed": stats["passed"],
            "failed": stats["failed"],
            "build_number": build_number,
            "build_tag": build_tag,
            "timestamp": datetime.now().strftime("%Y-%m-%dT%H:%M:%S+01:00"),
        }
        _log.info(f"Sending data to Elasticsearch: {document}")
        send_to_Elasticsearch(client, index_name, document)
```

Parametry funkce

- **client** – Objekt klienta Elasticsearch, používaný pro komunikaci s databází.
- **index_name** – Název indexu, do kterého budou výsledky ukládány.
- **build_number** – Jedinečný identifikátor buildu v rámci pipeline.
- **test_results** – Slovník obsahující výsledky unit testů.
- **build_tag** – Tag buildu, například „Release“ nebo „Nightly“.

Průběh zpracování

Funkce nejprve ověří, zda jsou k dispozici výsledky unit testů. Pokud nejsou k dispozici žádné výsledky, vypíše odpovídající zprávu do logu a ukončí vykonávání skriptu. V opačném případě iteruje přes jednotlivé testované komponenty a pro každou extrahuje počet úspěšných a neúspěšných testů. Na základě těchto dat vytvoří

dokument obsahující název komponenty, číslo buildu, tag buildu, časovou značku (*timestamp*) a počty prošlých a neprošlých testů. Následně tyto dokumenty odešle do systému Elasticsearch prostřednictvím funkce **send_to_Elasticsearch**.

3.4.5 Struktura Azure DevOps pipeline definované pomocí YAML souboru

Pipeline v prostředí Azure DevOps je navržena pro automatizaci celého procesu zpracování dat. Spouští se třemi různými způsoby na předem zvoleném poolu agentů. Pipeline začíná stažením potřebných dat ze vzdáleného úložiště, po kterém následuje příprava prostředí pro běh Python skriptů. Tato fáze zahrnuje nastavení požadované verze Pythonu a vytvoření izolovaného virtuálního prostředí. Po úspěšném nastavení jsou postupně spouštěny jednotlivé skripty pro zpracování dat.

Nastavení spouštění pipeline

Úvodní část YAML souboru definuje podmínky, za kterých se pipeline spouští. Jsou specifikovány tři různé způsoby spuštění, které jsou popsány níže:

```
resources:
  pipelines:
    - pipeline: source_pipeline
      project: IOGrid
      source: et2678.device-dev.build
      trigger: true
schedules:
  - cron: '0 6 * * Mon,Tue,Wed,Thu,Fri,Sat,Sun'
    displayName: 'Scheduled Run'
    branches:
      include:
        - main
    always: true
    batch: false
trigger:
  batch: true
  branches:
    include:
      - main
```

Blok *resources* umožňuje spustit tuto pipeline automaticky po úspěšném dokončení jiné pipeline s názvem **et2678.device-dev.build**. Jakmile je tato vstupní pipeline dokončena, navazující pipeline se spustí automaticky.

Blok *schedules* definuje plánované spouštění pipeline. V tomto případě je nastaven tak, aby se pipeline spustila každý den v 6 hodin ráno.

Blok *trigger* zajišťuje automatické spuštění pipeline při každé změně ve větvi *main*, například při commitu nebo vytvoření pull requestu.

Tato nastavení zajišťují, že se pipeline spouští v reakci na změny v kódu, po dokončení jiné pipeline a také na základě pravidelného časového plánu.

Nastavení poolu a výběr agenta

Pro správné spuštění pipeline je nejprve nutné specifikovat pool agentů a stanovit požadavky (*demands*) na výběr agenta, který splňuje definované podmínky.

Ukázka definice v YAML:

```
pool:
  name: SM-JUP-CZBRNO-IPS
  demands:
    - Agent.Name -equals elk_1.192.168.42.1
```

Pro běh této pipeline je specifikován pool agentů s názvem *SM-JUP-CZBRNO-IPS*. Tento pool zahrnuje agenty, na kterých mohou být úlohy pipeline spouštěny.

Nastavením požadavku je určeno, že se pipeline smí spustit pouze na agentovi se jménem *elk_1.192.168.42.1*. Tento agent je nasazen na linuxovém serveru a je připraven vykonávat jednotlivé kroky pipeline.

Specifikace jednotlivých kroků

```
steps:
  - task: JFrogGenericArtifacts@1
    inputs:
      command: 'Download'
      connection: 'Captain IOGrid V2 CZ-PRG'
      specSource: 'taskConfiguration'
      fileSpec: |
        {
          "files": [
            {
              "pattern": "et2678.device-dev.build/dev/*"
            }
          ]
        }
      failNoOp: true
  - task: UsePythonVersion@0
    inputs:
      versionSpec: '3.12'
    displayName: "Use Python 3.12"
  - script: |
      pip install pipenv && pipenv install --deploy
    displayName: "Setup Python Virtual Environment"
    workingDirectory: $(Build.SourcesDirectory)/python-scripts
  - script: |
      latest_mf_path=dev/$(ls dev | grep -F MF | sort -V | tail -n 1)
      latest_build_number_path=$latest_mf_path/$(ls $latest_mf_path | grep -E "[0-9]{8}.+[0-9]+ - (Nightly)|(Release)" | sort -V | tail -n 1)
      echo $latest_mf_path
      echo $latest_build_number_path
      echo "##vso[task.setvariable variable=build_number;]$latest_build_number_path"
    displayName: "Find Latest Build"
    continueOnError: true
  - script: pipenv run python -m fwTime.fw_time
    env:
      BUILD_SOURCES_DIR: $(Build.SourcesDirectory)
    displayName: "Run FwTime script"
    workingDirectory: $(Build.SourcesDirectory)/python-scripts
    continueOnError: true
```

Obrázek 3.2: Konfigurace Azure DevOps pipeline

Stažení artefaktů z JFrog Artifactory

Důležitým krokem pipeline je úkol *JFrogGenericArtifacts@1*, který slouží ke stažení souborů z úložiště JFrog Artifactory. Připojení je realizováno prostřednictvím služby *Captain IOGrid V2 CZ-PRG*. Stahované soubory odpovídají vzoru *et2678.device-dev.build/dev/**, což znamená, že budou staženy všechny soubory nacházející se v této cestě. Pokud nejsou nalezeny žádné soubory, pipeline nebude považována za neúspěšnou díky nastavení parametru *failNoOp: true*.

Nastavení prostředí pro běh Python skriptů

Dalším krokem je příprava prostředí pro spouštění Python skriptů. Používá se verze *Python 3.12*, která se nastavuje pomocí úkolu *UsePythonVersion@0*. Následně probíhá instalace nástroje *Pipenv*, který slouží ke správě závislostí a vytvoření izolovaného virtuálního prostředí projektu. Instalace se provádí pomocí následujícího příkazu:

```
pip install pipenv && pipenv install --deploy
```

Tento příkaz nejprve nainstaluje samotný nástroj *pipenv* a následně na základě souboru *Pipfile.lock* vytvoří a nastaví virtuální prostředí obsahující všechny potřebné závislosti.

Na obrázku níže je zobrazena ukázka souboru Pipfile.lock, který definuje přesné verze všech závislostí používaných v projektu:

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
autopep8 = "==2.0.4"
azure-core = "==1.29.5"
azure-devops = "==7.1.0b3"
certifi = "==2023.7.22"
charset-normalizer = "==3.3.2"
elastic-transport = "==8.10.0"
elasticsearch = "==8.10.1"
idna = "==3.4"
isodate = "==0.6.1"
msrest = "==0.7.1"
oauthlib = "==3.2.2"
pycodestyle = "==2.11.1"
python-dotenv = "==1.0.0"
requests = "==2.31.0"
requests-oauthlib = "==1.3.1"
requests_auth = "==8.0.0"
six = "==1.16.0"
typing_extensions = "==4.8.0"
urllib3 = "==2.0.7"
loguru = "==0.7.2"
airium = "==0.2.6"

[requires]
python_version = "3.12"
```

Obrázek 3.3: Pipfile.lock s definovanými závislostmi

Celý proces instalace závislostí probíhá *$(Build.SourcesDirectory)/python-scripts*, který slouží jako pracovní adresář pipeline. V tomto umístění se nacházejí všechny skripty potřebné ke spuštění.

Vyhledání posledního buildu

Další krok zahrnuje použití skriptu v jazyce *Bash*[9], který vyhledává nejnovější build v definované složce. Nejprve je pomocí následujícího příkazu identifikován nejnovější adresář ve složce *dev/*:

```
grep -F MF | sort -V | tail -n 1
```

Následně je v tomto adresáři vyhledáno nejnovější číslo buildu odpovídající následujícímu regulárnímu výrazu:

```
{[0-9]{8}.+[0-9]+ - (Nightly)|(Release)
```

Výsledná hodnota je uložena do proměnné prostřednictvím příkazu:

```
##vso[task.setvariable variable=build\_number;]
```

Tento krok zajišťuje, že pipeline vždy pracuje s nejnovější verzí buildu, čímž se eliminuje potřeba ručního zadávání.

Spuštění Python skriptu pro analýzu dat

V následujícím kroku se spouští Python skript pomocí příkazu:

```
pipenv run python -m fwTime.fw_time
```

Skript využívá systémovou proměnnou *BUILD_SOURCES_DIR*, která odkazuje na cestu ke staženým souborům. Tyto soubory slouží jako vstupní data pro analýzu, kterou skript provádí.

Odstranění virtuálního prostředí a dočasných souborů

Závěrečný krok pipeline slouží k úklidu pracovního prostředí. Spouští se skript, který odstraní vytvořené virtuální prostředí Pythonu a veškeré dočasné soubory vzniklé během zpracování.

```
- script: |
    pipenv --rm
    rm -rf dev
  displayName: "Delete Python Virtual Environment and Files"
```

[9]Bash (Bourne Again Shell) - příkazový řádek (shell).

3.5 Implementace dashboardu v Kibaně

Tato kapitola se zabývá návrhem a implementací dashboardů v prostředí Kibana, které slouží ke sledování a vizualizaci dat uložených v databázi Elasticsearch. Postupně budou představeny vybrané části dashboardů.

Hlavní dashboard s názvem **MultiComm** obsahuje klíčové vizualizace zaměřené na stav pipeline, výsledky unit testů, výstupy statické analýzy kódu a přehled otevřených chyb (open bugs). Vizualizace testovacích plánů z prostředí Azure DevOps je prezentována samostatně v dashboardu **Tests from CI/CD**. Třetím dashboardem je **Jitter Dashboard**, který slouží k vizualizaci výsledků měření síťových charakteristik.

3.5.1 Vizualizace stavu pipeline

Vizualizace stavu pipeline slouží k přehlednému sledování průběhu jednotlivých fází jejího zpracování. Data využívaná pro tuto část dashboardu jsou ukládána do samostatných indexů v databázi Elasticsearch.

Níže je uveden příklad dokumentu uloženého v indexu:
pipeline_build_firmware_status:


```
{
  "_index": "pipeline_build_firmware_status",
  "_id": "eFskmpUBB2j-F80qT1x",
  "_source": {
    "build_number": "20250315.1",
    "status": 0,
    "timestamp": "2025-03-15T12:00:00+02:00",
    "build_tag": "Release"
  },
  "fields": {
    "build_number": ["20250315.1"],
    "build_tag": ["Release"],
    "status": [0],
    "timestamp": ["2025-03-15T10:00:00.000Z"]
  }
}
```


V poli *_source* jsou uložena data, která byla do databáze Elasticsearch odeslána prostřednictvím Python skriptů. Každý záznam je při indexaci automaticky označen

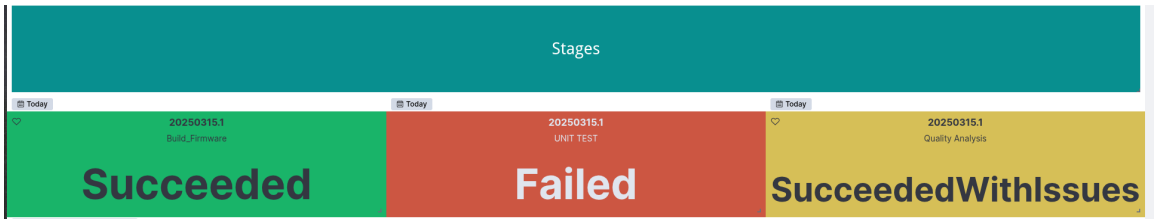
jedinečným identifikátorem *_id*. Pole *fields* obsahuje data shodná s obsahem pole *_source*, avšak ve formátu přizpůsobeném pro dotazování v prostředí Kibana.

Význam jednotlivých atributů dokumentu

- **build_number** – Jedinečný identifikátor buildu v rámci pipeline.
- **status** – Výsledek provedení dané fáze pipeline:
  - **1** – build byl úspěšně dokončen (*Succeeded*)
  - **0.5** – build byl dokončen s drobnými problémy (*Succeeded with Issues*)
  - **0** – build byl neúspěšný (*Failed*)
- **timestamp** – Datum a čas spuštění daného buildu.
- **build_tag** – Označení typu buildu, například *Release*.

Každá fáze je v dashboardu zobrazena jako samostatný barevně odlišený blok, který vizuálně označuje výsledek buildu:

- **Zelená (Succeeded)** – Build byl úspěšný.
- **Červená (Failed)** – Build selhal.
- **Žlutá (SucceededWithIssues)** – Build byl dokončen s drobnými problémy.



Obrázek 3.4: Zobrazení výsledků jednotlivých fází pipeline

Postup konfigurace vizualizace

Tato vizualizace je nakonfigurována tak, aby zobrazovala pouze poslední dokončený build. V sekci *Break down* je přidáno pole *build_number*, které umožňuje rozdělení dat podle jednotlivých buildů. Aby se zobrazoval vždy pouze nejaktuálnější záznam, je v nastavení *Number of values* ponechána hodnota *1*, v sekci *Functions* je zvolena funkce *Last value*, a pro správné řazení je v polích *Rank by* a *Rank direction* nastaveno *Alphabetical* a *Descending*. Díky tomu se jako první zobrazí build s nejvyšším číslem.

Číselné hodnoty stavu jsou převedeny na srozumitelné textové popisky pomocí funkce *Static lookup* v nastavení formátování dat, čímž se zvyšuje čitelnost výsledného výstupu.

Na závěr je vizualizace upravena také po vizuální stránce. Pod číslem buildu je zobrazován název příslušné fáze pipeline a barevné odlišení je nastavováno dynamicky podle stavu buildu. Tím je zajištěno, že vizualizace vždy odráží aktuální stav posledního buildu.

3.5.2 Vizualizace výsledků Unit Testů

V této části je popsána implementace vizualizace výsledků unit testů, které jsou ukládány do databáze Elasticsearch v rámci indexu **unit_test**. Vizualizace umožňuje analyzovat úspěšnost jednotlivých testovaných komponent a porovnávat výsledky napříč různými buildy.

Klíčové informace ze struktury dokumentu

- **build_number** – Jedinečný identifikátor buildu v rámci pipeline, ve kterém byly testy provedeny.
- **component** – Název testované komponenty.
- **passed** – Počet úspěšných testů.
- **failed** – Počet neúspěšných testů.
- **timestamp** – Datum a čas běhu testů.

Unit tests

COMPONENT	20250430.1 › PASSED	20250430.1 › FAILED	20250429.1 › PASSED	20250429.1 › FAILED
DFKA	69	0	69	0
KIConnection	8	3	10	1
LAIterator	1	0	1	0
MRC	16	1	16	1
RSDConnection	10	0	10	0
RSDEMrcn	5	1	5	1
RSDEiplOKep	2	3	4	1
RSDEp	55	0	52	3
RSDMtcpService	5	0	5	0
	Sum: 171	Sum: 8	Sum: 172	Sum: 7

Obrázek 3.5: Tabulkové zobrazení výsledků unit testů podle komponent a buildů

Tabulka zobrazuje výsledky unit testů rozdělené podle názvů testovaných komponent ve sloupci *COMPONENT* a čísel buildů, například *20250306.1* a *20250220.1*.

Pro každý build jsou v tabulce uvedeny dva samostatné sloupce s výsledky testů:

- **PASSED** – uvádí počet úspěšně provedených testů
- **FAILED** – uvádí počet selhaných testů

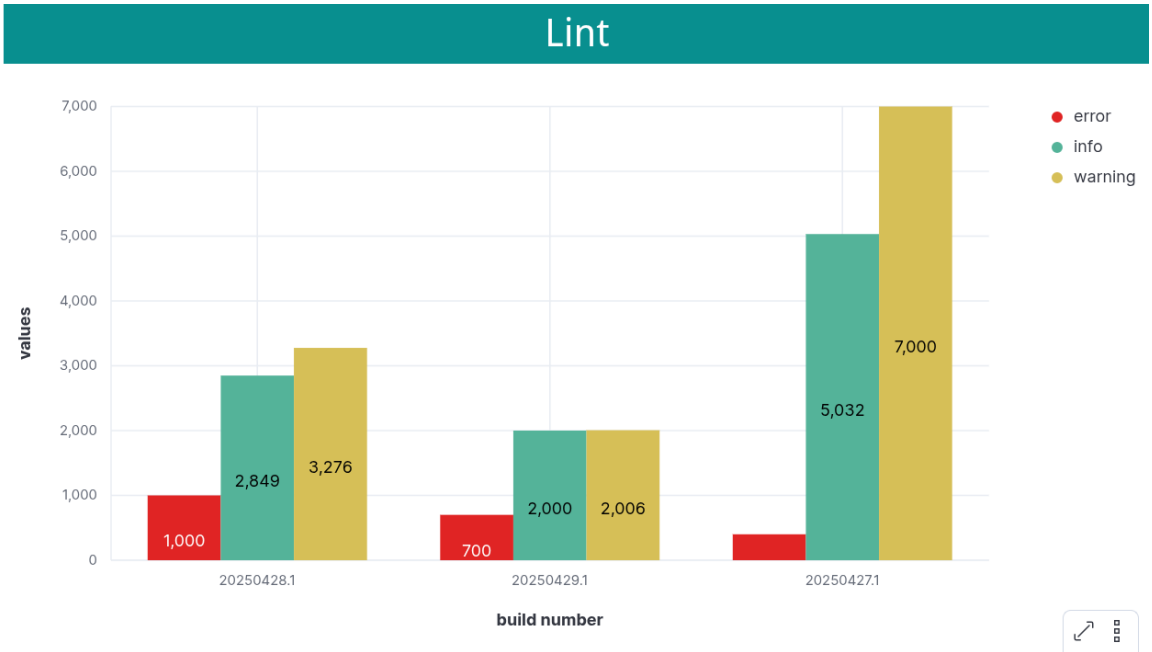
Poslední řádek tabulky obsahuje součty testů zvlášť pro každý build, což umožňuje rychle vyhodnotit celkovou úspěšnost jednotlivých buildů.

3.5.3 Vizualizace statické analýzy kódu (Lint)

Tato vizualizace zobrazuje výsledky statické analýzy kódu (Lint) pro jednotlivé buildy v rámci pipeline. Vizuální výstup je realizován ve formě sloupcového grafu.

Struktura vizualizace

- **Červená (error)** – označuje kritické chyby v kódu, které je nutné opravit.
- **Zelená (info)** – představuje informační hlášení, která obvykle nevyžadují zásah.
- **Žlutá (warning)** – značí varování upozorňující na potenciální problémy v kódu.



Obrázek 3.6: Grafické zobrazení chyb, varování a informací ze statické analýzy kódu

Postup konfigurace vizualizace

Konfigurace této vizualizace v Kibana Lens zahrnuje nastavení pole *build_number* na horizontální osu, což umožňuje rozdělení dat podle jednotlivých buildů. Na verti-

kální ose jsou zobrazeny počty chyb, informačních hlášení a varování, reprezentující různé úrovně problémů v kódu.

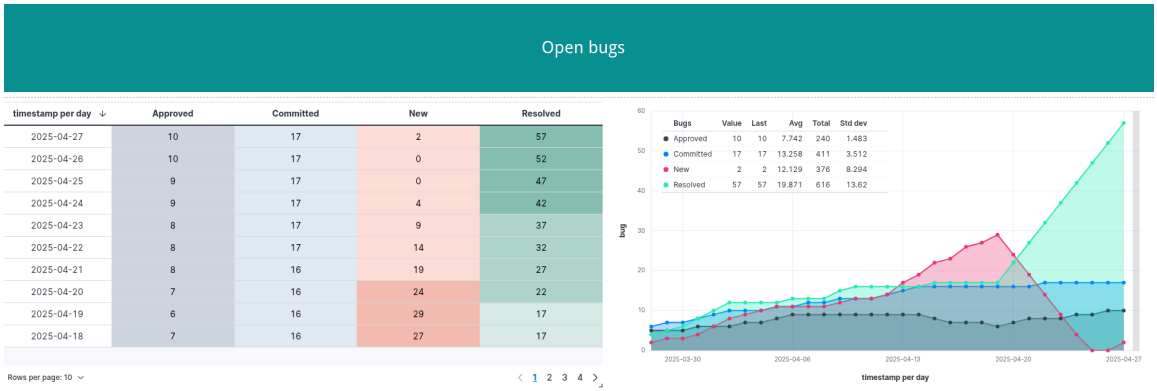
Pro lepší vizuální interpretaci byl zvolen typ grafu *Bar (Unstacked)*, který přehledně zobrazuje a umožňuje porovnávat počty chyb, informačních hlášení a varování napříč jednotlivými buildy. Barevné rozlišení – červená pro chyby, modrá pro informace a žlutá pro varování – umožňuje rychlou identifikaci závažnosti problémů v každém buildu.

3.5.4 Vizualizace Open Bugs

Tato vizualizace představuje zjednodušený pohled na životní cyklus chyb v rámci vývojového procesu. Zobrazuje pouze klíčové fáze správy chyb:

- **New (Nové)** – chyby, které byly zaznamenány, ale dosud nebyly přiřazeny k opravě.
- **Approved (Schválené)** – chyby schválené k opravě, u nichž zatím nebyla provedena implementace.
- **Committed** – chyby, jejichž opravy byly provedeny v kódu a zapsány do verzovacího systému, ale dosud nebyly nasazeny do produkčního prostředí.
- **Resolved (Vyřešené)** – chyby, které byly opraveny a označeny jako vyřešené.

Vizualizace je rozdělena na dvě části – tabulku a kumulativní graf. Zobrazuje celkový počet chyb v čase pro každou kategorii a umožňuje snadné sledování vývoje problémů v průběhu jednotlivých buildů.



Obrázek 3.7: Analýza řízení životního cyklu chyb

Postup konfigurace vizualizace

Pro konfiguraci této vizualizace je nutné vybrat typ grafu Area (kumulativní) a nastavit ho na *Unstacked*, což znamená, že jednotlivé kategorie se nebudou překrývat. Na horizontální osu se přiřadí pole *timestamp*, které reprezentuje časová razítka pro jednotlivé dny. Na vertikální ose je třeba zobrazit pole odpovídající kategoriím chyb: *New*, *Approved*, *Committed* a *Resolved*. V nastavení legendy je vhodné zvolit volbu *Auto* pro viditelnost a umístit ji pomocí volby *Inside*, aby se legenda zobrazovala uvnitř grafu.

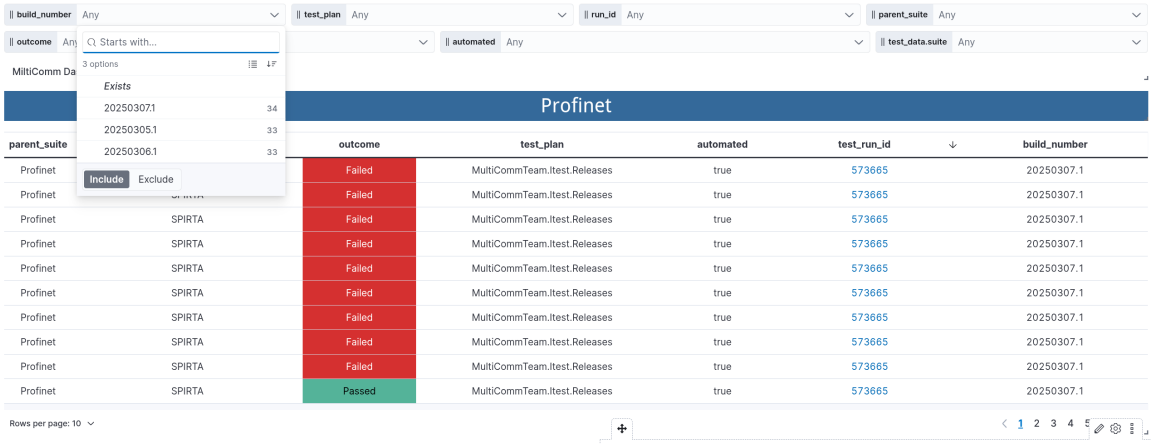
Pro konfiguraci tabulky je nutné přetáhnout pole *timestamp* do sekce *Rows* a pole kategorií chyb (*Approved*, *Committed*, *New*, *Resolved*) do sekce *Metrics*.

3.5.5 Vizualizace CI/CD testů

Tato vizualizace poskytuje přehled o výsledcích testů prováděných v prostředí Azure DevOps v rámci různých testovacích plánů. Umožňuje sledovat stav provedení jednotlivých testů, například *Passed*, *Failed* nebo *Aborted*, a porovnávat výsledky jednotlivých testovacích běhů (*runs*).

Klíčové sloupce tabulky

- **parent_suite** – název hlavního testovacího balíčku.
- **suite** – sada testů.
- **outcome** – výsledek testu:
  - **Červená** – Failed
  - **Zelená** – Passed
  - **Žlutá** – Aborted
- **test_plan** – identifikátor testovacího plánu.
- **automated** – označení, zda se jedná o automatický (*true*) nebo manuální (*false*) běh.
- **test_run_id** – ID běhu testovacího plánu.
- **build_number** – identifikátor běhu pipeline.
- **completed_date** – datum dokončení testu.



Obrázek 3.8: Vizualizace výstupů testovacích plánů

Postup konfigurace vizualizace

Konfigurace této vizualizace v *Kibana Lens* začíná přidáním polí do sekce *Rows*. Do této sekce se vkládají následující pole: parent_suite, suite, outcome, test_plan, automated a test_run_id.

Výsledky ve sloupci *outcome* jsou barevně rozlišeny pomocí dynamického režimu *Color mode: Dynamic*, kde úspěšné testy jsou označeny zelenou barvou (*Passed*), neúspěšné červenou (*Failed*) a přerušené testy žlutou barvou (*Aborted*). V sekci *Metrics* jsou přidána pole *build_number* a *completed_date*.

Pro snadný přístup k detailním informacím o jednotlivých testovacích bězích je pole *test_run_id* nastaveno ve formátu odkazu (URL). Použitá šablona vypadá následovně:

```
https://jupiter.tfs.siemens.net/IPS/IOGrid/_testManagement/
runs?_a=runCharts&runId={{value}}
```

Proměnná {{value}} v URL šabloně je při vykreslení automaticky nahrazena konkrétní hodnotou testovacího běhu, což umožňuje uživateli kliknutím přímo přejít na podrobnosti daného testu v Azure DevOps. Vizualizace také zahrnuje přednastavený filtr parent_suite = Profinet, který zobrazuje pouze testy patřící do této hlavní testovací sady.

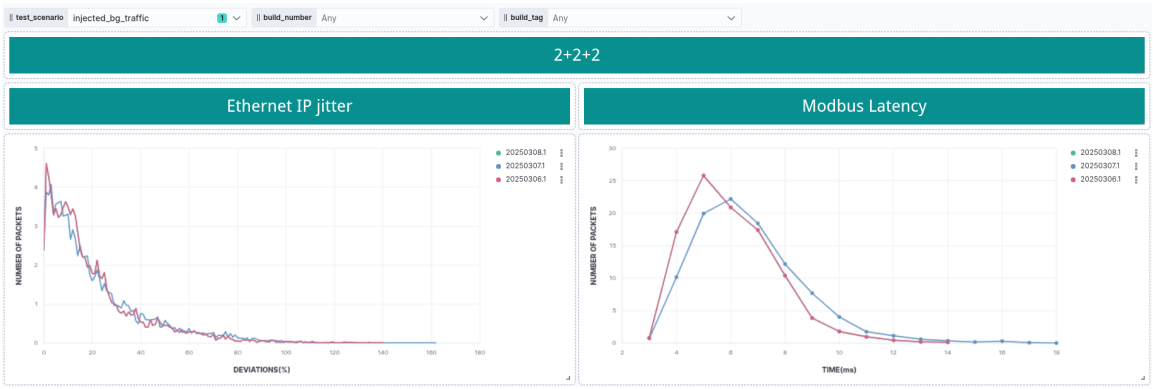
3.5.6 Vizualizace jitteru a latence

Tato vizualizace zobrazuje výsledky měření síťových charakteristik, latence a odchylky od předpokládané hodnoty periodicity zpráv (jitteru), pro komunikační protokoly *Ethernet/IP* a *Modbus TCP*.

Měření probíhá ve třech scénářích: se síťovým provozem na pozadí (*background traffic*), bez síťového provozu na pozadí (*no background traffic*) a se simulovaným síťovým provozem (*injected background traffic*).

Struktura vizualizace

- **Ethernet/IP** – čárový graf zobrazující hodnoty jitteru jednotlivých zpráv v čase, vypočtené na základě latence.
- **Modbus TCP** – čárový graf zobrazující přímé hodnoty latence při přenosu zpráv.



Obrázek 3.9: Grafické zobrazení latence a jitteru pro různé komunikační protokoly

Postup konfigurace vizualizace

Konfigurace této vizualizace zahrnuje nastavení pole *build_number* na horizontální osu, což umožňuje rozdělení dat podle čísla buildu. Na vertikální ose jsou zobrazeny hodnoty latence nebo jitteru, a to ve formě relativních hodnot vyjádřených v procentech vzhledem k celkovému počtu paketů.

Pro pohodlnější práci s dashboardem byla využita komponenta *Controls*, která poskytuje několik filtrů umožňujících omezit zobrazovaná data podle následujících kritérií:

- **testovacího scénáře (test_scenario)** – například se síťovým provozem na pozadí (background traffic) nebo bez něj.
- **čísla buildu (build_number)**,
- **build tagu (build_tag)**.

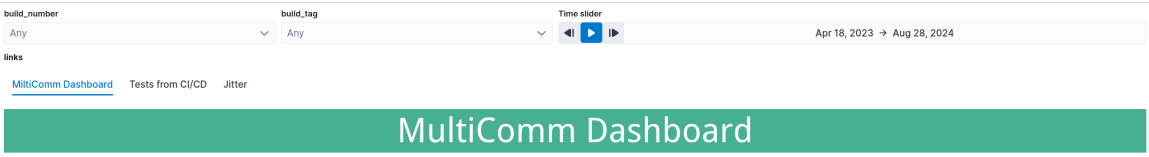
Vizualizace byla v nástroji *Lens* pevně nastavena podle:

- **typu měření (type.keyword)**, například typu 2+2+2, který odpovídá určité specifikaci zátěže nebo kombinaci testovaných podmínek,
- **názvu testovaného komunikačního protokolu (name.keyword)**, například *Ethernet IP* nebo *Modbus Latency*, aby vizualizace zobrazovala pouze relevantní výsledky pro konkrétní síťovou technologii.

Toto statické filtrování bylo nezbytné, protože všechna měřená data ze všech protokolů jsou ukládána do jednoho společného indexu. Přednastavené filtry zajišťují, že se v grafu vždy zobrazují pouze požadovaná data odpovídající danému typu měření a protokolu.

3.5.7 Přehled jednotlivých dashboardů a globální filtrování

Následující ukázka pochází z dashboardu MultiComm a zobrazuje navigační rozhraní v prostředí Kibana. Toto rozhraní umožňuje uživatelům filtrovat data a přepínat mezi jednotlivými dashboardy.



Obrázek 3.10: Navigace a globální filtrování v dashboardu Kibana

Dashboard obsahuje dva klíčové globální filtry:

- **build_number** - umožňuje uživateli vybrat číslo buildu a zobrazit pouze data vztahující se k danému buildu.
- **build_tag** - filtruje výsledky podle tagů, jako například *Release* a *Nightly*.

Po aplikaci filtru dochází k automatické aktualizaci všech tabulek, grafů a vizualizací na základě zvolených kritérií.

Doplňkové prvky dashboardu

Time Slider je dalším důležitým prvkem dashboardu, který umožňuje uživatelům dynamicky nastavovat časové rozmezí zobrazovaných dat. Časový rozsah lze definovat ručně nebo využít režim automatického posunu, označovaný jako *play mode*.

Markdown blok slouží jako vizuální prvek pro oddělení jednotlivých sekcí a přispívá ke zlepšení orientace uživatele v rámci dashboardu. V tomto případě slouží k

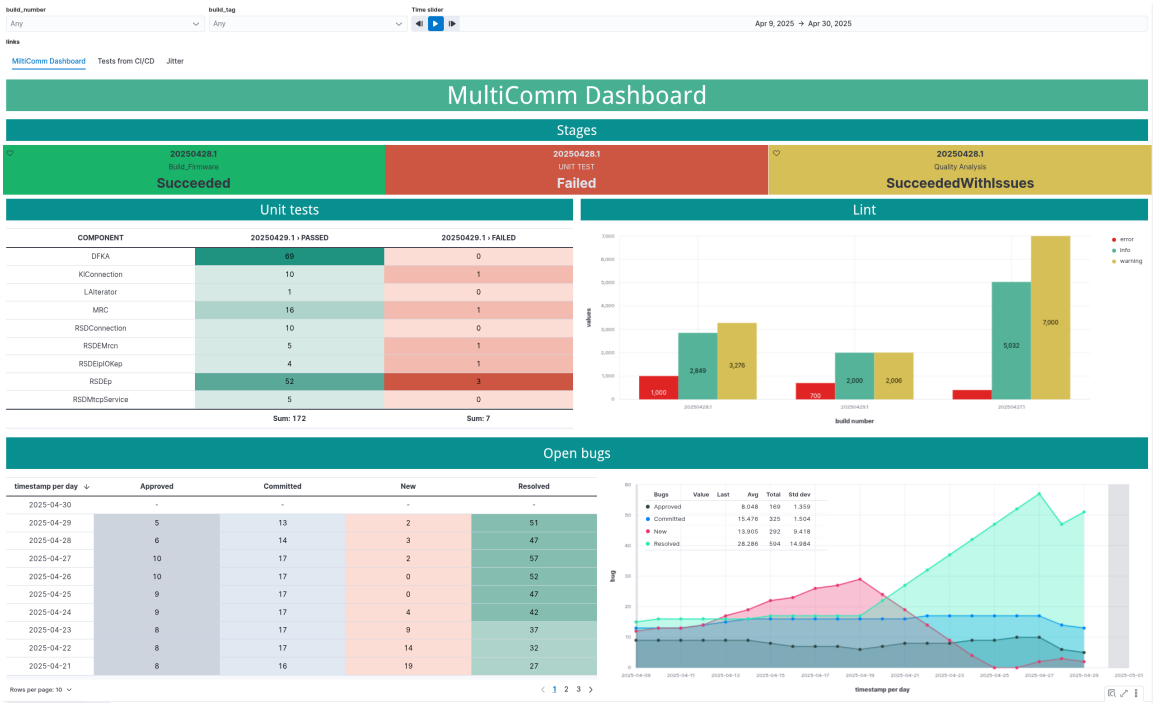
výraznému zobrazení nadpisu MultiComm Dashboard. V části **Panel options** byla nastavena vlastní úprava stylu pomocí CSS[10], kde byly specifikovány:

- Zarovnání textu na střed (text-align: center;)
- Barva textu (color: #FFF;)
- Velikost písma (font-size: 50px;)
- Použitý font (font-family: -apple-system, BlinkMacSystemFont, Segoe UI, Roboto, Oxygen, Ubuntu, Fira Sans, Droid Sans, Helvetica Neue, sans-serif;)

Finální dashboardy

V rámci řešení byly vytvořeny tři samostatné dashboardy, mezi kterými lze přepínat prostřednictvím sekce *Links* v prostředí Kibana.

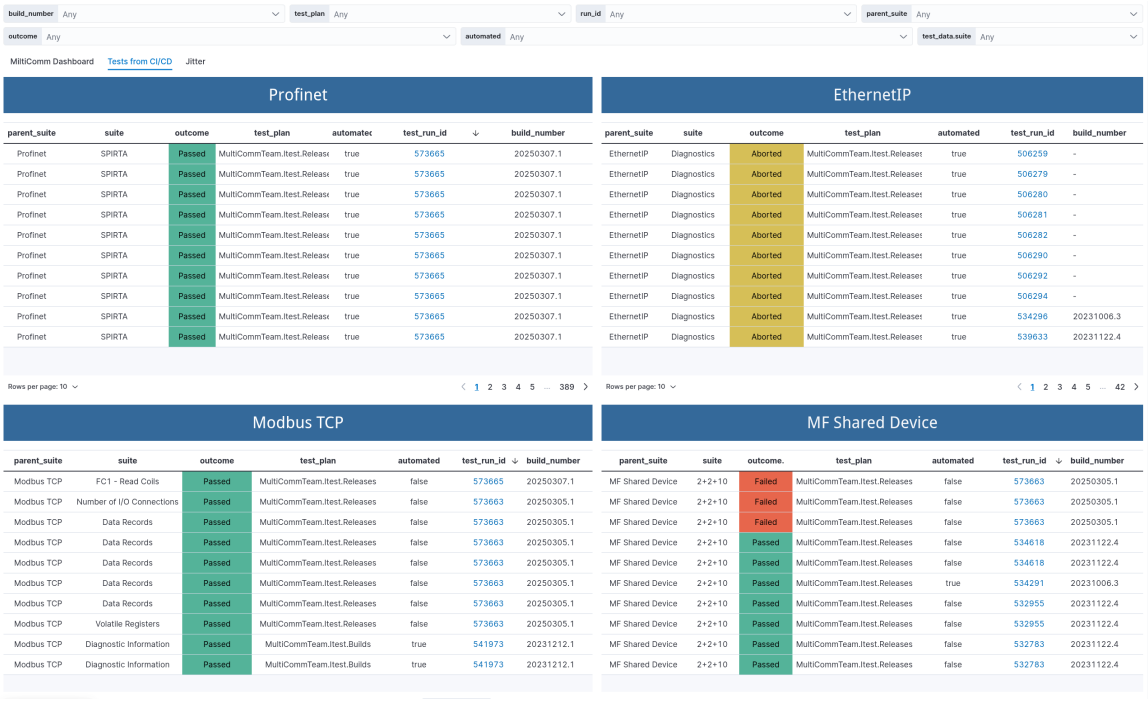
- **MultiComm Dashboard** – zastává roli hlavního dashboardu, který je sestaven ze sekcí: stav pipeline 3.5.1, výsledky unit testů 3.5.2, statická analýza kódu (Lint) 3.5.3 a vývoj otevřených chyb 3.5.4.



Obrázek 3.11: Ukázka Multicomm Dashboardu

[10]CSS (Cascading Style Sheets)

- **Tests from CI/CD Dashboard** – zobrazuje detailní výsledky testovacích plánů z prostředí Azure DevOps, které jsou popsány v sekci 3.5.5. Umožňuje analyzovat průběh testů pro jednotlivé komunikační protokoly.



Obrázek 3.12: Ukázka Tests from CI/CD Dashboardu

- **Jitter Dashboard** – vizualizuje síťové charakteristiky komunikačních protokolů, uvedené v sekci 3.5.6.

Obrázek 3.13: Ukázka Jitter Dashboardu

3.6 Zhodnocení implementace

V této kapitole byla popsána realizace dashboardového řešení. Na základě provedené analýzy dostupných technologií byly zvoleny nástroje jako Elasticsearch, Kibana, Docker a Azure DevOps.

Následně byla navržena a implementována architektura systému, která umožňuje automatizované zpracování dat pomocí Python skriptů v rámci pipeline. Veškerá data jsou ukládána do systému Elasticsearch a vizualizována prostřednictvím nástroje Kibana.

Přínosem této implementace je centralizovaný a přehledný pohled na klíčové metriky, který týmu umožňuje rychlou identifikaci problémů a sledování kvality vývoje firmwarové komponenty MultiComm. Významnou výhodou je rovněž možnost snadného rozšíření – dashboardy lze podle potřeby rozšiřovat o další tabulky, grafy nebo vizualizace.

3.7 Návrhy na zlepšení

Na základě popsané implementace lze doporučit následující návrhy, které by mohly dále zvýšit spolehlivost, bezpečnost a udržovatelnost systému:

Zajištění vysoké dostupnosti (High Availability)

V současné konfiguraci běží Elasticsearch na jednom uzlu, což představuje potenciální riziko nedostupnosti v případě jeho výpadku. Doporučuje se proto rozšířit architekturu o další uzel. To umožní nastavit replikaci dat mezi uzly a zajistit odolnost vůči výpadku jednotlivého uzlu. V případě výpadku primárního uzlu zůstane systém funkční, protože repliky dat budou dostupné na jiném uzlu. Tento přístup rovněž zvýší výkon, neboť zátěž se rozdělí mezi více uzlů.

Zvýšení bezpečnosti pomocí šifrované komunikace a autentizace

Doporučuje se aktivovat zabezpečený přístup pomocí protokolu HTTPS a aktivovat autentizaci přes LDAP[11] v systémech Elasticsearch a Kibana. Tím se zajistí šifrovaný přenos dat a možnost správy přístupových práv prostřednictvím existujících uživatelských účtů a skupin v LDAP. Tento způsob umožňuje centrální řízení oprávnění a přiřazování různých úrovní přístupu, například pouze pro čtení nebo pro správu dashboardů, bez nutnosti vytváření lokálních uživatelských účtů.

Správa životního cyklu dat (Index Lifecycle Management – ILM)

Dalším doporučeným krokem je aktivace a konfigurace správy životního cyklu indexů (ILM). Pomocí ILM lze automaticky mazat nebo archivovat zastaralé a neaktuální údaje po uplynutí stanovené doby. Tím se optimalizuje využití diskové kapacity a zároveň se zachová výkonnost systému i při dlouhodobém provozu.

[11] LDAP (Lightweight Directory Access Protocol)

ZÁVĚR

Cílem této bakalářské práce bylo navrhnout a implementovat systém dashboardů pro tým ve společnosti Siemens, který vyvíjí firmwarovou komponentu MultiComm. Práce se zaměřila na zavedení centralizovaného úložiště dat, automatizaci všech procesů pro sběr a zpracování dat a vytvoření přehledných dashboardů zobrazujících klíčové informace týkající se komponenty MultiComm.

V teoretické části byly představeny klíčové pojmy a technologie, které pomáhají lépe pochopit návrh a implementaci řešení. Byla zde rozebrána témata, jako jsou vývoj firmwarových systémů, práce s daty, kontejnerizace pomocí Dockeru, automatizace procesů prostřednictvím Azure DevOps a komunikace pomocí rozhraní REST API.

Po teoretické části následovala analýza současného stavu, která odhalila řadu nedostatků, jako je absence centralizovaného úložiště, složitost při vyhledávání informací, obtížné sledování trendů a chybějící vizualizační podpora pro zajištění lepšího přehledu o stavu vývoje komponenty. Na základě těchto zjištění byly definovány požadavky na nový systém, který by měl tyto problémy odstranit.

Implementace výsledného řešení využívá technologie Elasticsearch, Kibana, Docker a Azure DevOps. Byly vytvořeny kontejnery pro systémy Elasticsearch a Kibana, nakonfigurován lokální agent pro Azure DevOps a vyvinuty Python skripty pro automatizovaný sběr a zpracování dat, které se následně automaticky spouštějí pomocí pipeline v Azure DevOps. Na závěr byly vytvořeny dashboardy v prostředí Kibana, které umožňují týmu efektivně sledovat, porovnávat a filtrovat data vznikající v průběhu vývoje komponenty MultiComm. Navržené řešení je plně rozšiřitelné o další data a funkcionality podle potřeb týmu.

Literatura

[1] Athick, A. *Getting Started with Elastic Stack 8.0: Run powerful and scalable data platforms to search, observe, and secure your organization*. 1st. Packt Publishing, 2022. ISBN 9781800569492.

[2] Awati, R. *What is Firmware?* [online]. TechTarget, 2024 [cit. 2025-04-18]. Dostupné z: https://www.techtarget.com/whatis/definition/firmware.

[3] Beer, L. de. *Understanding GitLab: What is GitLab and How is it Used?* [online]. Deimos, 2023 [cit. 2025-04-18]. Dostupné z: https://www.deimos.io/blog-posts/understanding-gitlab-what-is-gitlab-and-how-is-it-used.

[4] Carasso, D. *Exploring Splunk: Search Processing Language (SPL) Primer and Cookbook* [online]. Splunk Inc., 2012 [cit. 2025-04-18]. Dostupné z: https://www.splunk.com/en_us/pdfs/ebooks/exploring-splunk.pdf.

[5] Gaitatzis, T. *Learn REST APIs: Your guide to how to find, learn, and connect to the REST APIs that powers the Internet of Things revolution*. 1st. BackupBrain Press, 2019. ISBN 9781989775004.

[6] Gormley, C. a Tong, Z. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. 1st. O'Reilly Media, 2015. ISBN 9781449358549.

[7] Hernández, R. J. *Building IoT Visualizations using Grafana: Power up your IoT projects and monitor with Prometheus, LibreNMS, and Elasticsearch*. 1st. Packt Publishing, 2022. ISBN 9781803236124.

[8] Humble, J. a Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1st. Addison-Wesley, 2010. ISBN 9780321601919.

[9] JFrog. *JFrog Artifactory - Get Started with the JFrog Platform* [online]. JFrog, 2025 [cit. 2025-04-18]. Dostupné z: https://jfrog.com/help/r/get-started-with-the-jfrog-platform/jfrog-artifactory.

[10] Kunta, S. *Virtualization and Containerization: How to Create a Safe, Efficient Solution with Both Technologies* [online]. IEEE Computer Society,

2023 [cit. 2025-04-18]. Dostupné z: https://www.computer.org/publications/tech-news/trends/virtualization-and-containerization.

[11] Meier, A. a Kaufmann, M. *SQL & NoSQL Databases: Models, Languages, Consistency Options, and Architectures for Big Data Management*. 1st. Springer Vieweg, 2019. ISBN 9783658245481.

[12] Microsoft. *Deploy an Azure Pipelines agent on Linux* [online]. Microsoft, 2024 [cit. 2025-04-18]. Dostupné z: https://learn.microsoft.com/en-us/azure/devops/pipelines/agents/linux-agent?view=azure-devops.

[13] Miell, I. a Sayers, A. H. *Docker in Practice*. 2nd. Manning Publications, 2019. ISBN 9781617294808.

[14] OpenAI. *ChatGPT (GPT-4 model)*. 2023 [cit. 2025-05-12]. Online large language model využit k opravení gramatických chyb. Dostupné z: https://chat.openai.com/.

[15] Roberts, L. *Qualitative vs Quantitative Data: What's the Difference?* [online]. TechnologyAdvice, 2023 [cit. 2025-04-18]. Dostupné z: https://technologyadvice.com/blog/information-technology/qualitative-vs-quantitative/.

[16] Srivastava, A. a Azarmi, B. *Learning Kibana 7: Build Powerful Elastic Dashboards with Kibana's Data Visualization Capabilities*. 2nd. Packt Publishing, 2019. ISBN 9781838556556.

[17] White, E. *Making Embedded Systems: Design Patterns for Great Software*. 1st. O'Reilly Media, 2011. ISBN 9781449302146.