

Monitoring komplexních GitHub repozitářů

Monitoring of Complex GitHub Repositories

Bc. Martin Korotwitschka

Diplomová práce

Vedoucí práce: Ing. Jakub Beránek PhD.

Ostrava, 2026

Zadání diplomové práce

Student:

Bc. Martin Korotwitschka

Studijní program:

N0613A140034 Informatika

Téma:

Monitoring komplexních GitHub repozitářů
Monitoring of Complex GitHub Repositories

Jazyk vypracování:

čeština

Zásady pro vypracování:

Cílem práce je vytvořit nástroj, který umožní sledovat aktivitu rozsáhlých GitHub repozitářů, jako je např. překladač Rustu či Linuxový kernel. Nástroj by měl využívat API GitHubu, git historii daného repozitáře a případně další zdroje dat, aby umožnil svým uživatelům pochopit, co se v daném repozitáři děje. Měl by umožňovat provádět dotazy např. pro zjištění, kteří přispěvatelé často modifikují nebo provádí code review v konkrétních oblastech kódu, jak dlouho čekají pull requesty na code review apod. Nástroj by měl umožňovat sledovat repozitář v průběhu času, a efektivně ukládat data repozitáře ve vlastní databázi. Nástroj by měl poskytovat přehlednou a interaktivní vizualizaci těchto údajů, ideálně ve formě webové aplikace. Zároveň by jej ale mělo být možné použít i jako knihovnu, aby šel použit v rámci existujících systémů pro monitoring repozitářů. Z toho důvodu by také měl být napsán v jazyku, který umožňuje jednoduché znovuvyužití napříč programovacími jazyky, např. v Rustu. Výsledný kód by měl být řádně zdokumentován a při jeho vývoji by měl být využit verzovací systém (např. git).

1. Nastudujte datový model GitHub repozitářů a API nabízené GitHubem.
2. Navrhněte strukturu databáze, která bude použita pro agregaci metadata repozitáře z různých zdrojů dat.
3. Naimplementujte sadu dotazů pro získání zajímavých informací o repozitáři.
4. Vytvořte rozhraní pro interaktivní vizualizaci získaných dat.
5. Otestujte nástroj na netriviálním GitHub repozitáři, např. překladači Rustu.

Seznam doporučené odborné literatury:

- [1] LOELIGER, Jon. Version Control with Git: Powerful tools and techniques for collaborative software development. 2012. O'Reilly Media. ISBN: 978-1449316389
- [2] KLABNIK, Steve and NICHOLS, Carol. The Rust Programming Language First Edition. 2018. No Starch Press. ISBN: 978-1593278281

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jakub Beránek**

Datum zadání: 01.09.2025

Datum odevzdání: 30.04.2026

Garant studijního programu: prof. RNDr. Václav Snášel, CSc.

V IS EDISON zadáno: 09.10.2025 11:05:04

Abstrakt

Hlavním cílem této práce je návrh a implementace nástroje pro sledování komplexních GitHub repozitářů. Motivací je nedostatek existujících nástrojů, které by efektivně pokrývaly potřeby rozsáhlých open source projektů. V teoretické části je provedena analýza datového modelu GitHub repozitářů a dostupných rozhraní. Praktická část se věnuje návrhu relační databáze pro agregaci dat z GitHub API, lokální Git historie a konfiguračních souborů týmů, a následné implementaci. Nad navrženou databází byla implementována sada analytických dotazů a interaktivní webové rozhraní pro vizualizaci výsledků. Výsledný nástroj byl otestován na datech repozitáře překladače Rustu, který čítá více než 200 000 issues a pull requestů.

Klíčová slova

git; github; rust; monitoring; analýza repozitáře; pull requesty; databáze

Abstract

The main goal of this thesis is the design and implementation of a tool for monitoring complex GitHub repositories. The motivation stems from the lack of existing tools that effectively cover the needs of large open source projects. The theoretical part analyzes the data model of GitHub repositories and the available interfaces. The practical part covers the design of a relational database for aggregating data from the GitHub API, local Git history, and team configuration files, followed by the system implementation. A set of analytical queries and an interactive web interface for result visualization were built on top of the designed database. The resulting tool was evaluated on data from the Rust compiler repository, which contains more than 200 000 issues and pull requests.

Keywords

git; github; rust; monitoring; repository analysis; pull requests; database

Poděkování

Chtěl bych poděkovat především svému vedoucímu práce Jakubu Beránkovi za cenné rady, a obecně jeho vedení obzvlášť ohledně praktické části práce. Dále bych rád poděkoval všem, kteří mi poskytli zpětnou vazbu k různým částem této práce, a v neposlední řadě mým blízkým za podporu během celého procesu psaní.

Obsah

Seznam obrázků	8
Seznam tabulek	9
Seznam výpisů zdrojového kódu	10
1 Úvod	13
2 Verzování	15
2.1 Verzování softwaru	15
2.2 Apache Subversion	17
2.3 Git	18
3 GitHub	23
3.1 Úvod do GitHubu	23
3.2 Základní funkce a model spolupráce	23
3.3 Application Programming Interface (API) GitHubu	27
4 Analýza existujících nástrojů	36
4.1 Nativní statistiky GitHubu	36
4.2 Curl Dashboard	37
4.3 Rustc PR Tracking	38
4.4 Shrnutí analýzy	38
5 Repozitář Rust překladače	40
5.1 Rust Team Application Programming Interface (API)	40
5.2 Štítky	41
5.3 Proces sloučení pull requestů	42
6 Návrh systému	44
6.1 Požadavky na systém	44

6.2	Hlavní návrhová rozhodnutí	47
6.3	Architektura systému	51
6.4	Volba databázového systému	53
6.5	Struktura databáze	55
6.6	Návrh dotazů	57
7	Implementace	59
7.1	Popis aplikace	59
7.2	Implementace databázového schématu	61
7.3	Specifika synchronizace dat	63
7.4	Zobrazování dat	65
8	Evaluace	71
8.1	Výkonnostní měření synchronizace	71
8.2	Měření výkonu analytických dotazů	73
9	Závěr	76
	Literatura	78
	Přílohy	80
A	Detailní architektura systému	81
B	Podrobný ER diagram databáze	83
C	Analytické SQL dotazy a indexy	85
C.1	Analytické dotazy	85
C.2	Databázové indexy	88
C.3	Výstupy EXPLAIN ANALYZE	91

Seznam obrázků

2.1	Příklad historie projektu řízené verzovacím systémem [2].	16
2.2	Porovnání centralizované a distribuované architektury verzovacích systémů [2]. . . .	17
2.3	Objektový model Gitu [4].	21
3.1	Typický průběh práce s forkem na GitHubu [5].	25
6.1	Celková architektura systému	52
6.2	Celkový přehled databázového schématu	56
7.1	Zobrazení vývoje počtu Pull Request (PR) ve stavu „Waiting for Review“ za posledních 365 dní	65
7.2	Souhrn všech stavů Pull Request (PR) za posledních 365 dní — plné zobrazení . . .	66
7.3	Souhrn stavů Pull Request (PR) po skrytí monotónně rostoucích křivek — detail Pull Request (PR) čekajících na akci	67
7.4	Soubory upravené týmem — plochý seznam	68
7.5	Soubory upravené týmem — treemap	69
7.6	Surová JavaScript Object Notation (JSON) struktura stromové hierarchie souborů .	70
A.1	Detailní architektura systému	82
B.1	Podrobný ER diagram databáze	84

Seznam tabulek

6.1	Srovnání vlastností SQLite a PostgreSQL pro účely analytického nástroje	55
7.1	Klíčové závislosti backendu aplikace	60
8.1	Časy jednotlivých fází počáteční synchronizace (rust-lang/rust, od 2023-01-01) . . .	72
8.2	Srovnání výkonu analytických dotazů před a po indexaci (průměr z 5 opakování) . .	75

Seznam výpisů zdrojového kódu

3.1	Ukázka Representational State Transfer (REST) Application Programming Interface (API) požadavku na GitHub pro získání událostí celého GitHubu.	27
3.2	Ukázka odpovědi REST API na požadavek z Výpisu 3.1 (zkráceno).	28
3.3	Ukázka GraphQL dotazu na otevřené issues repozitáře.	30
3.4	Zkrácená odpověď na GraphQL dotaz z Výpisu 3.3.	30
3.5	Ukázka korektního GraphQL dotazu.	32
3.6	Výpočet celkového limitu dotazu (Node limit)	32
3.7	Ukázka komplexního GraphQL dotazu	33
3.8	Výpočet celkového limitu komplexního dotazu (Node limit)	34
8.1	Log inkrementální aktualizace bez nových záznamů	73
C.1	S1 — Počet Pull Request (PR) ve stavu (štítek) k zadanému datu	85
C.2	S2 — Kumulativní počet sloučených PR	86
C.3	S3 — Soubory upravené přispěvateli týmu	86
C.4	S4 — Denní vývoj počtu PR v daném stavu	87
C.5	Indexy tabulky issues	88
C.6	Indexy tabulky issue_event_history	89
C.7	Indexy tabulky issue_labels_history	90
C.8	Indexy tabulky file_activity	91
C.9	Indexy tabulky contributors	91
C.10	Indexy tabulky contributors_teams	91
C.11	S1: EXPLAIN ANALYZE bez indexů (Execution Time: 73,5 ms)	92
C.12	S1: EXPLAIN ANALYZE s indexy (Execution Time: 20,0 ms)	93
C.13	S2: EXPLAIN ANALYZE bez indexů (Execution Time: 18,8 ms)	94
C.14	S2: EXPLAIN ANALYZE s indexy (Execution Time: 2,7 ms)	94
C.15	S3: EXPLAIN ANALYZE bez indexů (Execution Time: 50,6 ms)	95
C.16	S3: EXPLAIN ANALYZE s indexy (Execution Time: 41,0 ms)	96
C.17	S4: EXPLAIN ANALYZE bez indexů (Execution Time: 111,4 ms)	97
C.18	S4: EXPLAIN ANALYZE s indexy (Execution Time: 74,6 ms)	99

Seznam zkratek

API Application Programming Interface

CD Continuous Deployment

CI Continuous Integration

CI/CD Continuous Integration/Continuous Deployment

CLI Command Line Interface

CTE Common Table Expression

CVS Concurrent Versions System

FS File System

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JSON JavaScript Object Notation

ORM Object-Relational Mapping

PAT Personal Access Token

PR Pull Request

REST Representational State Transfer

SHA Secure Hash Algorithm

SPA Single-Page Application

SQL Structured Query Language

SVN Subversion

TCP/IP Transmission Control Protocol/Internet Protocol

URL Uniform Resource Locator

VCS Version Control System

WAL Write-Ahead Logging

Kapitola 1

Úvod

Distribuované verzovací systémy, zejména systém Git, umožňují vývojářům asynchronně spolupracovat ve sdíleném repozitáři. Kolem nich vznikly platformy jako GitHub, které dnes neslouží pouze jako úložiště zdrojových kódů, ale integrují i správu revizí kódu, sledování chyb a Continuous Integration/Continuous Deployment (CI/CD) procesy.

Nativní analytické nástroje platformy GitHub dostačují pro malé a středně velké projekty, ale u velkých open source repozitářů narážejí na své limity. Příkladem je repozitář překladače jazyka Rust (`rust-lang/rust`), kde spolupracují stovky vývojářů organizovaných do týmových hierarchií, otevírají se desítky Pull Request (PR) denně a stejně tak uzavírají. Projekt využívá vlastní stavové automaty řízené štítky a striktní pravidla schvalování vynucovaná automatizovanými boty. Zjistit například, jak dlouho průměrně čeká kód na revizi od konkrétního týmu, nebo identifikovat stagnující PR, je pomocí standardních nástrojů obtížné.

Cílem této diplomové práce je proto návrh a implementace analytického nástroje, který tyto informační mezery překlenuje. Navržený systém stahuje, normalizuje a analyzuje data ze tří zdrojů: z verzovacího systému Git, z Application Programming Interface (API) platformy GitHub a z externích organizačních struktur (Team API). Nástroj je implementován v jazyce Rust s ohledem na výkon při zpracování velkých objemů dat.

Práce nejprve teoreticky analyzuje mechanismy pro získávání dat z platformy GitHub, konkrétně Representational State Transfer (REST) API, GraphQL API a Webhooky, a popisuje specifika procesního řízení ve velkých open source repozitářích. Součástí je také zhodnocení nativních možností monitoringu platformy GitHub i specializovaných nástrojů třetích stran a identifikace jejich nedostatků v kontextu masivních projektů. Na základě těchto zjištění je navržena modulární softwarová architektura a databázový model schopný ukládat historická i inkrementální data. Výsledné řešení je vyhodnoceno na reálných datech z repozitáře `rust-lang/rust` prostřednictvím sady analytických dotazů zaměřených na hlubší porozumění procesu revize kódu.

Teoretická část práce popisuje principy verzovacích systémů, API platformy GitHub a obsahuje analýzu existujících monitorovacích nástrojů. Navazuje kapitola o návrhu systému s definicí

požadavků, návrhem architektury a strukturou databáze. Implementační kapitola se věnuje technickým specifikům vývoje analytického nástroje. Práce je zakončena kapitolou o vyhodnocení, která prezentuje výsledky měření a analytická data ze zkoumaného repozitáře.

Kapitola 2

Verzování

Tato kapitola se věnuje verzování softwaru a jeho významu pro vývoj a údržbu projektů. Dále je popsán konkrétní verzovací nástroj Git, jeho funkce a programovatelný přístup k jeho datům.

2.1 Verzování softwaru

Verzování je proces správy a sledování změn v projektu [1]. V tomto procesu se uchovává kompletní historie veškerých změn souborů, které jsou verzovány. Nejčastěji se takového nástroje používají u textových zdrojových kódů softwaru, avšak lze je využít pro libovolné typy souborů.

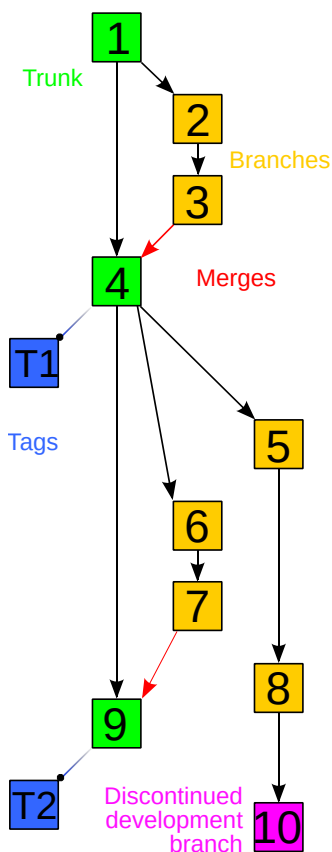
Version Control System (VCS) obvykle eviduje, kdo změnil jaký soubor a co konkrétního v tom souboru změnil. Díky tomu lze kdykoliv dohledat, kdo napsal jakou část kódu, případně proč byla tato část změněna [2]. Na základě těchto informací lze u VCS snadno vrátit zpět změny, které se ukázaly jako nevhodné.

VCS řeší také problém správy verzí, kdy před jeho zavedením bylo nutné uchovávat mnoho kopií téhož projektu s různými popisky identifikujícími jejich verzi, což vyžadovalo nadměrnou disciplínu a výrazně více úložného prostoru. U VCS jsou zpravidla ukládány pouze skutečné změny oproti předchozí verzi, nikoli celá kopie projektu, přičemž vše je spravováno interně daným nástrojem a uživateli tak odpadá starost s organizací a ukládáním jednotlivých verzí.

Další vlastností VCS je možnost souběžné práce více uživatelů na stejných souborech. VCS umožňuje řešit konflikty (tzv. *merge konflikty*), které vznikají při souběžné úpravě stejného souboru více uživateli [2].

Na Obrázku 2.1 je znázorněn příklad typické struktury historie projektu řízeného verzovacím systémem. Základní ideální struktura je lineární graf, tj. řada uzlů za sebou, kde každý další vychází z předchozího. Projekt má hlavní větev (Trunk), z níž se dále větví ostatní větve (Branches), například pro vývoj nové funkcionality. Větve se poté slučují zpět (Merges) a vznikají nové verze projektu. Z pohledu teorie grafů se jedná o acyklický orientovaný graf. Může se stát, že nově vyvíjená funkcionality je nevyhovující, a tak vzniká slepá větev (Discontinued development branch). Jakmile

je dosažen požadovaný stav v hlavní větvi, provádí se tzv. tagování (Tags). Tag většinou označuje novou verzi projektu nebo jiný důležitý milník.



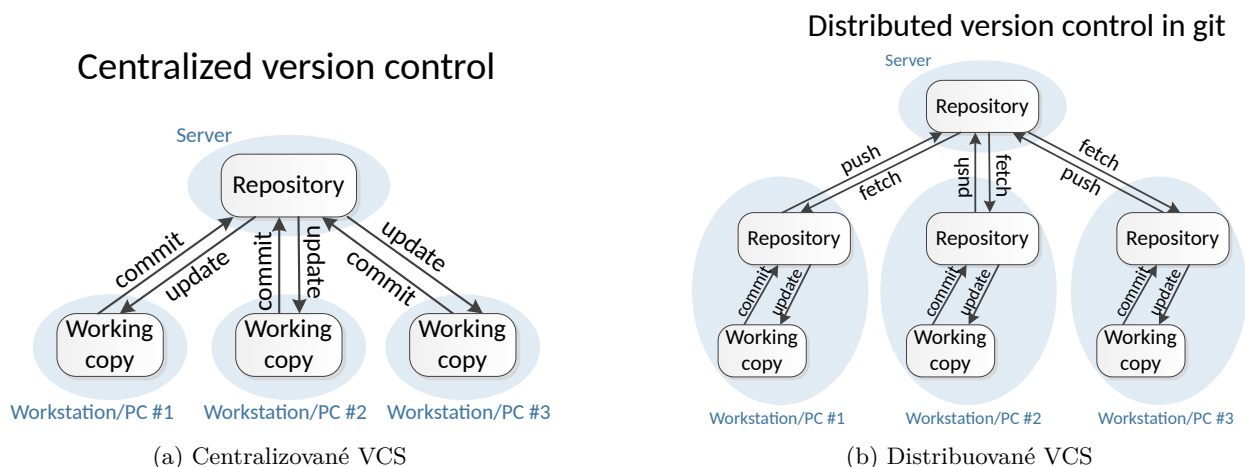
Obrázek 2.1: Příklad historie projektu řízené verzovacím systémem [2].

2.1.1 Centrální vs. distribuované VCS

Hlavní rozdíl mezi centrálními a distribuovanými VCS spočívá v tom, že distribuované VCS uchovávají kopii kompletní historie změn jak u každého uživatele, tak i na hostovacím serveru [2].

Na Obrázku 2.2 jsou zobrazeny oba typy architektury. U centrálního VCS se musí spoléhat výhradně na hostovací server. Pokud ten vypadne, v lepším případě uživatelé nemohou publikovat své změny, v horším případě při absenci zálohy dojde ke ztrátě celé historie. U distribuovaného VCS stačí, aby alespoň jeden uživatel měl projekt k dispozici, protože každý uživatel s lokální kopií projektu disponuje prakticky plnohodnotnou zálohou s kompletní historií a jednoduše se může projekt obnovit nebo znovu nasadit na jakoukoli jinou hostovací službu.

Rozdíl je patrný i v principu odesílání změn. U centrálního VCS se každá změna zapisuje přímo na server, zatímco u distribuovaného VCS se změna nejprve zaznamená do lokálního repozitáře a teprve následně se propaguje na hostovací službu. Tedy u centrálního VCS je vyžadováno připojení



Obrázek 2.2: Porovnání centralizované a distribuované architektury verzovacích systémů [2].

k serveru pro publikování změn, zatímco u distribuovaného není vyžadováno a změny na server lze zapsat později.

Rozdíl je patrný i v principu řešení konfliktů a slučování větví. Centralizovaný VCS umožňuje provést aktualizaci ze serveru, i když má uživatel rozpracované, dosud nezapsané soubory, a systém automaticky smíchá všechny změny, což může vést ke konfliktům. Distribuovaný VCS uživatele nejprve donutí zaznamenat změny do lokální historie a teprve poté lze provést sloučení mezi dvěma pevnými body historie, z nichž vzniká nový pevný bod.

2.2 Apache Subversion

Apache Subversion, neboli taky Subversion (SVN) je VCS, který vznikl v roce 2004 jako náhrada za Concurrent Versions System (CVS) [3]. Jedná se o open source centralizovaný verzovací systém.

Jeho hlavní implementační specifika jsou: atomické revize a transakce, větvení pomocí *levných kopií* (Cheap copies), verzování adresářů a metadat, podpora binárních souborů, podpora neslučitelných souborů a přístup k datům přes Hypertext Transfer Protocol (HTTP), Hypertext Transfer Protocol Secure (HTTPS) a protokol SVN [3].

Globální revize a transakce

SVN používá globální revize napříč všemi soubory. Revize je snímek stavu repozitáře zaznamenaný v okamžiku zápisu změn. U starších systémů (např. CVS) byly revize vedeny samostatně pro každý soubor, přičemž každý soubor měl vlastní historii a číslo revize.

Každý zápis do repozitáře je v SVN atomický a je zaznamenan jako jedna revize zahrnující všechny dotčené soubory. To znamená, že zápis se buď provede celý, nebo se neprovede vůbec. Každá revize má jeden globální identifikátor, který jednoznačně reprezentuje stav celého repozitáře [3].

Větvení pomocí levných kopií

Větvení v SVN nemá žádnou speciální interní reprezentaci. Větvě jsou realizovány na úrovni adresářové struktury uživateli (typicky `/branches`, `/tags`, `/trunk`).

Větvení se provádí příkazem `svn copy`. Interně nedochází k fyzické duplikaci dat, ale vytvoří se pouze symbolický odkaz na původní data v dané revizi. Tvorba nové větve je tímto způsobem velice rychlá [3].

Verzování adresářů a metadat

SVN odstraňuje nedostatek starších systémů, které verzovaly pouze obsah souborů, nikoli strukturu adresářů.

Přesuny, přejmenování a kopírování souborů jsou verzovány a zachovávají návaznost na historii souboru. SVN dále umožňuje připojit k adresáři metadata ve formátu klíč:hodnota (např. `svn:ignore`, `svn:externals`), přičemž tato metadata jsou verzována stejně jako samotný obsah souborů [3].

Podpora libovolných typů souborů

Systém byl navržen tak, aby efektivně pracoval i se soubory jinými než textovými. U binárních souborů jsou ukládány pouze změny oproti předchozí revizi, nikoli celá kopie souboru, čímž se předchází neúměrnému nárůstu velikosti repozitáře [3].

Uzamykání neslučitelných souborů

Při vývoji se může stát, že pracujeme se soubory, které algoritmicky sloučit nelze, jako jsou například obrázky nebo zkompilevané binární soubory. SVN umožňuje takovéto soubory uzamknout na úrovni serveru, aby je ostatní uživatelé nemohli přepisovat, dokud není tento zámek opět uvolněn.

Přístup k datům

SVN nabízí flexibilní metody přístupu jak přes HTTP/HTTPS obohacené o verzovací funkce nebo přes vlastní protokol svserve běžící nad Transmission Control Protocol/Internet Protocol (TCP/IP) [3], který je technicky jednodušší a rychlejší než HTTP/HTTPS.

2.3 Git

Git je open source distribuovaný VCS vytvořený v roce 2005 Linusem Torvaldsem primárně pro správu zdrojového kódu jádra operačního systému Linux. Vychází z komerčního VCS BitKeeper a zakládá si na několika klíčových požadavcích [4]:

- Rychlost
- Jednoduchý návrh
- Silná podpora nelineárního vývoje pro tisíce paralelních větví
- Plně distribuovaný charakter
- Schopnost efektivně spravovat velké projekty, jako je jádro operačního systému Linux

Oproti ostatním VCS, které uchovávají historii jako seznam změn, Git uchovává historii jako sérii snímků (snapshots). Záznam takového snímku se nazývá commit. Tento termín označuje jak konkrétní bod v historii projektu, tak i samotnou operaci jeho vytvoření. Při každém commitu se uloží kompletní stav všech souborů, nikoli pouze jejich rozdíly. Pokud se soubor oproti předchozímu snímku nezměnil, uloží se pouze odkaz na jeho předchozí verzi. Tímto přístupem je dosaženo efektivního větvení a slučování [4].

V základu pracuje Git se třemi stavy souborů:

Modifikované (modified) soubor byl změněn, ale zatím nebyl uložen do historie.

Připravené (staged) soubor je označen pro zahrnutí do dalšího commitu.

Zapsané (committed) změny jsou zaznamenány v lokální historii.

Z těchto stavů vyplývají i hlavní oblasti projektu na disku [4]:

Pracovní adresář (Working directory) obsahuje lokální kopii jedné verze projektu. Tyto soubory jsou vytaženy z lokální databáze Gitu.

Oblast připravených změn (Staging area) soubor, v němž jsou uloženy informace o tom, co bude obsahovat příští commit.

Adresář Gitu (Git directory neboli `.git`) obsahuje historii a metadata projektu.

Kromě synchronizace dat se serverem jsou v Gitu všechny operace lokální, čímž je dosažena vysoká rychlost a možnost plně pracovat odkudkoliv bez nutnosti být připojen k síti [4].

Než se v Gitu cokoli uloží, spočítá se nejprve kontrolní součet (checksum) pomocí algoritmu Secure Hash Algorithm (SHA)-1, který se pak používá k odkazování na uložená data. Tímto způsobem je zaručeno, že nelze provést žádnou operaci, o které by Git nevěděl, a data jsou tak vždy konzistentní [4].

Při jakékoliv běžné operaci v Gitu jsou data do databáze pouze přidávána, nikdy mazána. Ke ztrátě zapsaných dat může dojít jedině při záměrném použití pokročilých operací, které mění historii repozitáře. Díky tomu lze bez obav provádět rozsáhlé experimenty nebo změny, protože k dříve zapsanému stavu se lze vždy vrátit [4].

2.3.1 Úložiště adresované obsahem (Content-addressable storage)

Git databáze je fundamentálně založena na principu souborového systému s adresovatelným obsahem jako úložiště klíč–hodnota, kde klíčem je hash obsahu a hodnotou je samotný soubor. Tento klíč je při každém commitu vypočítáván a vrácen pro pozdější použití. Klíč je generován pomocí algoritmu SHA-1, který vytvoří 40znakový řetězec z obsahu souboru [4].

Fyzicky jsou tyto objekty ukládány do složky `.git/objects`. V této složce jsou podsložky pojmenované prvními dvěma znaky hashe a v těchto podsložkách jsou zkomprimované soubory s daty pojmenované zbylými 38 znaky hashe [4]. Například objekt s hashem

`e965c8f1a2b3c4d5e6f7g8h9i0j1k2l3m4n5o6` bude uložen jako
`.git/objects/e9/65c8f1a2b3c4d5e6f7g8h9i0j1k2l3m4n5o6`.

2.3.2 Objektový model Gitu

Git používá tři hlavní typy objektů pro ukládání dat:

Bloby (Binary Large Objects) Představují nejnižší úroveň uložení dat v Gitu. Blob obsahuje pouze obsah souboru bez názvu původního souboru nebo jakýchkoliv metadat.

Stromy (Tree Objects) Mají na starost názvy souborů a jejich hierarchii, čímž simulují jednoduchý File System (FS) operačního systému. Strom se skládá z jednoho či více záznamů, které obsahují název souboru, SHA-1 hash blobu, typ objektu, na který odkazuje (blob nebo strom), a oprávnění. Díky této struktuře je jednoduché zachytit snímek celého projektu.

Commity (Commit Objects) I když stromový objekt zachycuje snímek projektu, neobsahuje kontext o tom, kdo tento stav uložil, kdy a z jakého důvodu.

Každý commit obsahuje tato metadata:

Odkaz na kořenový strom (root tree) Hash stromu, který reprezentuje snímek projektu v okamžiku commitu.

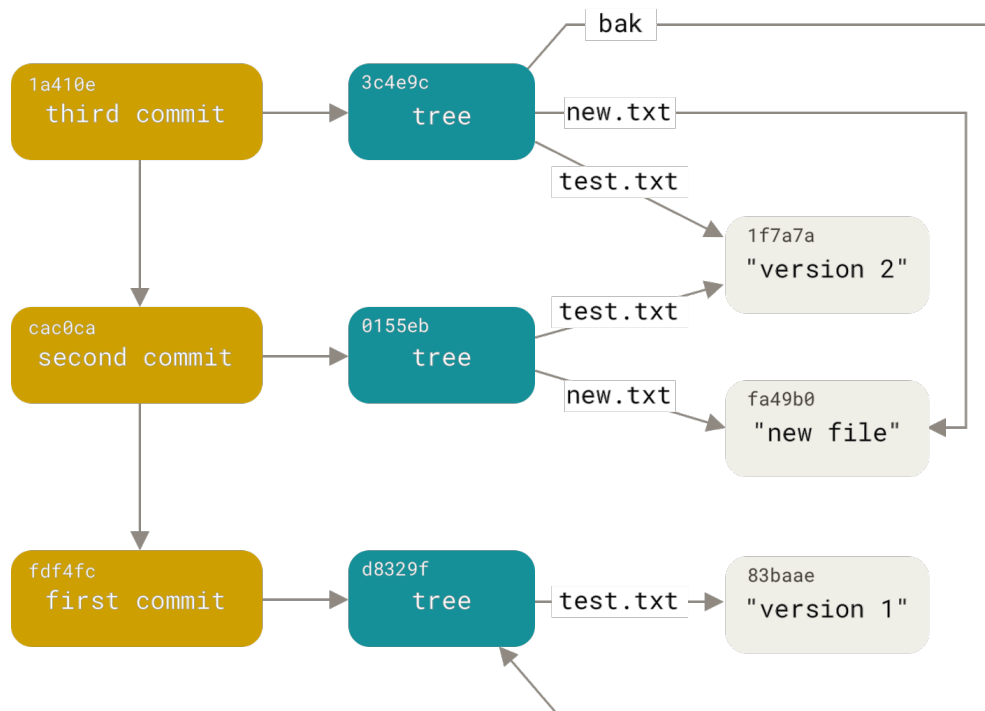
Odkazy na rodičovské commity Hashe předchozích commitů. První commit nemá žádného rodiče, běžné commity mají jednoho rodiče a merge commity mají dva nebo více rodičů.

Metadata o autorovi Jméno, e-mail a časové razítko osoby, která vytvořila danou změnu.

Metadata committera Jméno, e-mail a časové razítko osoby, která změnu fakticky uložila do repozitáře.

Zpráva (commit message) Krátký popis změn, které commit obsahuje.

Na Obrázku 2.3 je zobrazena kompletní interní struktura repozitáře složená ze všech typů objektů. Oranžové objekty jsou Commit objects (2.3.2), zelené jsou kořeny stromových objektů (2.3.2) [4].



Obrázek 2.3: Objektový model Gitu [4].

Větve jako pohyblivé ukazatele

Dřívější VCS, jako například SVN, přistupovaly k větvím jako k fyzickým kopiím dat. To znamená, že pro vytvoření nové větve bylo nutné zkopírovat celý projekt do nové složky, což bylo prostorově i časově náročné [4].

V Gitu jsou naproti tomu větve pouze pohyblivé ukazatele směřující na konkrétní commit. Fyzicky se jedná o jeden textový soubor ve složce `.git/refs/heads/`, který obsahuje pouze hash commitu. Vytvoření větve tedy znamená vytvoření nového souboru s hashem commitu, což je velice rychlé bez ohledu na velikost projektu [4].

Aby Git dokázal určit, na jaké větvi se uživatel právě nachází, využívá speciální ukazatel zvaný HEAD. Tento ukazatel je textový soubor uložený v `.git/HEAD`, který namísto přímého hashu obsahuje symbolickou referenci na aktuální větev. Když uživatel vytvoří nový commit, Git přečte HEAD, zjistí aktuální větev a posune její ukazatel tak, aby odkazoval na nově vzniklý commit [4].

Údržba a komprese dat

Se skutečností, že Git při každém commitu ukládá celé snímky projektu jako plné bloby, může vyvstát otázka ohledně celkové velikosti repozitáře. Git proto rozlišuje dva způsoby uložení objektů:

volné objekty (loose objects) pro nově vzniklé objekty a zabalené objekty (packed objects) pro dlouhodobé úložiště [4].

Při běžné práci je každý objekt zkomprimován pomocí knihovny zlib a uložen jako samostatný volný objekt. Liší se pouze metadata o tom, o jaký typ objektu se jedná [4].

Když počet volných objektů překročí určitou mez, Git automaticky spustí údržbu. Během ní jsou všechny volné objekty zabaleny do jednoho velkého binárního souboru zvaného packfile. Tato operace se většinou spouští automaticky na pozadí při příkazu `git push` nebo ji lze vyvolat ručně příkazem `git gc` [4].

Při vytváření packfilu je využívána delta komprese. U tohoto způsobu komprese je nejnovější verze souboru uložena v plném rozsahu a všechny předchozí verze jsou uchovávány jako seznam změn vůči nejnovější verzi. Tím je zajištěn rychlý přístup k posledním změnám, které uživatel nejčastěji potřebuje, zatímco méně využívaná historie se dopočítává, a zároveň se výrazně zmenší celková velikost databáze [4].

2.3.3 Programovatelný přístup ke Gitu

Jeden z mnoha programovatelných přístupů ke Gitu je knihovna *libgit2*¹. Jedná se o reimplementaci Git knihovny napsanou v jazyce C bez jakýchkoli externích závislostí, se zaměřením na programátorsky přívětivé API. Knihovna poskytuje také tzv. language bindings pro různé jazyky, včetně Rustu, Pythonu, Ruby a dalších. Díky tomu je možné využívat Git pohodlně v ostatních aplikacích bez nutnosti znalosti jazyka C [4].

Jedna z nejzajímavějších vlastností této knihovny je podpora tzv. backendů (pluginů). Libgit2 umožňuje vývojářům napojit vlastní úložiště pro různé typy operací a vytvářet vlastní pravidla pro konfiguraci, ukládání referencí nebo i pro databázi objektů [4].

¹<https://libgit2.org/>

Kapitola 3

GitHub

Tato kapitola se věnuje platformě GitHub, která je jedním z nejpobulárnějších hostovaných řešení pro správu Git repozitářů. V této kapitole budou popsány klíčové funkce GitHubu a jeho API.

3.1 Úvod do GitHubu

GitHub je jedna z mnoha hostovaných služeb pro správu Git repozitářů. Rozšiřuje základní funkcionalitu Gitu o sociální funkce, webové rozhraní, CI/CD a další. Nejznámější alternativy jsou *GitLab*¹, který je na rozdíl od GitHubu open source a umožňuje vlastní provoz (self-hosting), a *Bitbucket*² [5].

Po vzniku Gitu pro jádro operačního systému Linux se brzy ukázalo, že vlastní hostování repozitářů není vždy dostačující a vznikla poptávka po sdílené cloudové službě. V roce 2008 proto vznikl GitHub a od té doby se stal největší platformou pro open source vývoj. V roce 2018 byl odkoupen společností Microsoft [5].

Hlavní rozdíl mezi Gitem a GitHubem spočívá v tom, že Git je nástroj pro správu verzí (VCS), zatímco GitHub je kompletní vývojová platforma. Kromě samotného Gitu nabízí možnosti CI/CD a nasazení, sledování chyb a požadavků (issue tracking), správu PR, diskuzní fóra komunity a další sociální funkce. PR je žádost o začlenění změn z jedné větve do druhé. Autor navrhne úpravy kódu a ostatní členové týmu je mohou zkontrolovat, okomentovat a schválit nebo zamítnout. GitHub také umožňuje hostování statických stránek prostřednictvím služby *GitHub Pages*³, která se využívá například pro hostování dokumentace projektů. Samozřejmostí je webové rozhraní pro veškerou správu včetně úpravy kódu a kontroly PR.

3.2 Základní funkce a model spolupráce

V této sekci jsou popsány nejpoužívanější funkce GitHubu a modely interakce mezi vývojáři [6].

¹<https://gitlab.com>

²<https://bitbucket.org>

³<https://pages.github.com>

3.2.1 Repozitáře

Repozitář představuje základní prvek na GitHubu. Jedná se o místo, kde je ukládán zdrojový kód a jiné soubory projektu. Repozitáře mohou být veřejné nebo soukromé a mohou mít více spolupracovníků [6]. Každý repozitář je jednoznačně identifikován kombinací vlastníka (uživatelský účet nebo organizace) a názvu repozitáře, například `rust-lang/rust`.

Forky

Fork je repozitář vytvořený pod jiným uživatelským účtem, který vychází z původního repozitáře a sdílí jeho historii. Forky jsou využívány primárně pro přispívání do cizích projektů. Nejčastěji je vytvořen fork, v něm jsou provedeny změny a následně je otevřen PR do původního repozitáře. Možnost navrhnout sloučení změn do původního projektu je hlavním rozdílem mezi forkem a klonem repozitáře. Fork se tedy částečně chová jako samostatná větev, avšak bez nutnosti mít přímý zápis do původního repozitáře. Tento způsob práce umožňuje bezpečně experimentovat bez přímého ovlivnění původního projektu [7].

Každý fork je plně samostatný a má vlastní větve, členy, issues, pull requesty, wiki a další funkce.

3.2.2 Issues

Issues (v překladu tikety, úkoly nebo problémy) představují jednu ze základních a nejvyužívanějších funkcí GitHubu. Jsou používány ke sledování chyb, nápadů, požadavků na funkce a dalších úkolů spojených s projektem [8].

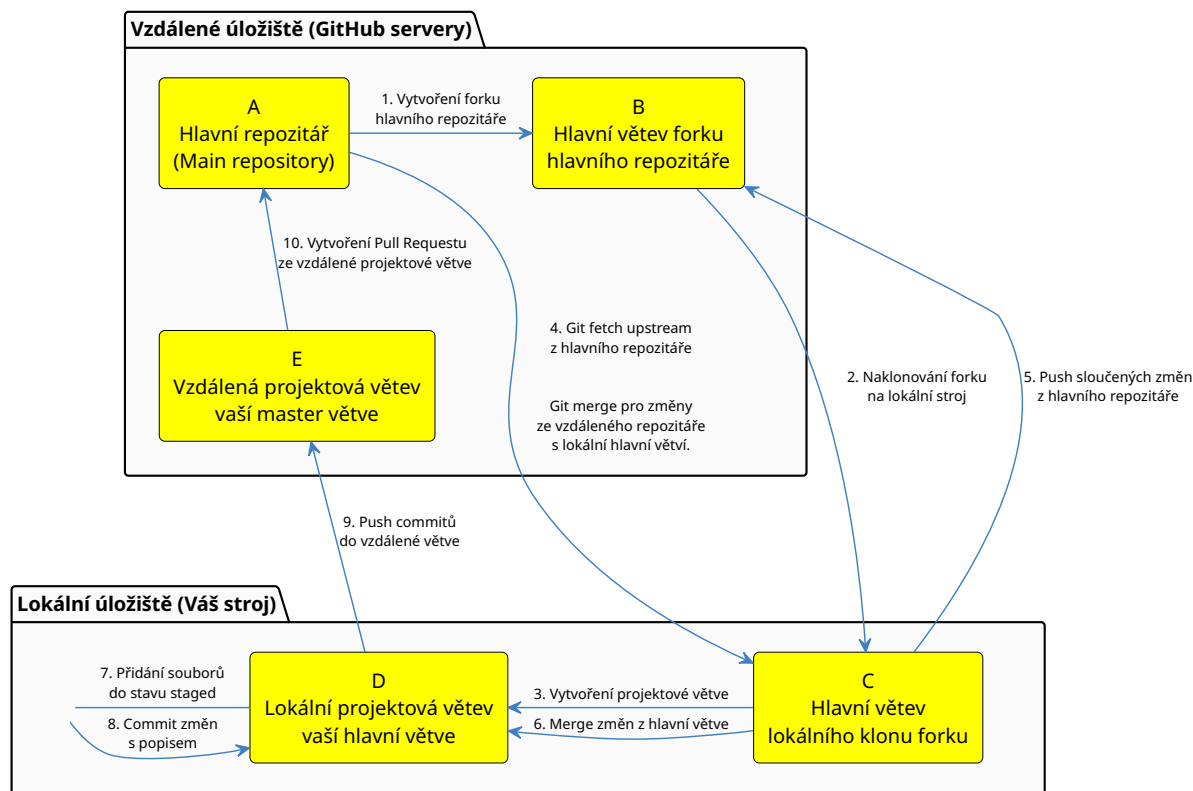
Jedna issue může mít sub-issues, aby mohl být větší úkol rozdělen na více menších částí. Pokud je issue závislá na dokončení jiné issue, lze tuto závislost explicitně vyjádřit a označit úkol jako blokový. Současně mohou být přidávána metadata, například labely, typ issue, přiřazení nebo milníky. Issue může být přiřazena konkrétnímu uživateli, což usnadňuje organizaci práce i sledování odpovědnosti.

3.2.3 Pull Requests

Pull requests (PR) představují mechanismus, kterým může být navrženo začlenění změn do projektu. Jde o speciální případ Issue takže všechny PR jsou i issues, ale ne všechny issues jsou PR. Umožňují vytvořit požadavek na provedení změn ve zdrojovém kódu. Před samotným sloučením obvykle proběhne proces code review, při němž ostatní vývojáři procházejí navržené změny, komentují konkrétní řádky kódu a schvalují nebo odmítají začlenění. Tím je podporováno udržení kvality kódu a včasné odhalení chyb [7].

Za zmínku stojí také dobrá provázanost mezi issues a PR. Pokud PR řeší konkrétní issue, je možné tuto issue automaticky uzavřít pomocí klíčových slov, například `Closes #123`, kde `#123` představuje číslo issue.

Typický workflow zahrnuje vytvoření forku a založení nové větve, například pro konkrétní issue. Poté jsou provedeny změny, které jsou odeslány do forku, a následně je z dané větve otevřen PR směrem do původního repozitáře. Pokud změny jsou schváleny, může být PR sloučen [7, 5].



Obrázek 3.1: Typický průběh práce s forkem na GitHubu [5].

Typický postup vytváření PR je znázorněn na Obrázku 3.1. V prvním kroku je vytvořen fork původního repozitáře. Následně je fork naklonován na lokální stroj a je vytvořena nová větev pro práci na nové funkcionalitě nebo pro opravu chyby (2 a 3). V průběhu práce je pravidelně prováděn příkaz `git fetch upstream` pro synchronizaci s původním repozitářem (4), změny jsou odesílány do forku (5) a vývojová větev je průběžně aktualizována (6). Při úpravě kódu jsou změněné soubory pravidelně přidávány do připraveného stavu (7) a následně commitovány (8). Po dokončení práce jsou změny odeslány do příslušné větve ve forku (9). Nakonec je vytvořen PR z forku do původního repozitáře (10) [5].

3.2.4 Štítky

Štítky (labels) představují klíčový nástroj pro kategorizaci a prioritizaci nejen issues, ale také PR a diskuzí. Štítky jsou vytvářeny v rámci repozitáře a následně přiřazovány jednotlivým issues, pull requestům a diskuzím. V každém novém repozitáři je k dispozici základní sada štítků, například `bug`,

`enhancement`, `question` nebo `help wanted`. Tyto štítky mohou být podle potřeby upravovány, přidávány i odstraňovány [8].

3.2.5 GitHub Actions

Platforma GitHub kromě hostování repozitářů poskytuje také řešení pro automatizaci procesů, jako je CI/CD. Jedná se o sadu praktik moderního vývoje softwaru. Continuous Integration (CI) označuje průběžné automatické sestavování a testování kódu po každé změně, Continuous Deployment (CD) pak automatické nasazování otestované verze do produkčního prostředí. Cílem je odhalovat chyby co nejdříve a zkrátit dobu mezi napsáním kódu a jeho nasazením [9].

Toto řešení se nazývá GitHub Actions a umožňuje přímo v prostředí GitHubu vytvářet a spouštět vlastní workflow.

Triggery

Triggery jsou události, které spouštějí workflow. Mezi nejběžnější patří `push`, `pull_request`, `schedule` pro plánované spouštění a `workflow_dispatch` pro manuální spuštění. Triggery umožňují flexibilně reagovat na různé události v repozitáři.

Workflow

Workflow je definováno v souboru s příponou `.yml` nebo `.yaml`, který je umístěn v adresáři `.github/workflows/`. Jedná se o automatizovaný proces složený z definic jednotlivých kroků. V repozitáři může existovat více workflow, které se spouštějí na základě různých podmínek, například pouze při `push` do hlavní větve.

Jobs a Steps

Workflow se skládá z jednotlivých úloh, které se označují jako jobs. Každý job je dále tvořen dílčími kroky, které se označují jako steps. Jobs standardně běží paralelně, lze je však nakonfigurovat tak, aby na sebe navazovaly. Každý krok může představovat buď příkaz spuštěný v shellu, nebo použití předdefinované akce (action). Steps oproti Jobs běží sekvenčně.

Runners

Runners zajišťují běh jednotlivých workflow a jejich kroků. Jedná se o virtuální servery, které mohou být hostovány buď GitHubem, nebo si je může uživatel provozovat sám.

Integrace s PR

Jednou z výhod GitHub Actions je integrace s PR. Při vytvoření nového PR mohou být automaticky spuštěny workflows a jejich výsledek je následně přehledně zobrazen přímo u daného PR. Takové

workflows často obsahují testy, kontrolu sestavení nebo ověření pravidel formátování kódu. Tím je zajištěna další vrstva kontroly kvality a je sníženo riziko, že se do hlavní větve dostane změna narušující funkčnost projektu [9, 7].

3.3 API GitHubu

GitHub poskytuje dva typy API pro přístup k datům i pro plnou interakci s platformou. API GitHubu umožňuje uživatelům automatizovat procesy a integrovat vlastní aplikace s platformou GitHub [10]. V této sekci jsou popsány základní vlastnosti a možnosti obou API, včetně omezení, která je nutné zohlednit při jejich využívání.

3.3.1 REST API

REST API umožňuje přístup k datům a funkcím GitHubu prostřednictvím HTTP požadavků. Každý požadavek na REST API obsahuje HTTP metodu a cestu. V závislosti na konkrétním endpointu musí být specifikovány další parametry, například hlavičky, informace o autentizaci, parametry dotazu nebo tělo požadavku [10]. Pro interakci s různými zdroji jsou využívány základní HTTP metody [10]:

- **GET** pro získání dat (např. seznam issues nebo detail PR),
- **POST** pro vytvoření nových zdrojů (např. vytvoření issue),
- **PUT** pro aktualizaci existujících zdrojů,
- **DELETE** pro odstranění zdrojů,
- **PATCH** pro částečnou aktualizaci zdrojů (např. změna stavu issue).

Každý endpoint je definován cestou. Cesta může obsahovat statické i dynamické části. Například endpoint pro získání issues repozitáře je `GET /repos/{owner}/{repo}/issues`, kde `owner` a `repo` představují dynamické části, které musí být nahrazeny konkrétními hodnotami [10]. Ukázka takového požadavku je uvedena ve Výpisu 3.1.

```
1 curl --request GET \  
2 --url "https://api.github.com/events?per_page=2&page=1" \  
3 --header "Accept: application/vnd.github+json" \  
4 --header "X-GitHub-API-Version: 2022-11-28" \  
5 https://api.github.com/events
```

Výpis 3.1: Ukázka REST API požadavku na GitHub pro získání událostí celého GitHubu.

HTTP/2 200 OK
Content-Type: application/json; charset=utf-8

```
1  [  
2    {  
3      "id": "11056651313",  
4      "type": "PushEvent",  
5      "actor": {  
6        "id": 41898282,  
7        "login": "github-actions[bot]",  
8        "display_login": "github-actions",  
9        "url": "https://api.github.com/users/github-actions[bot]"  
10     },  
11     "repo": {  
12       "id": 1184208499,  
13       "name": "IsBenben/xxxh-fans",  
14       "url": "https://api.github.com/repos/IsBenben/xxxh-fans"  
15     },  
16     "payload": {  
17       "repository_id": 1184208499,  
18       "push_id": 33348345827  
19     },  
20     "public": true,  
21     "created_at": "2026-04-26T09:11:54Z"  
22   }  
23 ]
```

Výpis 3.2: Ukázka odpovědi REST API na požadavek z Výpisu 3.1 (zkráceno).

Data jsou nejčastěji zprostředkována ve formátu JavaScript Object Notation (JSON), který zajišťuje dobrou čitelnost jak pro člověka, tak pro strojové zpracování. V závislosti na endpointu lze využít také jiné typy formátování obsahu, například `diff` nebo `patch` [10].

GitHub REST API je verzováno a využívá kalendářní verzování. Hlavička GitHub API verze `X-GitHub-API-Version` není povinná, ale je silně doporučována. Pokud tato hlavička není uvedena, je použita výchozí verze definovaná v dokumentaci, například `2022-11-28`. Verze API se do požadavku zapisuje ve formátu `X-GitHub-API-Version: 2022-11-28`. Tím je zajištěno, že novější verze nezmění chování integrace navázané na starší verzi. GitHub také poskytuje seznam zpětně nekompatibilních změn mezi verzemi, návody a dokumentaci pro migraci na novější verze API pro jednodušší přechod [11].

I když jsou některé endpointy veřejně dostupné bez autentizace, většina z nich vyžaduje autentizaci pomocí tokenu v hlavičce požadavku. Token lze získat více způsoby, například pomocí Personal Access Token (PAT), prostřednictvím GitHub App nebo jako vestavěný `GITHUB_TOKEN` v rámci GitHub Actions [9, 10].

3.3.2 GraphQL API

GitHub poskytuje také GraphQL API, které klientům umožňuje dotazovat se přesně na požadovaná data. Jedná se o odlišný přístup k získávání dat, který přináší vyšší flexibilitu a často také menší objem přenášených dat. Zatímco REST API vrací předem definované objekty z více endpointů, GraphQL využívá jeden endpoint, přes který je možné vyžádat přesně ta data, která jsou potřeba [12, 10].

Datový dotazovací jazyk GraphQL má následující vlastnosti:

Specifikace Na API serveru je definováno schéma, podle kterého jsou validovány klientské dotazy.

Silná typovost Ve schématu jsou přesně definovány typy a vazby mezi objekty.

Introspektivita Je možné získat podrobné informace o schématu.

Hierarchická struktura Tvar GraphQL dotazu odpovídá tvaru JSON dat, která jsou vrácena.

Aplikační vrstva GraphQL není pouze modelem uložení dat ani databázovým dotazovacím jazykem. Označení *Graph* odkazuje na grafové struktury definované ve schématu, kde uzly představují objekty a hrany vztahy mezi nimi.

Mezi časté problémy REST API patří nadměrné stahování dat (over-fetching) a nedostatečné stahování dat (under-fetching) [12, 10].

Over-fetching nastává, když klientská aplikace potřebuje pouze část vlastností objektu, ale v REST API je vrácen celý objekt s mnoha dalšími daty. To vede k neefektivnímu přenosu dat.

Under-fetching nastává, když klientská aplikace potřebuje více dat, než je dostupné v jedné odpovědi REST API, a z toho důvodu musí být provedeno více požadavků na různé endpointy.

GraphQL API tyto problémy omezuje tím, že umožňuje získat pouze ta data, která jsou skutečně potřebná. Tím může být dosaženo výrazného snížení přeneseného objemu dat. Současně je však třeba počítat s tím, že dotazy mohou být velmi komplexní a náročné na zpracování [12].

Dotazy jsou v GraphQL obvykle odesílány pomocí HTTP metody `POST` na jediný endpoint. Operace, které mají být provedeny, i požadovaná návratová data jsou definovány v těle požadavku [12]. Ukázka dotazu a odpovědi je uvedena ve Výpisech 3.3 a 3.4.

```
1 query {
2   repository(owner: "rust-lang", name: "rust") {
3     issues(first: 3, states: OPEN) {
4       nodes {
5         number
6         title
7         createdAt
8         labels(first: 5) {
9           nodes { name }
10        }
11      }
12    }
13  }
14 }
```

Výpis 3.3: Ukázka GraphQL dotazu na otevřené issues repozitáře.

```
1 {
2   "data": {
3     "repository": {
4       "issues": {
5         "nodes": [
6           {
7             "number": 138540,
8             "title": "Fix ICE in borrow checker",
9             "createdAt": "2026-04-20T10:32:00Z",
10            "labels": {
11              "nodes": [
12                { "name": "C-bug" },
13                { "name": "T-compiler" }
14              ]
15            }
16          }
17        ]
18      }
19    }
20  }
21 }
```

Výpis 3.4: Zkrácená odpověď na GraphQL dotaz z Výpisu 3.3.

Rozlišovány jsou dva hlavní typy operací: *Query* pro získávání dat a *Mutation* pro změnu dat. Operace typu *Query* jsou určeny výhradně pro čtení dat a umožňují získat komplexní strom informací v jednom dotazu. Operace typu *Mutation* jsou určeny pro vytváření, úpravu i mazání dat. Používány jsou například při vytvoření nové issue, změně stavu PR nebo odstranění komentáře [12].

Autentizace pro GraphQL API funguje obdobně jako pro REST API, tedy pomocí autentizačního tokenu v hlavičce požadavku [12, 10]. Na GitHubu je pro GraphQL endpoint vyžadována autentizace,

protože musí být řízena náročnost dotazů a spotřeba prostředků [12].

3.3.3 Specifika omezování a limity

GitHub omezuje počet dotazů, které mohou být provedeny v daném časovém období. Tyto limity pomáhají zajišťovat stabilitu služby pro všechny uživatele a současně slouží jako ochrana před zneužitím. Některé endpointy mohou mít navíc specifické limity z důvodu vyšší náročnosti nebo odlišné implementace [13].

Pro neautentizované uživatele je limit REST API nastaven na 60 požadavků za hodinu. Toto omezení je vyšší při autentizaci. Autentizovaní uživatelé nebo aplikace mají limit 5 000 požadavků za hodinu pro daný účet. Pokud je aplikace vlastněna nebo schválena organizací GitHub Enterprise Cloud, může být limit zvýšen až na 15 000 požadavků za hodinu. Ovšem tyto dotazy se počítají do osobního limitu uživatele, který je vlastníkem nebo oprávněným uživatelem aplikace. Aktuální stav limitu je možné zjistit pomocí endpointu `GET /rate_limit` [13].

Stránkování (*pagination*) je technika rozdělení velkého množství výsledků do menších částí zvaných stránky. Klient postupně načítá jednotlivé stránky dat pomocí kurzorů nebo offsetů namísto jednoho velkého dotazu, čímž se snižuje zátěž serveru i přenášený objem dat [12].

Z důvodu odlišné architektury je GraphQL API limitováno bodovým systémem. Každý dotaz má přiřazenu hodnotu *cost* podle své složitosti. Náročnost dotazu je mimo jiné odvozena od počtu uzlů (*node limit*). U stránkovaných spojení musí být obvykle zadán limit v rozsahu 1 až 100 položek pomocí parametrů `first` nebo `last`. Celkový počet vrácených uzlů v jednom dotazu nesmí překročit 500 000. Omezen je také čas zpracování. Pokud dotaz není vyhodnocen v časovém limitu 10 sekund, je ukončen a je vrácena chyba [14].

GraphQL API také poskytuje možnost získat informaci o celkové ceně dotazu (*cost*) pomocí objektu `rateLimit { cost }` přímo v dotazu. Cena může být odhadnuta na základě počtu požadavků potřebných k naplnění jednotlivých unikátních spojení v dotazu. Výsledná hodnota je následně vydělena konstantou 100 a zaokrouhlena na celé číslo [14].

Příklad GraphQL dotazu a výpočet složitosti Ve Výpisu 3.5 je zobrazen dotaz, ve kterém je požadováno 50 repozitářů a pro každý z nich 10 nejnovějších issues.

```

1 query {
2   viewer {
3     repositories(first: 50 ) {
4       edges {
5         repository:node {
6           name
7
8           issues(first: 10 ) {
9             totalCount
10            edges {
11              node {
12                title
13                bodyHTML
14              }
15            }
16          }
17        }
18      }
19    }
20  }
21 }

```

Výpis 3.5: Ukázka korektního GraphQL dotazu.

```

1 50 = 50 repozitářů
2 +
3 50 x 10 = 500 issues v repozitářích
4 -----
5 = 550 celkových uzlů (nodes)

```

Výpis 3.6: Výpočet celkového limitu dotazu (Node limit)

Odhad celkové ceny při plném naplnění všech limitů lze vyjádřit jako 1 bod za získání repozitářů a 500 bodů za získání issues, tedy $(1 + 50 \times 10)/100 = 5,01$. Po zaokrouhlení je výsledná cena 5 bodů. Detailní výpočet počtu uzlů pro tento dotaz je uveden ve Výpisu 3.6 [14].

Ve výpisu 3.7 je zobrazen další komplexní dotaz, ve kterém je požadováno prvních 50 repozitářů, pro každý z nich prvních 20 PR a pro každý PR prvních 10 komentářů. Současně je pro každý repozitář požadováno prvních 20 issues a pro každou issue prvních 10 komentářů. Takový dotaz má 22 050 uzlů, jejichž výpočet je uveden ve Výpisu 3.8, a odhadovanou cenu $(1 + 20 \times 50 \times 10 + 20 \times 50 \times 10)/100 = 200,01$, tedy po zaokrouhlení 200 bodů [14].

```
1 query {
2   viewer {
3     repositories(first: 50 ) {
4       edges {
5         repository:node {
6           name
7
8           pullRequests(first: 20 ) {
9             edges {
10              pullRequest:node {
11                title
12
13                comments(first: 10 ) {
14                  edges {
15                    comment:node {
16                      bodyHTML
17                    }
18                  }
19                }
20              }
21            }
22          }
23
24          issues(first: 20 ) {
25            totalCount
26            edges {
27              issue:node {
28                title
29                bodyHTML
30
31                comments(first: 10 ) {
32                  edges {
33                    comment:node {
34                      bodyHTML
35                    }
36                  }
37                }
38              }
39            }
40          }
41        }
42      }
43    }
44  }
45 }
```

Výpis 3.7: Ukázka komplexního GraphQL dotazu

1	50		= 50 repositories
2	+		
3	50	x 20	= 1,000 pullRequests
4	+		
5	50	x 20 x 10	= 10,000 pullRequest comments
6	+		
7	50	x 20	= 1,000 issues
8	+		
9	50	x 20 x 10	= 10,000 issue comments
10			
11			= 22,050 total nodes

Výpis 3.8: Výpočet celkového limitu komplexního dotazu (Node limit)

3.3.4 Webhooks API

Webhooky představují způsob získávání dat, při kterém je komunikace inicializována serverem, nikoliv klientem. Webhooky umožňují přihlášení k odběru událostí, které v softwarovém systému nastávají, a automatické přijímání dat o těchto událostech téměř v reálném čase [15].

Webhooky jsou alternativou k polling přístupu, při kterém klient pravidelně odesílá dotazy na server z důvodu zjištění nových událostí. Při použití webhooků klient jednorázově vyjádří zájem o vybrané typy událostí a server následně notifikace odesílá automaticky, jakmile k události dojde [15].

Alternativa k přístupu Polling

V následujících odstavcích jsou porovnány oba přístupy z hlediska latence, spotřeby API limitů a spolehlivosti doručení.

Metoda pravidelného dotazování (Polling) Tato metoda spočívá v tom, že monitorovací služba pravidelně odesílá dotazy na API, zda došlo ke změně dat. Například může být každou minutu odeslán dotaz, zda v repozitáři nevznikla nová issue. Tento přístup má dvě hlavní nevýhody. První nevýhodou je, že data nejsou získávána v reálném čase, ale se zpožděním úměrným délkou intervalu mezi dotazy. Druhou nevýhodou je neefektivní využití limitů API, protože při velkém množství sledovaných entit může být značná část limitu spotřebována na dotazy, které vracejí nezměněnou nebo prázdnou odpověď [15].

Výhody a nevýhody Webhooků Webhooky obvykle vyžadují méně zdrojů než polling přístup. Současně je dosaženo lepší škálovatelnosti při sledování většího množství zdrojů, protože není nutné periodicky odesílat samostatný dotaz pro každý sledovaný objekt. Další výhodou je získání informace o změně téměř v reálném čase, protože notifikace je odeslána bezprostředně po vzniku události [15].

Webhooky mají i své nevýhody. Jednou z nejvýznamnějších je požadavek na trvalou veřejnou dostupnost přijímací služby. Pokud služba dostupná není, může dojít k nedoručení části událostí, i když GitHub podporuje opakované doručení [15]. Z tohoto důvodu bývá v praxi často použit hybridní přístup, kdy webhooky slouží jako primární mechanismus získávání událostí a polling je využit jako záložní synchronizační mechanismus [15, 10].

Kapitola 4

Analýza existujících nástrojů

V této kapitole jsou analyzovány existující nástroje pro monitorování a sledování statistik vývoje softwaru. Cílem je identifikovat jejich silné a slabé stránky a na základě toho vymezit prostor pro vlastní řešení.

Mnoho velkých open source projektů, jako je například překladač jazyka Rust, si nevystačí s obecnými nástroji pro monitorování, které nabízí GitHub. Proto si tvoří vlastní nástroje přizpůsobené jejich specifickým potřebám a workflow.

4.1 Nativní statistiky GitHubu

GitHub nabízí základní nativní statistiky dostupné přímo v záložce *Insights* na stránce každého repozitáře. Zahrnují přehled otevřených a zavřených PR, aktivitu přispěvatelů a další údaje.

Obsah a funkce

Hlavním místem pro získání obecných statistik a grafů je záložka Insights. Nejdůležitější části zahrnují pohledy Pulse, Contributors a Commits. U všech statistik lze vybrat časové období, nejčastěji den, týden, měsíc nebo rok. U některých statistik je z důvodu velkého množství dat toto období omezeno, například na maximálně poslední měsíc.

Pulse Poskytuje rychlý přehled o aktivních a uzavřených PR, otevřených a uzavřených issues, nejaktivnějších autorech a základní sumarizaci aktivit včetně přidáných a odebraných řádků kódu. Obsahuje i seznam uzavřených PR za zvolené časové období. Statistika jsou k dispozici za poslední den, 3 dny, týden a měsíc.

Contributors Záložka poskytuje informace o nejaktivnějších přispěvatelích, včetně automatizovaných účtů (botů). Zobrazuje počet commitů jak celkový, tak za každého přispěvatele zvlášť. Počet commitů je dynamicky agregován podle zvoleného časového období (minimálně 1 týden).

Časový rozsah lze vybrat od posledních dvou týdnů po celou dobu existence repozitáře. Pokud repozitář obsahuje méně než 10 000 commitů, zobrazí tento pohled i počet změněných řádků pro každého přispěvatele.

Commits Záložka obsahuje graf celkového počtu commitů za poslední rok, přičemž počet commitů je agregován po týdnech.

Přehledný způsob zobrazení většího množství PR nebo issues v jednom grafu zde chybí. V záložce Issues je však možné filtrovat pomocí pokročilých filtrů, které umožňují kombinovat podmínky pomocí booleovských operátorů a závorek. Výsledkem je seznam issues nebo PR s celkovým počtem otevřených a zavřených položek.

Dalším nástrojem jsou GitHub Projects, kde jsou issues k jednotlivým funkcionalitám rozděleny do stavů definovaných vlastníkem repozitáře (například TODO nebo IN PROGRESS). Tento nástroj však neposkytuje žádné statistiky a slouží spíše k organizaci práce než k získávání informací o projektu.

Zhodnocení

Nativní statistiky GitHubu poskytují pouze základní přehled o celkovém projektu. Jejich hlavní výhodou je nativní dostupnost bez nutnosti cokoli nasazovat nebo udržovat.

Statistiky ohledně PR nebo issues jsou zde obtížněji dostupné a pro jejich získání je nutné sledovaný repozitář dobře znát. Na druhou stranu, k nalezeným PR nebo issues lze přejít přímo z výsledků filtrování.

4.2 Curl Dashboard

Curl je multiplatformní nástroj příkazové řádky a taktéž knihovna (*libcurl*) pro přenos dat pomocí různých síťových protokolů [16]. Curl Dashboard¹ je statický nástroj pro zobrazení různých statistik o projektu curl za celou dobu jeho existence. Jeho primárním účelem není operativní řízení vývoje, nýbrž slouží jako zdroj informací z dlouhodobého hlediska pro sledování zdraví projektu a jeho trendů.

Obsah a funkcionalita

Jedná se o statickou stránku zobrazující denně generované grafy a statistiky o projektu curl. Tyto grafy jsou ve formátu obrázků a jsou generovány pomocí skriptů analyzujících data z git repozitáře a API GitHubu projektu curl.

Na rozdíl od GitHub Insights (viz Sekce 4.1) zde není interaktivní přístup k datům ani možnost výběru zobrazovaného časového rozsahu. Kromě dat dostupných přímo na GitHubu jsou zde zobrazeny i další informace, jako například statistiky bug bounty programu, počty chyb a bezpečnostních

¹<https://curl.se/dashboard.html>

zranitelností, distribuce commitů podle časových pásem nebo poměr počtu úkolů (TODO) k počtu známých chyb.

Zhodnocení

Curl Dashboard poskytuje velmi komplexní a detailní pohled na dlouhodobý stav a zdraví projektu curl. Hlavní výhodou je velké množství grafů a statistik pokrývajících širokou škálu aspektů projektu. Samotné grafy jsou jednoduché, přehledné a srozumitelné.

Nevýhodou tohoto nástroje je absence interaktivity. Velké množství grafů na jedné stránce může ztěžovat orientaci a nalezení požadovaných informací. Protože se jedná o čistě statickou stránku, není možné získat detailní informace o konkrétním dni nebo měsíci a při výskytu výrazné odchylky v grafu nelze zjistit, co přesně se stalo.

4.3 Rustc PR Tracking

Rustc PR Tracking² je statická stránka zobrazující informace o PR z repozitářů organizace rust-lang. Nástroj závisí na konkrétních štítcích a procesním workflow používaném v rámci této organizace.

Obsah a funkcionalita

Nástroj je primárně zaměřen na sledování stavu PR a aktivity v nich. Sleduje repozitáře organizace rust-lang, konkrétně překladač Rustu, crates.io, rust-analyzer a clippy. Crates.io je registr balíčků pro ekosystém jazyka Rust.

Zobrazovaná data lze filtrovat podle časového rozsahu, nejméně za poslední den, nejvýše za posledních 999 dní. Stránka se aktualizuje denně.

Zhodnocení

Jedná se o jednoduchý statický web s denní aktualizací dat. Nástroj je snadno pochopitelný a přehledný. Poskytuje detailní vhled do stavu PR, zejména co se týče čekání na code review nebo na sloučení.

Nevýhodou je úzké zaměření výhradně na PR, issues ani celkové metriky repozitáře nejsou sledovány.

4.4 Shrnutí analýzy

GitHub Insights poskytuje pouze základní statistiky ohledně PR a issues. Pro detailní zobrazení je nutné znát sledovaný repozitář a aktivně využívat pokročilé filtry. Výhodou oproti ostatním

²<https://rust-lang.github.io/rustc-pr-tracking/>

nástrojům je přímý přístup k jednotlivým PR a issues přímo z výsledků filtrování. Curl Dashboard nabízí komplexní vhled do celkové historie projektu, včetně dat, která přímo nesouvisí s kódem (distribuce commitů podle časových pásem, počty bezpečnostních zranitelností), a poskytuje tak širší pohled na projekt. Chybí zde však detailní informace o konkrétních PR nebo issues. Rustc PR Tracking nabízí přehledný pohled na stav PR ve frontě ke schválení, ale zcela opomíjí issues a celkové metriky projektu.

Žádný z analyzovaných nástrojů přímo nepomáhá s operativním vývojem, jedná se spíše o nástroje pro získávání informací než pro řešení konkrétních problémů. Rovněž chybí řešení, které by propojovalo detailní pohled na PR a issues s historickými daty a aktivitou nad soubory v repozitáři. Tato mezera je hlavní motivací pro vytvoření vlastního nástroje.

Kapitola 5

Repozitář Rust překladače

Rust je systémový programovací jazyk zaměřený na bezpečnost paměti a výkon, jehož vývoj zastřešuje nezisková organizace Rust Foundation. Vývoj probíhá otevřeně na GitHubu v rámci organizace rust-lang, která sdružuje stovky přispěvatelů a spravuje desítky repozitářů. Samotný překladač je vyvíjen v repozitáři rust-lang/rust, kam denně přispívají vývojáři z celého světa s různými rolemi a oprávněními. Tato komplexita si vyžaduje automatizaci procesů, jako jsou přiřazování reviewerů, správa štítků nebo řízení merge fronty [17, 18].

Tato kapitola se zaměřuje na specifika a charakteristiku repozitáře Rust překladače. Z důvodu jeho complexity a interního workflow je tento repozitář ideálním kandidátem pro analýzu a sledování vývoje. Denně jsou zde otevírány a uzavírány desítky PR a issues s různými návrhy a úpravami. Jsou zde rovněž popsána specifika relevantní pro co nejefektivnější vzhled do stavu a průběhu vývoje projektu.

5.1 Rust Team API

Organizace rust-lang je rozdělena do specializovaných týmů, z nichž každý odpovídá za konkrétní oblast projektu, například za vývoj překladače, návrh jazyka nebo správu infrastruktury. Členové týmů mají různá oprávnění, jako je schvalování PR nebo přidělování štítků, přičemž tato oprávnění jsou svázána právě s příslušností k danému týmu.

Organizace rust-lang spravuje přístupová práva uživatelů prostřednictvím speciálního repozitáře `rust-lang/team`¹, který obsahuje seznam týmů, jejich členů a přidělených rolí. Tento repozitář funguje jako jediný zdroj pravdy (*single source of truth*) pro informace o tom, kdo má jaká oprávnění a jaké týmy v organizaci existují.

Struktura týmů a jejich přiřazení uživatelů je definována pomocí souborů ve formátu TOML, které jsou spravovány prostřednictvím PR. Na základě tohoto repozitáře je generováno veřejné Team API využívané automatizačními boty v repozitářích organizace.

¹<https://github.com/rust-lang/team>

Toto řešení umožňuje provoz automatizačních botů, jako je bors nebo triage, v repozitáři překladače Rustu. Tito boti zajišťují přidělování kontroly PR a issues podle interních algoritmů a automaticky přiřazují štítky označující aktuální stav daného PR nebo issue.

5.1.1 Příspěvatelé a členové týmů

V kontextu repozitáře rust-lang/rust je nutné rozlišovat dva pojmy.

Příspěvatel (contributor) je jakýkoliv GitHub uživatel, který vytvořil PR nebo issue v repozitáři. Příspěvatel nemusí být přiřazen k žádnému týmu. Příspěvatelé jsou identifikováni svým GitHub uživatelským jménem a jednoznačným číselným identifikátorem.

Člen týmu je příspěvatel, který je zároveň evidován v Team API jako člen nějakého týmu organizace rust-lang. Členové týmů mají zpravidla udělena dodatečná oprávnění odpovídající oblasti, za níž jejich tým odpovídá. Například právo schvalovat PR nebo přiřazovat štítky. Jeden uživatel může být členem více týmů současně.

5.1.2 Bezpečnost

Bezpečnost Team API zajišťuje několik opatření. Prvním z nich je zabránění neoprávněnému přístupu prostřednictvím spuštění škodlivého kódu v CI. Úspěšný útok by mohl vést k získání plných práv k celé infrastruktuře organizace. Ochrana spočívá v tom, že nikdo kromě hlavních administrátorů, ani běžní správci (maintaineri), nemá přístup k tajným tokenům, nemůže upravovat soubory pipeline, provádět sloučení neschváleného kódu ani si zvyšovat vlastní oprávnění.

Druhá skupina bezpečnostních opatření chrání před hrozbami manipulace s obsahem repozitáře. Takový útok by mohl provést maintainer a mohl by zahrnovat například změnu Uniform Resource Locator (URL) domovské stránky za falešnou, eskalaci vlastních práv nebo práv napříč více týmy, případně smazání ostatních uživatelů. Ochranou proti těmto útokům je povinnost schválení změn ostatními maintainery a zavedení kontrol v CI, které ověřují, zda změny v souborech TOML nevedly k podezřelé aktivitě [19].

5.2 Štítky

Výchozí stavy, které poskytuje GitHub (otevřeno, zavřeno, sloučeno), jsou pro mnoho malých projektů dostačující, avšak v repozitáři překladače Rustu jsou nedostatečné. Proto jsou zde automaticky přiřazovány různé druhy štítků, jako například `S-waiting-for-author` nebo `S-waiting-for-review`, díky nimž je zřetelně patrné, v jakém stavu se daný PR nebo issue právě nachází.

Štítky lze primárně rozdělit do následujících kategorií:

- `S-*` – stavy, ve kterých se daný PR nebo issue právě nachází. Příkladem jsou štítky `S-waiting-for-review` (PR čeká na code review) nebo `S-waiting-for-author` (PR čeká na úpravy od autora). Tyto štítky jsou zásadní pro sledování stavu v průběhu vývoje.

- **T-*** – štítky označující tým, jehož se daný PR nebo issue týká. Například **T-lang** pro jazykový tým nebo **T-compiler** pro tým pracující na překladači.
- **A-*** – štítky pro oblasti, kterých se daný PR nebo issue dotýká. Například **A-macros** pro makra nebo **A-diagnostics** pro zprávy o chybách a upozorněních.
- **E-*** – výzvy ke spolupráci (*calls for participation*). Například **E-easy** označuje issue vhodnou pro nové přispěvatele a **E-needs-mcve** žádá o minimalizaci reprodukce chyby [20].

Typů štítků je v repozitáři výrazně více, avšak výše uvedené kategorie jsou základní pro pochopení fungování systému štítků a jeho možností pro sledování stavu a kategorizaci jednotlivých PR a issues [20].

5.3 Proces sloučení pull requestů

Sloučení PR v repozitáři rust-lang/rust neprobíhá standardním způsobem pomocí merge na GitHubu. Místo toho je tento proces řízen specializovaným botem bors, který zajišťuje, že každý PR je před sloučením otestován v aktuálním stavu větve **main**. Tím je zabráněno situaci, kdy PR sice prochází testy v okamžiku code review, ale po sloučení s mezitím aktualizovanou větví **main** testy selžou [21].

5.3.1 Bot bors

Bors je bot pro správu fronty sloučení (*merge queue*). Jeho instance pro repozitář rust-lang/rust je dostupná na adrese <https://bors.rust-lang.org/queue/rust> a vystupuje pod účtem **@bors** na GitHubu. Bot je naprogramován v jazyce Rust a jeho zdrojový kód je dostupný v repozitáři rust-lang/bors [21, 22].

5.3.2 Průběh sloučení pull requestu

Průběh sloučení PR v repozitáři rust-lang/rust lze shrnout do následujících kroků [23]:

1. **Otevření PR** – Přispěvatel otevře PR s navrhovanými změnami. Bot **@rustbot** automaticky přiřadí reviewera na základě změněných souborů a přidělí odpovídající štítky.
2. **Code review** – Přiřazený reviewer přezkoumá změny a případně si vyžádá úpravy (štítek **S-waiting-on-author**). Jakmile je PR připraven ke sloučení, reviewer jej schválí příkazem **@bors r+** v komentáři.
3. **Fronta sloučení** – Příkaz **r+** zařadí PR do fronty sloučení bota bors.
4. **Spuštění CI** – Bors vytvoří pomocnou větev **automation/bors/auto**, do které sloučí změny z PR s aktuálním stavem větve **main**, a spustí kompletní sadu testů CI.

5. **Sloučení** – Pokud testy projdou, bors sloučí PR do větve `main` a uzavře jej. Pokud testy selžou, bors PR zamítne a informuje autora o výsledku.

Kapitola 6

Návrh systému

Tato kapitola se zaměřuje na návrh a implementaci nástroje pro monitorování velkých repozitářů, konkrétně se zaměříme na repozitář Rust překladače.

6.1 Požadavky na systém

Před návrhem systému je nutné vymezit, *co* má systém poskytovat (funkční požadavky) a *jaké vlastnosti* má při provozu splňovat (nefunkční požadavky).

6.1.1 Funkční požadavky

1. Webová aplikace s jednoduchým přístupem

- Aplikace je dostupná v prohlížeči, bez nutnosti instalace klientského software.
- Aplikace je read-only a nevyžaduje autentizaci uživatelů, aby ji mohl používat kdokoli ze zájemců o vývoj sledovaného repozitáře.

2. Vždy aktuální data o sledovaném repozitáři

- Aplikace průběžně získává nová data o PR, issues a aktivitě nad soubory.
- Aplikace umožňuje opakované spouštění synchronizace bez ztráty konzistence dat a doplňuje nové i změněné záznamy.

3. Historizace stavů v čase

- Aplikace uchovává časový vývoj stavů a atributů PR a issues, aby bylo možné rekonstruovat jejich historický stav.
- Aplikace dokáže historii doplnit (*backfill*) i zpětně pro záznamy vzniklé v době, kdy nebyla v provozu.

4. Analytické pohledy a filtrování

- Aplikace umožňuje zobrazovat statistiky a dotazy v různých časových oknech.
- Aplikace umožňuje filtrovat výsledky podle relevantních parametrů (např. týmu, cesty v repozitáři, identifikátoru PR či issue).
- Pro práci s většími datovými sadami je k dispozici stránkování výsledků.
- Konkrétní sada poskytovaných dotazů je vymezena v Podsekci 6.1.2.

5. Self-healing po výpadku

- Pokud aplikace nějakou dobu neběží nebo dojde k chybě při synchronizaci, je schopna chybějící data doplnit při dalším spuštění.
- Aplikace minimalizuje duplicitní záznamy a udržuje konzistentní stav i při opakovaném zpracování stejných vstupů.

6. Jednoduchý deployment

- Aplikaci je možné spustit v lokálním i serverovém prostředí bez náročné ruční konfigurace.
- Citlivé údaje, jako například přístupové tokeny, jsou předávány externí konfigurací, nikoliv pevně zapsány v kódu.

6.1.2 Specifikace analytických dotazů

Funkční požadavek na analytické pohledy vyžaduje sadu konkrétních dotazů nad nasbíranými daty. Tato podsekcce pro každý dotaz uvádí jeho účel, vstupní parametry a očekávaný výstup. Návrh realizace jednotlivých dotazů je popsán v Sekci 6.6.

Q1 – Historický stav PR k danému datu Při revizi kódu ve velkém repozitáři je užitečné zpětně zjistit, v jakém stavu se konkrétní PR nacházel v určitém období. Tento dotaz umožňuje rekonstruovat historii stavů PR na základě zaznamenaných událostí a změn štítků.

Vstup: identifikátor PR, počáteční datum, počet dní zpětně.

Výstup: seznam stavů PR v zadaném časovém rozmezí, každý s časovým razítkem.

Q2 – Počet PR v konkrétním stavu k datu Pro sledování trendů v procesu revize kódu je potřeba zjistit, kolik PR se nacházelo v daném stavu k určitému dni. Opakovaným voláním tohoto dotazu pro různá data lze sestavit časovou řadu zachycující vývoj počtu PR v konkrétním stavu.

Vstup: stav PR (výčtový typ), datum.

Výstup: počet PR v zadaném stavu k danému datu.

Q3 – PR čekající na zpracování V rozsáhlém repozitáři snadno vzniká situace, kdy některé PR dlouho čekají na pozornost. Tento dotaz zobrazuje aktuální PR ve stavech `S-waiting-on-review`, `S-waiting-on-bors` nebo `S-waiting-on-author` (viz Sekce 5.2), čímž umožňuje identifikovat nejstarší čekající PR.

Vstup: (žádné specifické parametry, pracuje s aktuálním stavem).

Výstup: stránkovaný seznam PR seřazený podle doby čekání.

Q4 – Události konkrétní issue k datu Pro pochopení vývoje konkrétní issue je potřeba vidět chronologický přehled jejích událostí. Dotaz vrací zaznamenané akce jako uzavření, znovuotevření nebo komentáře k zadanému datu.

Vstup: identifikátor issue, datum.

Výstup: seznam událostí dané issue k zadanému datu s časovými razítky. Pokud issue neexistuje, vrací se prázdná odpověď.

Q5 – Nejčastěji upravované soubory přispěvatelem Při rozdělování práce v týmu je užitečné vědět, na kterých částech projektu se daný vývojář nejvíce podílí. Dotaz vrací soubory, které přispěvatel nejčastěji upravoval v zadaném období.

Vstup: jméno přispěvatele, počet dní zpětně od aktuálního data.

Výstup: seřazený seznam souborů s počtem úprav a metadaty přispěvatele.

Q6 – Přispěvatelé, kteří upravili soubor nebo složku Tento dotaz nabízí opačný pohled k Q5. K zadané cestě souboru nebo adresáře vrací přispěvatele, kteří v dané oblasti prováděli změny. Cesta je vyhledávána jako prefix, takže dotaz pokrývá i celý adresář včetně podadresářů.

Vstup: cesta k souboru nebo složce, časové okno.

Výstup: stránkovaný seznam přispěvatelů s metadaty a celkovým počtem.

Q7 – Soubory upravené členy týmu Pro koordinaci práce mezi týmy je potřeba zjistit, na kterých souborech nebo modulech pracují členové konkrétního týmu. Výsledek lze zobrazit jako plochý seznam, stromovou strukturu seskupenou do zvolené hloubky adresářové hierarchie, nebo jako úplný adresářový strom.

Vstup: název týmu, časové okno, počet úrovní hierarchie pro seskupení.

Výstup: seznam souborů (plochý nebo hierarchický) seřazený podle počtu úprav.

6.1.3 Nefunkční požadavky

1. Výkon

- Aplikace musí efektivně zpracovávat data z rozsáhlého repozitáře s velkým množstvím PR a issues.

- Odezva analytických dotazů musí zůstat únosná i nad historickými daty v řádu desítek tisíc PR.

2. Spolehlivost a odolnost

- Aplikace musí korektně zpracovat i neúplná data z externích zdrojů.
- Při dílčích chybách musí být chyba zaznamenána a musí být možné synchronizaci zopakovat bez poškození již uložených dat.

3. Konzistence a kvalita dat

- Aplikace musí minimalizovat duplicitní záznamy a udržovat konzistentní stav.
- Aplikace musí uchovávat časové souvislosti událostí pro pozdější analýzu trendů.

4. Provozovatelnost

- Aplikace musí být konfigurovatelná pomocí externí konfigurace (sledovaný repozitář, přístupové tokeny, připojení k databázi).
- Aplikace musí poskytovat logování pro diagnostiku běhu a chyb.

5. Bezpečnost

- Přístupové tokeny k externím službám nesmí být pevně zapsány v kódu.
- Vstupní parametry uživatelských dotazů musí být validovány a chybové stavy bezpečně zpracovány.

6.2 Hlavní návrhová rozhodnutí

Po vymezení požadavků je nutné rozhodnout, *jakým způsobem* budou jednotlivé vlastnosti aplikace dosaženy. Tato sekce postupně rozebírá hlavní rozhodnutí, která ovlivnila podobu výsledné architektury. Formu nasazení aplikace, zdroje dat o repozitáři, způsob získávání aktuálních dat společně se schopností doplnit chybějící historii a volbu nástroje pro vizualizaci.

Další újeji zaměřené rozhodnutí, volba konkrétního databázového systému, je probíráno v Sekci 6.4.

6.2.1 Forma nasazení aplikace

Z požadavku na webovou aplikaci s jednoduchým deploymentem vyplývá otázka, jak má být aplikace strukturována jako celek. Zvažovány byly tři varianty.

Statická jednostránková aplikace s předgenerovaným datasetem Tímto způsobem je realizován například nástroj rustc-pr-tracking (viz Sekce 4.3). Veškerá data jsou připravena dopředu a poskytována jako statické soubory, které prohlížeč zpracovává lokálně. Výhodou je triviální

nasazení (libovolný statický hosting, žádný server) a nulová provozní režie. Nevýhodou je omezená velikost datasetu, neboť veškerá data musí být stažena do prohlížeče, a nemožnost provádět nad daty dynamické dotazy s uživatelskými parametry.

Backend s databází a oddělený frontend Klient-serverová architektura, kde backend zpřístupňuje data uložená v databázi přes API a frontend slouží pouze pro jejich zobrazení. Toto řešení umožňuje uchovávat data ve vysokém objemu a efektivně se nad nimi dotazovat pomocí Structured Query Language (SQL). Cenou je složitější nasazení (běžící server, databáze, případná kontejnerizace) a nutnost provozovat backend nepřetržitě.

Serverless funkce nad spravovanou databází Varianta, která odstraňuje potřebu vlastního serveru. Za to je ale aplikace vázána na konkrétního cloudového poskytovatele, je omezena na jím podporované databázové systémy a lokální vývoj je složitější.

Zvolené řešení. Aplikace je realizována jako backend nad relační databází s odděleným webovým frontendem. Hlavním důvodem je objem dat a potřeba dynamických analytických dotazů, které ve statické variantě nejsou realizovatelné. Závislost na konkrétním cloudovém ekosystému u serverless varianty by zároveň zhoršila přenositelnost a znesnadnila lokální vývoj. Jednoduchost lokálního nasazení je řešena kontejnerizací.

6.2.2 Zdroje dat o repozitáři

Analytické pohledy vyžadují data o PR, issues, jejich historii, změnách souborů a příslušnosti přispěvatelů k týmům. Otázkou je, odkud tato data získávat.

Pouze GitHub API GitHub by byl ideálním jediným zdrojem, ale pro tento projekt nestačí. Endpoint pro soubory změněné v rámci PR by bylo nutné volat pro každý z tisíců PR zvlášť, což by přineslo další výrazné navýšení počtu požadavků a zpomalilo synchronizaci kvůli síťové latenci [10]. Limity počtu požadavků (viz Podsekce 3.3.3) by tak byly rychle překročeny. Informace o příslušnosti přispěvatelů k oficiálním týmům Rust projektu GitHub API neposkytuje vůbec.

Pouze lokální git repozitář Klon repozitáře poskytuje přímý přístup ke struktuře commitů a jejich obsahu bez síťové režie. Sám o sobě však neobsahuje metadata o PR a issues (stavy, štítky, recenze, komentáře, časovou osu událostí), bez nichž nelze požadované analytické pohledy realizovat.

Kombinace více zdrojů Každý zdroj poskytne část dat: GitHub API poskytuje metadata o PR a issues včetně jejich časové osy, lokální git repozitář slouží k výpisu souborů změných v merge commitech a Rust Team API (viz Sekce 5.1) mapuje přispěvatele do týmů.

Zvolené řešení. Aplikace kombinuje všechny tři zdroje. Lokální git odbourává tisíce API požadavků, které by jinak mohly narážet na rate limit. Team API doplňuje informace o týmech a jejich přispěvatelích. GitHub API zůstává primárním zdrojem metadat o PR a issues a jejich časové ose.

6.2.3 Aktuálnost dat a self-healing

Z požadavků na vždy čerstvá data a na schopnost doplnit data po výpadku vyplývá otázka, jakým mechanismem aplikace získává nové i chybějící události z GitHubu.

Webhooky GitHub při každé relevantní události zašle HTTP požadavek na předem registrovaný endpoint [15]. Přístup je vhodný pro reaktivní zpracování v reálném čase, avšak má dvě podstatná omezení. Vyžaduje veřejně dostupný endpoint, což komplikuje lokální vývoj i nasazení v prostředích bez veřejné IP adresy. A z principu neposkytuje historická data události, ke kterým došlo před registrací webhooku nebo v době výpadku aplikace, jsou nenávratně ztraceny.

Pravidelný polling REST API stavu PR a issues Aplikace by v pravidelných intervalech načítala aktuální stav PR a issues a detekovala změny. Implementačně jednoduché, ale nezachytí mezistavy mezi dvěma odečty (například krátkodobě přidáný a odebraný štítek) a neposkytuje úplnou časovou osu událostí.

Timeline events API GitHub pro každý PR a issue zpřístupňuje endpoint vracející úplný chronologický seznam zaznamenaných událostí [10]. Tímto způsobem lze provést jak prvotní naplnění historických dat, tak inkrementální doplňování. Cenou je vysoký počet API požadavků při prvotní synchronizaci.

Zvolené řešení. Aplikace používá Timeline events API jako primární zdroj historie a aktuálních událostí. Synchronizace je inkrementální a opakovaně spustitelná, čímž je dosaženo požadované self-healing vlastnosti. Po libovolně dlouhém výpadku je při dalším spuštění historie automaticky doplněna. Webhooky byly zamítnuty kvůli závislosti na veřejném endpointu a neschopnosti dohnat data zpětně. Polling není dostatečný pro rekonstrukci kompletní časové osy.

6.2.4 Vizualizace dat

Ze specifikovaných analytických dotazů vyplývají dvě hlavní formy vizualizace: časové řady stavů PR (dotazy Q1 a Q2) vyžadující interaktivní spojnicové grafy a hierarchická data o upravených souborech (dotaz Q7) vyžadující prostorovou agregaci stromové struktury. Otázkou je, jaký nástroj pro vizualizaci zvolit.

Monitorovací platforma (Grafana) Grafana¹ je v oblasti monitoringu zavedeným standardem s přímou podporou PostgreSQL. Pro tento projekt se však ukázala jako nevhodná ze dvou

¹<https://grafana.com>

důvodů. Integrace analytických SQL dotazů obsahujících podmíněnou logiku a vícenásobné klauzule `WITH` naráží na limity při předávání dynamických parametrů. A vizualizace hierarchických textových dat (cesty k souborům seskupené do adresářové struktury) není nativními panely Grafany podporována. Grafana nabízí pouze plochý tabulkový výpis, který u stovek záznamů postrádá přehlednost.

Vlastní frontendový klient Jednostránková webová aplikace komunikující s backendem přes REST API. Poskytuje plnou kontrolu nad předáváním parametrů dotazů a umožňuje implementovat libovolné vizualizace včetně treemap pro stromová data a interaktivních grafů s možností skrývat jednotlivé řady. Cenou je nutnost implementace a údržby vlastního kódu.

Zvolené řešení. Vlastní frontendový klient ve formě jednostránkové webové aplikace. Hlavním důvodem je potřeba vizualizací, které hotové monitorovací nástroje nepodporují, zejména prostorová agregace souborových cest do treemap a plná flexibilita při parametrizaci dotazů. Konkrétní pohledy a jejich implementace jsou popsány v Sekci 7.4.

6.2.5 Volba rozhraní GitHub API

GitHub nabízí dvě hlavní rozhraní pro přístup k datům repozitáře: REST API (viz Podsekce 3.3.1) a GraphQL API (viz Podsekce 3.3.2). Pro stahování dat z GitHubu bylo v této práci zvoleno REST API.

Jedním z hlavních důvodů pro toto rozhodnutí byl způsob získávání událostí z časové osy (Timeline events), který je pro rekonstrukci úplné historie stavů PR a issues důležitý (viz Podsekce 6.2.3). Ačkoliv GraphQL umožňuje dotazování na tyto události, vyžaduje explicitní definici typů pro každou jednotlivou událost, což značně komplikuje flexibilní stahování kompletní historie. Dalším významným faktorem bylo využití knihovny *Octocrab*². Ta poskytuje robustního REST klienta pro GitHub s předpřipravenými datovými strukturami pro všechny endpointy. Díky tomu zcela odpadla nutnost manuální implementace serializace a deserializace JSON odpovědí, což prokazatelně zjednodušilo implementaci synchronizační vrstvy.

Zvolený přístup nicméně přináší nutný architektonický kompromis spojený s datovým modelem GitHubu. V něm je každý PR vnitřně považován zároveň i za Issue [10]. Zatímco REST API poskytuje specializovaný endpoint pro načítání samotných PR se všemi potřebnými detaily, endpoint určený pro issues vrací sdílený seznam obsahující obě entity.

Při synchronizaci samotných issues je proto nutné načítat data z tohoto endpointu a záznamy představující PR na úrovni aplikace dodatečně filtrovat a ignorovat. Ačkoliv by nasazení GraphQL API pro tuto konkrétní část synchronizace umožnilo dotazovat typy `Issue` a `PullRequest` zcela odděleně, a tím eliminovat stahování redundantních dat, bylo rozhodnuto zachovat jednotný přístup.

²<https://github.com/xampprocky/octocrab>

Výhody plynoucí z použití jedné technologie napříč celou aplikací a využití robustních datových modelů knihovny Octocrab tuto mírně sníženou efektivitu při stránkování vyvažují.

6.3 Architektura systému

Tato kapitola popisuje, jak byla navržena architektura systému. Jsou zde popsány jednotlivé moduly systému a způsob jejich vzájemné komunikace.

Celková architektura systému je znázorněna na Obrázku 6.1.

Systém je navržen jako monolitická aplikace s vrstvenou strukturou zahrnující vstupní vrstvu, aplikační a integrační logiku a prezentační vrstvu. Vrstvená struktura je dosažena za pomoci principu oddělení zodpovědností (SoC). Tento přístup umožňuje snadnou rozšiřitelnost a vysokou udržitelnost. Monolit je v rámci klient-serverové architektury rozdělen na backendovou část implementovanou v Rustu, která zajišťuje zpracování dat a vystavení API pro frontend, a frontendovou část implementovanou ve Vue.js, která zobrazuje data uživatelům.

6.3.1 Vstupní vrstva

Vstupní vrstva zajišťuje komunikaci s externími zdroji dat. Obsahuje moduly pro komunikaci s API GitHubu, Team API a pro načítání dat z git repozitáře.

Vrstva se stará o načítání issues a PR z API GitHubu a taktéž o mapování změněných souborů k PR z lokální kopie git repozitáře, pokud je daný PR sloučen.

6.3.2 Zpracování a agregace

Tento modul zajišťuje zpracování, transformaci a obecnou orchestraci dat získaných ze vstupní vrstvy. Dochází zde ke sloučení dat například z API GitHubu a Team API, aby bylo možné přiřadit přispěvatele z Gitu do týmů z Team API.

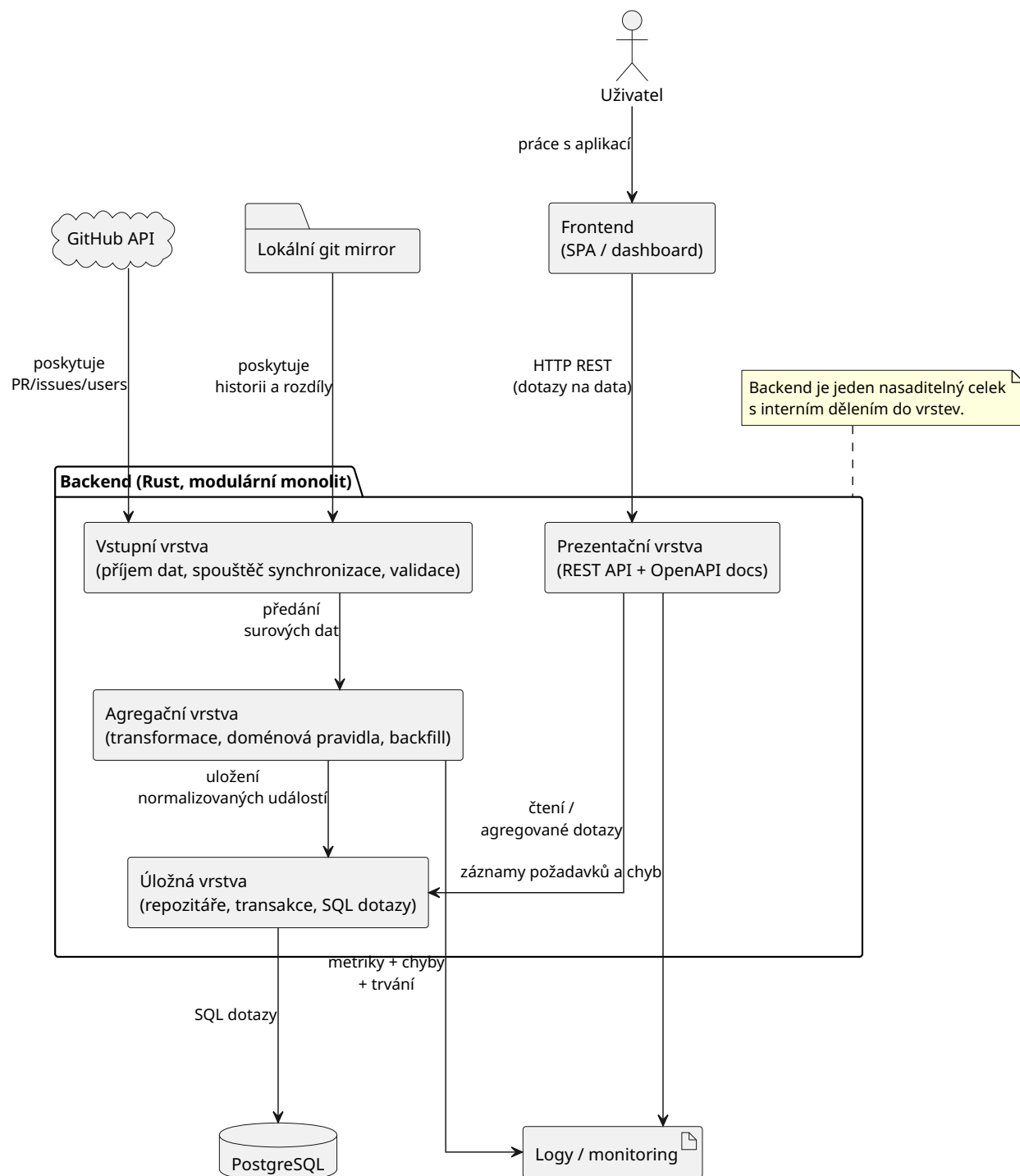
Tato vrstva rovněž transformuje data pro vložení do databáze a zajišťuje jejich konzistentní stav. Probíhá zde také měření doby trvání jednotlivých operací, jako je načítání dat ze vstupní vrstvy nebo vkládání dat do databáze.

6.3.3 Úložná vrstva

Úložná vrstva zajišťuje ukládání normalizovaných dat do databáze. Poskytuje rozhraní pro ukládání a načítání dat určené pro vrstvu zpracování a prezentační vrstvu.

Jsou zde využívány tzv. upserty pro zajištění konzistence dat a jejich aktuálního stavu. Tato vrstva dále zajišťuje migrace schématu a evoluci databáze a indexů v průběhu vývoje systému.

Architektura systému (zobecněná verze)



Obrázek 6.1: Celková architektura systému

6.3.4 Prezentační vrstva

Prezentační vrstva zajišťuje vystavení dat uživatelům. Obsahuje backendové API poskytující endpointy pro frontend a samotnou aplikaci Vue.js, která data zobrazuje.

API je navrženo podle principů REST a poskytuje přímé endpointy pro frontend. Je rozděleno do modulů podle případů užití nebo typu požadovaných dat. Endpointy jsou zdokumentovány a obsahují příklady odpovědí i chybových stavů s možností spustit vlastní požadavek přímo na daný endpoint. Endpointy primárně volají úložnou vrstvu pro získání dat, jedná se tedy o tenkou vrstvu nad databází.

6.3.5 Celkové vlastnosti systému

První vlastností je pozorovatelnost a monitorování průběžného chování systému. Systém měří dobu trvání nejdůležitějších operací, což umožňuje identifikovat výkonnostní úzká hrdla (*bottlenecks*). Dlouhotrvající procesy jsou navíc opatřeny vizualizací průběhu v reálném čase, je-li to technicky možné. Systém také samostatně zaznamenává důležité události a chyby pro pozdější analýzu a ladění.

Druhou vlastností je robustnost a odolnost vůči chybám a výpadkům. Systém je navržen tak, aby při chybě ať už z externích API nebo z interní databáze, zaznamenal vzniklou chybu a pokusil se pokračovat v běhu. Tato vlastnost společně s možností opětovného načtení dat po výpadku zajišťuje, že systém zůstává funkční a poskytuje aktuální data i při dočasných problémech s externími zdroji nebo interních chybách.

6.4 Volba databázového systému

Při návrhu bylo nutné zvolit konkrétní systém pro správu databáze. Nejprve bylo rozhodnuto o typu databáze, a poté byl vybrán konkrétní systém.

6.4.1 Relační vs. dokumentová databáze

Prvním rozhodnutím byl výběr mezi dokumentovou a relační databází. Dokumentová databáze nabízí větší flexibilitu a umožňuje ukládat celé JSON odpovědi bez striktní struktury, avšak pro účely tohoto systému byla zvolena relační databáze.

Důvodem je analytická povaha dotazů a potřeba agregovat data přes více entit. Relační databáze umožňuje efektivní spojování tabulek a skupinové operace, které jsou v nich dobře optimalizovány. Díky primárním a cizím klíčům navíc zajišťuje integritu dat a předem definovaná struktura zjednodušuje validaci a konzistenci na straně aplikace.

6.4.2 SQLite

SQLite je odlehčený souborový databázový systém bez samostatného serverového procesu [24]. Celá databáze je uložena v jediném souboru a databázový stroj je vložen přímo do aplikace jako knihovna. Tím odpadá nutnost instalace serveru, čímž se výrazně zjednodušuje lokální vývoj a nasazení. SQLite je proto vhodné zejména pro prototypování, nízkou souběžnost nebo jako vestavěná databáze mobilních a desktopových aplikací [25].

Nevýhodou je omezená podpora souběžných zápisů. Ve výchozím režimu (Rollback Journal) dochází při zápisu k uzamčení celého souboru. Tento problém částečně řeší Write-Ahead Logging (WAL) režim (`PRAGMA journal_mode=WAL;`) [26], který umožňuje souběžné čtení za probíhajícího zápisu, avšak souběžné zápisy jsou stále serializovány. Nasazení ve vícekontejnerovém prostředí je kvůli sdílenému souboru problematické. Moderní cloudové platformy jako *Cloudflare D1*³ nebo *Turso*⁴ tato omezení řeší, avšak SQLite má stále omezenější sadu analytických SQL funkcí.

6.4.3 PostgreSQL

PostgreSQL je plnohodnotný serverový relační databázový systém [27] fungující na architektuře klient-server. Umožňuje souběžný přístup více klientů, pokročilou správu transakcí a efektivní využití systémových prostředků. Lokální nastavení vyžaduje spuštění serveru, které lze zjednodušit kontejnerizací. V cloudu je PostgreSQL dostupný jako spravovaná služba u všech hlavních poskytovatelů (Amazon RDS, Google Cloud SQL, Azure). Oproti vestavěným databázím zavádí architektura klient-server přídatnou síťovou latenci (TCP/IP), která je však při dávkovém zápisu zanedbatelná.

6.4.4 Výsledné rozhodnutí

Pro implementaci byl zvolen PostgreSQL. Rozhodujícím argumentem byla podpora analytického SQL, zejména klauzule `DISTINCT ON` [28], kterou SQLite nenabízí a která zjednodušuje dotazy typu point-in-time pro zjištění stavu každého PR v daném okamžiku. PostgreSQL dále poskytuje pokročilejší okenní funkce [29] pro výpočty nad časovými řadami a propracovaný plánovač dotazů s podporou různých typů indexů. Cloudové platformy moderního ekosystému SQLite tyto analytické konstrukty nativně nepodporují. Zatímco SQLite serializuje souběžné zápisy i v WAL módu, PostgreSQL díky MVCC (multiversion concurrency control) a zamykání na úrovni řádků zvládá zápisy lépe.

Pro lokální vývoj je server spouštěn přes kontejnerizaci, takže vývojář nemusí instalovat PostgreSQL přímo do systému a vývojové prostředí zůstává shodné s produkčním nasazením.

Srovnání obou systémů shrnuje Tabulka 6.1.

³<https://developers.cloudflare.com/d1/>

⁴<https://turso.tech>

Tabulka 6.1: Srovnání vlastností SQLite a PostgreSQL pro účely analytického nástroje

Kritérium	Databázový systém	
	SQLite	PostgreSQL
Architektura	Vestavěná knihovna.	Klient-server architektura.
Lokální nasazení	Zcela bezúdržbové, stačí jeden soubor.	Vyžaduje instalaci serveru nebo spuštění přes kontejnerizaci.
Cloudové nasazení	Tradiční kontejnerové nasazení problematické; moderní platformy (D1, Turso) řeší omezení, avšak s omezenější sadou analytických SQL funkcí.	Nativní podpora, dostupné jako spravovaná služba.
Souběžné zápisy	Ve výchozím režimu zamyká celý soubor; WAL mód serializuje souběžné zápisy.	Plná podpora souběžnosti a nezávislých transakcí.
Analytické dotazy	Chybí <code>DISTINCT ON</code> a pokročilé okenní funkce. Vhodné pro jednodušší dotazy.	Pokročilý optimalizátor, <code>DISTINCT ON</code> , okenní funkce a bohatá podpora datových typů.

6.5 Struktura databáze

Při návrhu struktury databáze bylo klíčové zajistit, aby schéma efektivně pojmul velké množství dat a zároveň umožňovalo rychlé provádění analytických dotazů.

6.5.1 Rozdělení štítků a událostí

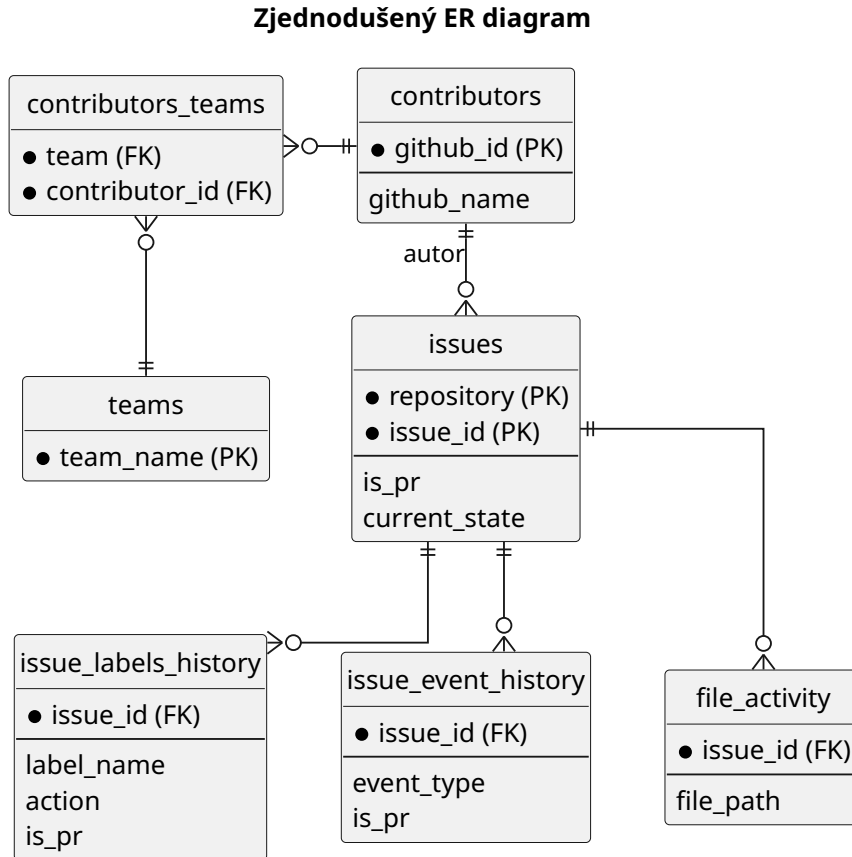
Každý PR a issue má dvě kategorie historických změn. První kategorií jsou *události* – akce jako sloučení, uzavření, znovuotevření, přidání komentáře a podobně, které jsou reprezentovány typem události a časovým razítkem. Druhou kategorií jsou *štítky* – metadata přiřazovaná k PR nebo issue (například `S-waiting-on-review`), u nichž je nutné sledovat štítek, časové razítko a zda byl štítek přidán nebo odebrán. Tyto štítky se sledují především kvůli sledování stavů (viz Sekce 5.2).

Ukládání kompletní historie událostí a štítků je nezbytné pro správné fungování dotazů Q1, Q2 a Q4 (viz Podsekce 6.1.2), které rekonstruují historický stav PR a issues k zadanému datu. Bez těchto dat nelze přesně zpětně určit, v jakém stavu se daný PR nacházel v konkrétním okamžiku.

Protože události a štítky mají odlišné atributy a slouží jiným analytickým účelům, jsou uloženy ve dvou samostatných tabulkách: `issue_event_history` a `issue_labels_history`. Jejich sloučení do jedné tabulky by vyžadovalo volitelné sloupce pro atributy druhého typu, nebo by vedlo k redundanci dat.

6.5.2 Diagram databáze

Celková struktura databáze je znázorněna na Obrázku 6.2. Schéma má tři hlavní logické části: strukturu přispěvatelů a přiřazení do týmů, aktuální stav PR a issues a historii změn issues zahrnující zaznamenané události, změny štítků a soubory upravené v rámci sloučených nebo uzavřených PR.



Obrázek 6.2: Celkový přehled databázového schématu

Podrobnější ER diagram databáze je uveden v příloze (viz Obrázek B.1). Centrem schématu je tabulka `issues`, která obsahuje aktuální stav issue i PR. Tato tabulka je společná pro oba typy záznamů, protože i na straně GitHubu je každý PR zároveň issue a rozlišení probíhá pouze pomocí příznaku. Například u endpointu pro získání issues se rozdíl mezi PR a issue pozná primárně pomocí přítomnosti pole `pull_request` v objektu entity v odpovědi. Kvůli tomuto jedinému rozdílu je na tabulce implementován podmíněný index, konkrétně dva: jeden pro záznamy, kde `is_pr = true`, a druhý, kde `is_pr = false`. Tento index efektivně rozděluje prohledávání tabulky na dvě části, čímž zmenšuje jeho velikost a urychluje vyhledávání.

Do historických entit jsou pouze přidávány nové události a záznamy jsou neměnné. Ostatní záznamy v tabulkách jsou aktualizovány podle posledního známého stavu.

Data o týmech jsou sice dostupná přes Team API (viz Sekce 5.1) jako jednorázová JSON odpověď, avšak pro analytické dotazy Q6 a Q7 (viz Podsekce 6.1.2) je nutné je mít v databázi. Tyto dotazy vyžadují SQL `JOIN` tabulky `file_activity` s přiřazením přispěvatelů do týmů. Bez lokálních dat by bylo nutné při každém dotazu volat Team API a filtrovat výsledky v aplikační vrstvě. Data jsou proto uložena do tabulek `teams` a `contributors_teams` a obnovována při každé synchronizaci, aby odrážela aktuální složení týmů.

Vztah mezi týmem a přispěvatelem je realizován jako vazba m:n přes spojovací tabulku `contributors_teams`, protože každý člen může být ve více týmech a každý tým má více členů.

Ohledně souborů je zaznamenáváno pouze to, kdo daný soubor upravil, konkrétní provedené změny se neukládají. Tabulka `file_activity` slouží jako základ pro dotazy Q5, Q6 a Q7 (viz Podsekce 6.1.2). Cílem těchto dotazů je zjistit, na jakých částech projektu se daný přispěvatel nebo tým podílel, nikoli sledovat konkrétní upravené funkcionality.

6.6 Návrh dotazů

Tato sekce popisuje, jakým způsobem jsou analytické dotazy definované v Podsekci 6.1.2 realizovány. Pro každý dotaz je uvedeno, z jakých datových zdrojů čerpá a které tabulky databáze využívá.

Všechny dotazy jsou striktně čtecí, parametrizované identifikátorem repozitáře a pracují výhradně s daty uloženými v databázi (viz Sekce 6.5). Tím je zaručena nízká latence odpovědí i nad velkým množstvím historických záznamů.

6.6.1 Historická dostupnost dat

Dotazy Q1, Q2 a Q4 pracují s historickým vývojem PR a issues. Aby bylo možné tyto dotazy spolehlivě zodpovědět, musí mít systém v databázi kompletní sekvenci událostí pro každý PR a issue, včetně změn štítků a stavů. Nestačí tedy ukládat pouze aktuální stav, je nutné uchovávat celou historii změn.

6.6.2 Realizace jednotlivých dotazů

Q1 – Historický stav PR k danému datu Dotaz rekonstruuje stavy PR v zadaném časovém rozmezí. Data jsou čerpána z REST API GitHubu (viz Podsekce 3.3.1), konkrétně z endpointu pro Timeline events, kde jsou zaznamenávány stavové štítky začínající prefixem `S-*` (viz Sekce 5.2). V databázi dotaz využívá tabulky `issues`, `issue_labels_history` a `issue_event_history`.

Q2 – Počet PR v konkrétním stavu k datu Dotaz počítá PR v daném stavu k zadanému datu. Využívá tytéž datové zdroje jako Q1. Pracuje s tabulkami `issues` a `issue_labels_history`. Pro každý PR je potřeba určit jeho poslední známý stavový štítek před zadaným datem.

Opakovaným voláním tohoto dotazu pro různá data lze na prezentační vrstvě sestavit časovou řadu vývoje počtu PR v konkrétním stavu.

- Q3 – PR čekající na zpracování** Na rozdíl od Q1 a Q2 tento dotaz nepracuje s historickými daty, ale s aktuálním stavem. V databázi vyhledává PR, jejichž poslední stavový štítek odpovídá jednomu ze tří čekacích stavů. Využívá tabulky `issues` a `issue_labels_history`.
- Q4 – Události konkrétní issue k datu** Data pro tento dotaz pocházejí z Timeline API GitHubu (viz Podsekce 3.3.1), kde jsou zaznamenány akce jako uzavření, znovuotevření nebo komentáře. V databázi dotaz využívá tabulky `issues` a `issue_event_history`.
- Q5 – Nejčastěji upravované soubory přispěvatelem** Data o aktivitě nad soubory plynou z lokálního git repozitáře (viz Sekce 2.3). Pro každý merge commit spojený s PR jsou extrahovány dotčené soubory a uloženy do tabulky `file_activity` spolu s identifikátorem přispěvatele. Dotaz dále využívá tabulku `contributors` pro metadata přispěvatele.
- Q6 – Přispěvatelé, kteří upravili soubor nebo složku** Tento dotaz nabízí inverzní pohled k Q5. Cesta je vyhledávána jako prefix, takže dotaz pokrývá jak konkrétní soubor, tak celý adresář. Data pocházejí ze všech tří hlavních zdrojů: lokálního git repozitáře (viz Sekce 2.3), REST API GitHubu (viz Podsekce 3.3.1) a Team API (viz Sekce 5.1). V databázi využívá tabulky `file_activity`, `contributors` a `contributors_teams`.
- Q7 – Soubory upravené členy týmu** Tento dotaz kombinuje data ze všech tří zdrojů. V databázi vyžaduje propojení tabulek `file_activity`, `contributors_teams` a `teams`. Výsledek lze vrátit jako plochý seznam, stromovou strukturu seskupenou do zvolené hloubky, nebo jako úplný adresářový strom. Tato flexibilita umožňuje prezentační vrstvě (viz Podsekce 6.3.4) přizpůsobit vizualizaci jak detailním grafům na úrovni souborů, tak přehledovým heatmapám celých modulů.

Kapitola 7

Implementace

Tato kapitola popisuje aspekty implementace nástroje. Nejprve je popsáno schéma databáze, dále jsou vysvětleny přístupy k efektivnímu načítání dat z GitHub API a lokálního git repozitáře. Závěr kapitoly se věnuje vizualizaci dat prostřednictvím vlastního webového rozhraní.

7.1 Popis aplikace

Jedná se o serverovou aplikaci napsanou v jazyce Rust (edice 2021) [30], jejímž účelem je sběr, uložení a analýza dat z GitHub repozitářů. Aplikace pracuje s databází PostgreSQL a vystavuje REST API, přes které jsou data zpřístupněna frontendové části. Architektura systému je podrobněji popsána v Sekci 6.3.

Aplikace je taktéž rozdělena do modulů podle funkcionality a dá se zkompileovat i jako knihovna, která poskytuje jednotlivé funkcionality. Vysokoúrovňové funkcionality jsou synchronizace dat, poskytování API a zpracování dat pro analytické dotazy a veškerá funkcionality jako je v aplikaci.

Backendová část je kompilována do jednoho binárního souboru se třemi provozními režimy vybíranými pomocí Command Line Interface (CLI) podpříkazů. Frontendová část je webová Single-Page Application (SPA) aplikace postavená na frameworku Vue 3, která komunikuje s REST API backendem a zobrazuje data uživatelům v podobě interaktivních grafů a tabulek.

7.1.1 Technologie

Backend je postaven na asynchronním runtime *Tokio*¹, který zajišťuje souběžné zpracování HTTP požadavků a paralelní běh synchronizační smyčky v pozadí. Jako HTTP framework je použita

¹<https://tokio.rs>

knihovna *Axum*², která je postavena nad Tokio a ve spojení s knihovnou *Aide*³ automaticky generuje dokumentaci API ve formátu *OpenAPI*⁴.

Pro přístup k databázi je použita knihovna *sqlx*⁵, která umožňuje psát SQL dotazy přímo v kódu a zároveň je ověřuje oproti databázovému schématu již při kompilaci. Tím je zajištěno, že typové nesrovnalosti mezi dotazem a datovým modelem jsou odhaleny dříve, než je aplikace spuštěna.

Přehled dalších klíčových závislostí je uveden v Tabulce 7.1.

Tabulka 7.1: Klíčové závislosti backendu aplikace

Knihovna	Verze	Role v systému
tokio	1.48	Asynchronní runtime (multi-vláknový)
axum	0.8	HTTP framework pro REST API
sqlx	0.8	Asynchronní PostgreSQL driver, migrace
clap	4.5	Parsování CLI argumentů
octocrab	0.49	Klient pro GitHub REST API
git2	0.20	Lokální git operace
rust_team_data	–	Data o členech týmů Rust projektu [19]
aide	0.15	Automatická generace OpenAPI dokumentace
backoff	0.4	Exponenciální zpoždění opakování požadavků

Jako databáze je použit PostgreSQL 14 (Alpine), spouštěný prostřednictvím Docker Compose na portu 5431. Schéma databáze je spravováno prostřednictvím migrací *sqlx* a je podrobněji popsáno v Sekci 7.2.

7.1.2 Režimy spuštění

Aplikace je ovládána přes CLI a rozlišuje tři provozní režimy, volené pomocí podpříkazů:

sync-all -sync <N|YYYY-MM-DD> [-full-history] Proveďte počáteční nebo inkrementální načtení dat z GitHub API. Parametr **-sync N** načte posledních *N* stránek výsledků (každá stránka obsahuje 100 záznamů). Parametr **-sync YYYY-MM-DD** načte všechny záznamy aktualizované od zadaného data. Příznak **-full-history** navíc pro každý PR a issue stáhne kompletní historii událostí (štítky, události, sloučení apod.) z GitHub timeline API.

backfill Doplní historii událostí pro záznamy, které byly uloženy do databáze bez plné historie — například při prvotní synchronizaci bez příznaku **-full-history**. Podpříkaz vyhledá záznamy z tabulky **issues** bez záznamů v tabulce **issue_event_history** a pro každý z nich zpětně stáhne celou historii z GitHub API. Podrobněji je tento mechanismus popsán v Podsececi 7.3.1.

²<https://github.com/tokio-rs/axum>

³<https://docs.rs/aide/latest/aide/>

⁴<https://learn.openapis.org/>

⁵<https://github.com/launchbadge/sqlx>

`serve` Spustí REST API server naslouchající na konfigurovatelné adrese (výchozí nastavení `0.0.0.0:7878`). Zároveň je spuštěna synchronizační smyčka (`StateMonitor`), která v intervalu nakonfigurovaných sekund automaticky načte a uloží záznamy aktualizované od posledního běhu (viz Sekce 7.3.1). Dokumentace API je dostupná na URL `/docs` (Scalar), `/docs/redoc` (Redoc) a `/docs/swagger` (Swagger UI). V tomto režimu s API serverem komunikuje frontendová SPA aplikace postavená na Vue 3.

7.2 Implementace databázového schématu

Tato sekce popisuje konkrétní implementační rozhodnutí při tvorbě databázového schématu. Navazuje na návrh popsany v Sekci 6.5 a rozvádí detaily, které se týkají volby primárních klíčů, unikátních omezení, indexů a dalších databázových entit.

7.2.1 Primární klíče

Tabulka `issues` používá složený přirozený klíč `(repository, issue)`, protože číslo issue je unikátní pouze v rámci jednoho repozitáře. Zahrnutí sloupce `repository` do primárního klíče umožňuje, aby systém od začátku podporoval sledování více repozitářů bez nutnosti změny schématu.

Historické tabulky `issue_event_history`, `issue_labels_history` a `file_activity` oproti tomu používají umělý klíč `id` typu `SERIAL`. Přirozený klíč by zde nebyl efektivní. Umělý klíč zjednodušuje cizí klíče v případných navazujících tabulkách a zrychluje interní operace databáze s řádky.

7.2.2 Unikátní omezení na historických tabulkách

I přes použití umělého klíče jsou na historických tabulkách definována unikátní omezení podle přirozeného klíče záznamu. Konkrétně:

- `issue_event_history`: `(repository, issue, timestamp, event)`
- `issue_labels_history`: `(repository, issue, timestamp, label)`
- `file_activity`: `(repository, issue, timestamp, file_path)`

Tato omezení zabraňují vzniku duplicitních záznamů při opakovaném spuštění synchronizace. Pro vkládání dat je využíván mechanismus *upsert* (`INSERT ... ON CONFLICT DO NOTHING`), který při konfliktu unikátního omezení záznam přeskočí bez chyby. Za druhé omezení slouží jako přirozené indexy pro dotazy filtrující podle uvedených sloupců.

7.2.3 Denormalizace sloupce `is_pr`

Sloupec `is_pr` je přítomen nejen v tabulce `issues`, ale také ve všech historických tabulkách. Jde o záměrnou denormalizaci. Při analytických dotazech, které pracují pouze s historickými událostmi nebo změnami labelů, je tak možné filtrovat záznamy podle typu (issue vs. PR) bez nutnosti provádět `JOIN` s tabulkou `issues`. Tento přístup snižuje složitost dotazů a zlepšuje jejich výkon pro často prováděné dotazy.

7.2.4 Volitelné sloupce (Nullable)

Některá data nemusí být z GitHubu vždy dostupná, schéma proto v tabulce `issues` využívá volitelné sloupce:

`contributor_id` Může být prázdný (`NULL`), pokud autor svůj GitHub účet smazal nebo anonymizoval. Systém tak dokáže uložit záznam i bez známého autora.

`merge_sha` Obsahuje hash commitu pouze u úspěšně sloučených (merged) pull requestů. Pokud je PR stále otevřený nebo byl zavřen, zůstává hodnota `NULL`.

7.2.5 Hierarchie týmů

Tabulka `teams` obsahuje sloupec `subteam_of`, který je cizím klíčem odkazujícím na stejnou tabulku. Tímto způsobem je implementována stromová hierarchie týmů, kdy jeden tým může být podtýmem jiného. Aby nedošlo k nevalidnímu stavu, kdy by tým byl sám sobě nadřazeným týmem, je na tabulce definováno `CHECK` omezení `team != subteam_of`.

7.2.6 Indexy pro analytické dotazy

Kromě podmíněných indexů nad sloupcem `is_pr` v tabulce `issues`, popsanych v Sekci 6.5, jsou definovány další indexy optimalizující analytické dotazy.

Na historických tabulkách jsou indexovány sloupce `timestamp` a `repository`, protože analytické dotazy téměř vždy filtrují záznamy podle časového rozsahu nebo konkrétního repozitáře. Bez těchto indexů by databáze musela provádět sekvenční průchod celých tabulek.

Na tabulce `file_activity` je navíc definován index nad sloupcem `file_path`, protože dotazy identifikující přispěvatele podle konkrétního souboru nebo skupiny souborů patří mezi nejčastěji prováděné analytické operace.

7.2.7 Absence ORM

Schéma bylo vytvořeno a je spravováno přímo pomocí SQL skriptů bez použití Object-Relational Mapping (ORM) vrstvy. Tento přístup byl zvolen záměrně, protože analytické dotazy jsou v mnoha případech příliš komplexní a specifické pro abstrakce, které ORM poskytuje. Přímé SQL umožňuje

plně využít specifické funkce PostgreSQL, jako jsou podmíněné indexy, `INSERT ... ON CONFLICT` nebo agregační *window functions*, které by v ORM vyžadovaly nestandardní rozšíření nebo ruční psaní SQL.

7.3 Specifika synchronizace dat

Tato sekce popisuje konkrétní implementační přístupy a úskalí při synchronizaci dat z GitHubu a z lokálního git repozitáře.

7.3.1 Inkrementální načítání

Při prvním spuštění synchronizace je nutné načíst historická data. Repozitář `rust-lang/rust` v době psaní této práce obsahuje přibližně 92 000 PR a 61 000 issues. Opakovat takový plný průchod při každém obnovení dat by bylo jak časově nákladné, tak zbytečně zatěžující z pohledu počtu požadavků na API GitHubu.

Z tohoto důvodu systém rozlišuje dva režimy synchronizace (`SyncMode`):

- **Last(N)** – načte posledních *N* stránek výsledků (PR nebo issues), seřazených sestupně podle data vytvoření. Slouží pro rychlé doplnění nejnovějších záznamů nebo při prvotním spuštění.
- **Since(datum)** – načítá záznamy seřazené sestupně podle časového razítka a ukončí se ve chvíli, kdy narazí na záznamy starší než zadané datum. Tento režim je využíván při průběžné synchronizaci v pozadí.

V rámci serveru běží na pozadí smyčka (`StateMonitor`), která se každých 60 sekund spustí a zavolá synchronizaci v režimu `Since(last_known_event_timestamp)`, kde jako počáteční datum slouží časová značka posledního záznamu v tabulce `issue_event_history`. Tím je zajištěno, že jsou průběžně doplňovány pouze záznamy, které se od posledního spuštění změnil.

Pro případ, že od počátečního naplnění databáze chybí historické záznamy (například protože při prvním synchronizaci nebyla volána možnost `-full-history`), je k dispozici příkaz `backfill`. Ten vyhledá záznamy v tabulce `issues`, ke kterým dosud v tabulce `issue_event_history` neexistují žádné řádky, a pro každý takový záznam zpětně načte celou historii událostí z GitHub API.

7.3.2 Limity GitHub API a jejich vliv na návrh

GitHub REST API pro autentizované uživatele umožňuje 5 000 požadavků za hodinu [10]. V kontextu průběžné synchronizace, kdy jsou každých 60 sekund zasílány pouze požadavky pro záznamy s nedávnou aktivitou, je reálné zatížení výrazně pod tímto limitem, a to v řádu desítek až stovek požadavků za hodinu. Limity tedy nejsou při normálním provozu překračovány.

Přesto je v implementaci přítomna ochrana pro případ rozsáhlé počáteční synchronizace nebo zpětného doplnění (`backfill`), kdy se může počet požadavků přiblížit limitům. Pro každý požadavek na API GitHubu je použita funkce `retry_on_rate_limit`, která detekuje chyby odpovídající překročení limitu (HTTP status 429, řetězce `"rate limit"`, `"too many requests"` nebo `"secondary rate"` v chybové zprávě) [10] a automaticky opakuje požadavek s exponenciálním zpožděním. Počáteční čekání je 5 sekund, maximální prodleva je 60 sekund a celková maximální doba opakování je 5 minut. Pokud se požadavek nezdaří ani po uplynutí této doby, chyba je vyhozena do konzole. Samotná synchronizace běží asynchronně zároveň s API aplikace pro frontend, tento timeout tedy není blokující pro běžný běh aplikace. Tento mechanismus slouží jako pojistka, nikoli jako primární strategie pro správu limitu.

7.3.3 Timeline events vs. webhooks

GitHub nabízí dva přístupy k získání průběžné historie změn PR a issues: webhooks a timeline events API.

Webhooks fungují jako push notifikace. GitHub při každé relevantní události zašle HTTP požadavek na předem registrovaný endpoint. Tento přístup je vhodný pro aplikace, které potřebují reagovat na události v reálném čase [15]. Pro účely této práce je však nevhodný ze dvou důvodů. Za prvé, webhook vyžaduje veřejně dostupný endpoint, což komplikuje lokální vývoj a testování. Za druhé, webhooks neposkytují historická data, takže jimi nelze zpětně získat události, ke kterým došlo před registrací webhooku.

Systém proto používá timeline events API, které pro každý PR nebo issue vrátí seznam všech zaznamenaných událostí v chronologickém pořadí. Tím je možné jak zpětně naplnit historii, tak průběžně doplňovat nové události. Načítání je optimalizováno tak, že pro každý záznam je kontrolována časová značka posledního uloženého záznamu v databázi. Pokud načtená stránka událostí obsahuje záznamy s časovou značkou starší než tato hodnota, načítání je ukončeno.

7.3.4 Kombinace GitHub API a git repozitáře

Informace o souborech změněných v rámci sloučeného PR jsou dostupné přes API GitHubu prostřednictvím endpointů pro výpis souborů daného PR [10]. Použití tohoto endpointu by však při zpracování tisíců PR přineslo výrazné navýšení počtu požadavků na API, a taktéž by bylo pomalé kvůli síťové latenci.

Z tohoto důvodu jsou informace o změněných souborech získávány přímo z lokální kopie git repozitáře pomocí knihovny `git2` [31]. Pro každý sloučený PR je znám SHA merge commitu, který je uložen v tabulce `issues`. Pomocí funkce `diff` je tento commit porovnán s jeho prvním rodičovským commitem, čímž vzniká seznam souborů změněných v rámci daného PR [4, 31]. Tento přístup nevyžaduje žádné síťové požadavky na API GitHubu a je výrazně rychlejší, protože git repozitář je k dispozici lokálně.

Nevýhodou lokálního přístupu je nutnost mít k dispozici naklonovaný repozitář `rust-lang/rust`, který má v době psaní této práce velikost přibližně 6 GB (samotná git historie bez zdrojových souborů). Systém repozitář při prvním spuštění automaticky naklonuje a při každé synchronizaci stáhne nejnovější změny pomocí `git fetch`.

7.4 Zobrazování dat

Tato sekce popisuje implementaci vizualizací navržených v Sekci 6.2.4.

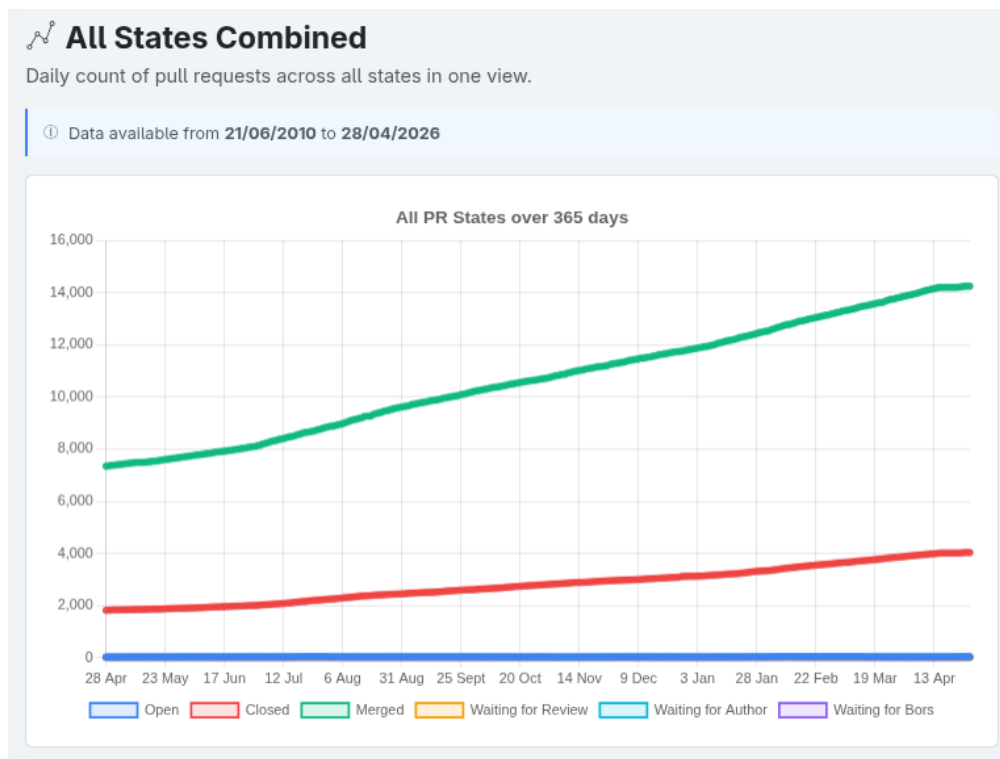
Hlavní vizualizací je zobrazení vývoje stavu PR v čase (*PR State Over Time*), zachycené na Obrázku 7.1. Uživatel zvolí repozitář, stav PR, kotevní datum a počet zpětně zobrazovaných dní. Na základě těchto parametrů je zavolán endpoint `/api/pr/prs-in-state-over-time`, který vrátí denní počty PR v daném stavu. Nad grafem jsou zobrazeny souhrnné statistiky za zvolené období: aktuální hodnota, průměr, minimum, maximum a trend (rozdíl mezi aktuální hodnotou a průměrem).



Obrázek 7.1: Zobrazení vývoje počtu PR ve stavu „Waiting for Review“ za posledních 365 dní

Druhým pohledem je souhrn všech stavů naráz (*All States Combined*), zobrazený na Obrázcích 7.2 a 7.3. V tomto režimu jsou na jediném grafu vyneseny křivky pro všechny stavy současně. Uživatel může jednotlivé křivky interaktivně skrývat a zobrazovat kliknutím na jejich popisek v legendě. Tato funkce je zvláště užitečná při porovnání stavů s výrazně odlišnými hodnotami na ose y — například

skrytím monotónně rostoucích křivek *Open*, *Closed* a *Merged* se odhalí detailnější průběh stavů *Waiting for Review*, *Waiting for Author* a *Waiting for Bors*, jak ukazuje Obrázek 7.3.



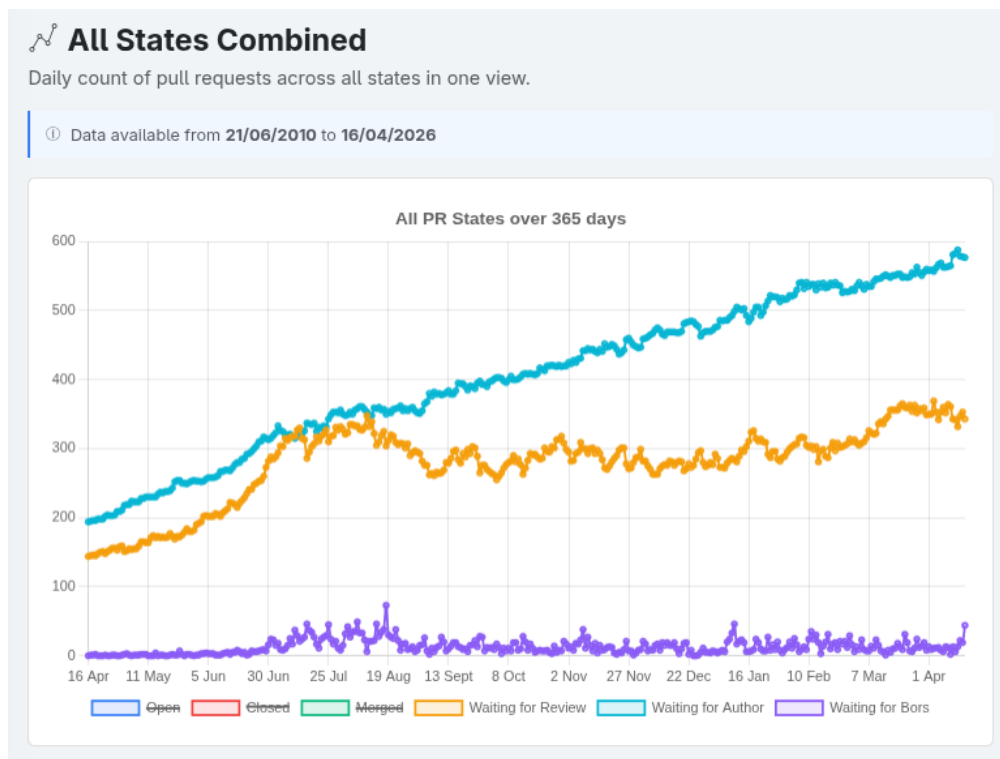
Obrázek 7.2: Souhrn všech stavů PR za posledních 365 dní — plné zobrazení

Další vizualizací jsou soubory upravené členy daného týmu v zadaném časovém okně, dostupné přes endpoint `/api/pr/files-modified-by-team`. Uživatel zadá název repozitáře, název týmu, kotvení datum a počet dní pro zpětný pohled. Výsledkem je seznam souborů seřazený sestupně podle počtu úprav. Klíčovým parametrem je `group_level`, který určuje způsob seskupení výsledků:

- `none` (výchozí) — plochý seznam jednotlivých souborů s počtem úprav,
- číslo (např. `1`, `2`) — seskupení do adresářové hierarchie do zadané hloubky; hodnota `1` seskupí soubory do složek nejvyšší úrovně (`src/`, `library/` apod.), hodnota `2` dále rozdělí výsledky na podadresáře (`src/doc/`, `library/core/` apod.),
- `all` — úplná adresářová hierarchie rekurzivně až na úroveň jednotlivých souborů.

Možnost plochého seznamu je zahrnuta z důvodu, že v takovém případě není nutná transformace dat z databáze do stromové struktury. Splošťování stromové struktury na straně klienta by navíc bylo pomalejší než přímé zobrazení dat ve správném formátu.

Díky parametru `group_level` lze snadno identifikovat, na které části projektu se daný tým v daném období zaměřoval, aniž by bylo nutné procházet stovky záznamů v plochem seznamu.

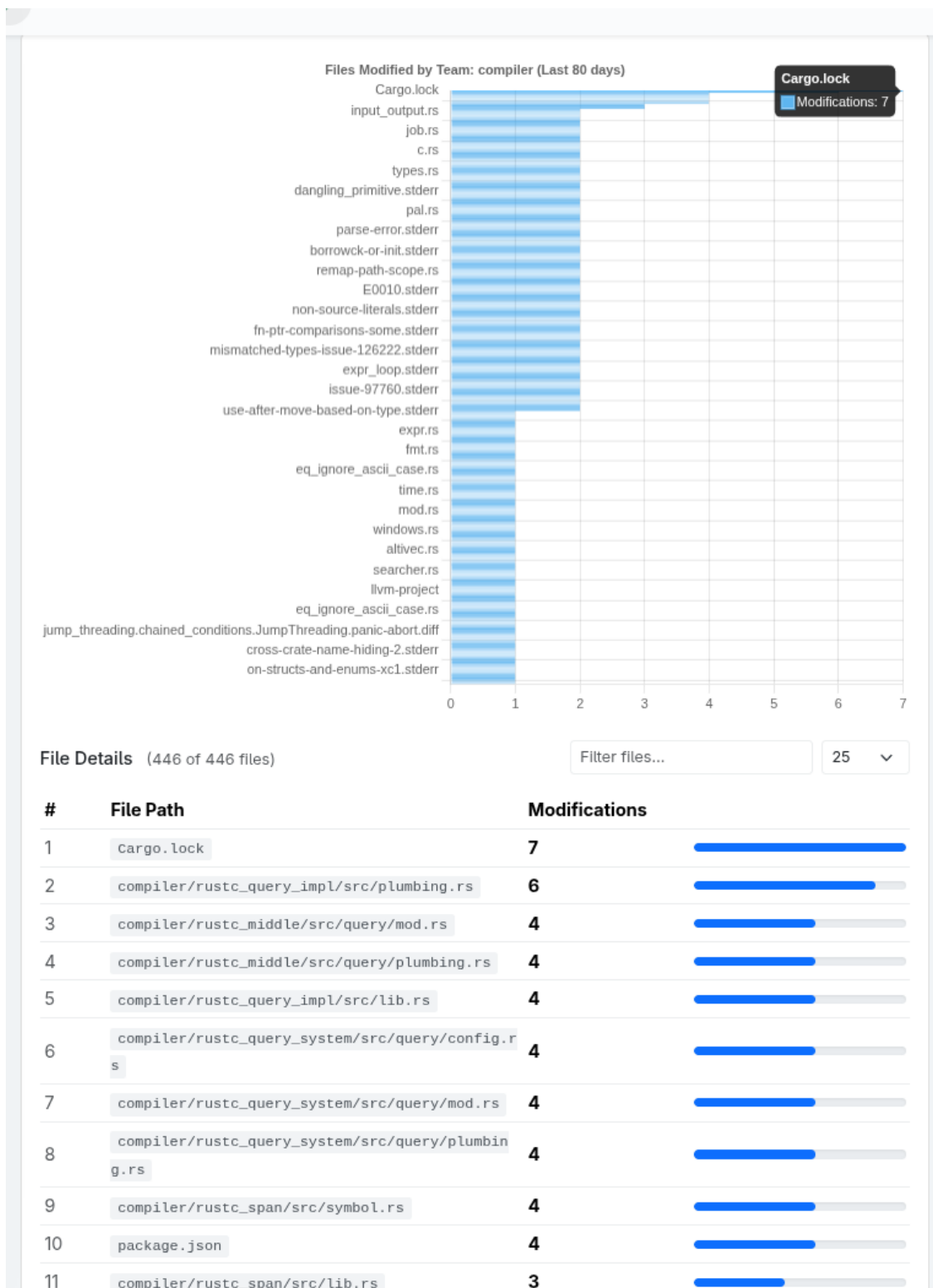


Obrázek 7.3: Souhrn stavů PR po skrytí monotónně rostoucích křivek — detail PR čekajících na akci

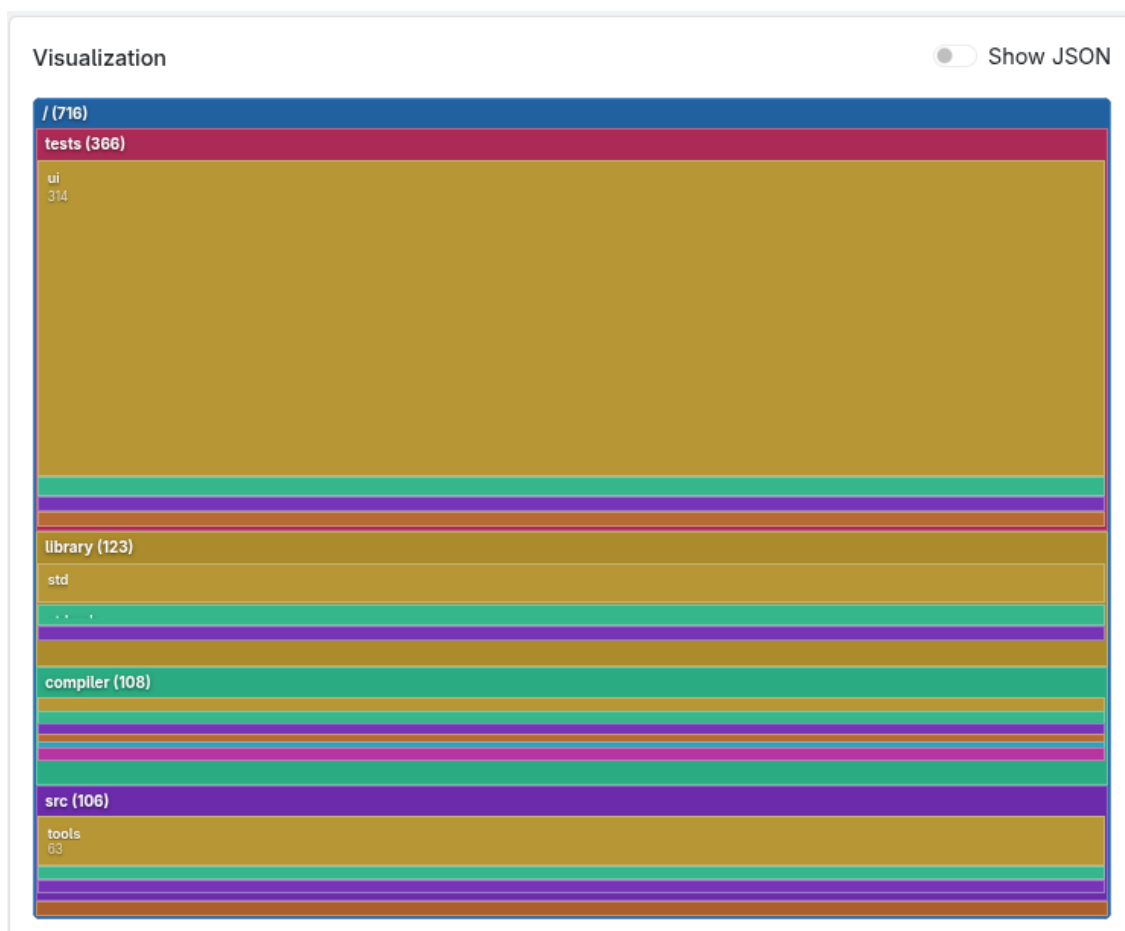
Výchozí zobrazení bez seskupení (`group_level=none`) je zachyceno na Obrázku 7.4. Výsledky jsou prezentovány jako horizontální sloupcový graf doplněný tabulkou se seřazeným seznamem souborů a jejich počty úprav.

Při seskupení s parametrem `group_level=all` je výsledná stromová struktura vizualizována jako treemap, viz Obrázek 7.5. Plocha každého bloku odpovídá relativnímu počtu úprav dané složky nebo souboru vůči ostatním. Kliknutím na blok lze přejít do příslušné podúrovně hierarchie. Díky tomu lze na první pohled odhalit, které části repozitáře jsou daným týmem nejaktivněji upravovány.

Frontend navíc umožňuje přepnout do režimu zobrazení surové JSON odpovědi z API, jak ukazuje Obrázek 7.6. Tento pohled slouží zejména pro ladění a ověřování správnosti vrácené stromové struktury.



Obrázek 7.4: Soubory upravené týmem „compiler“ za posledních 80 dní — plochý seznam (`group_level=none`)



Obrázek 7.5: Soubory upravené týmem — hierarchická vizualizace (treemap, `group_level=all`)

Visualization Show JSON

```
▼{
  modifications: 716,
  name: "/",
  ▼children: [
    ▼{
      modifications: 366,
      name: "tests",
      ▼children: [
        ▼{
          modifications: 314,
          name: "ui"
        },
        ▼{
          modifications: 20,
          name: "codegen-llvm"
        },
        ▼{
          modifications: 16,
          name: "ui-fulldeps"
        },
        ▶{ ... },
        ▶{ ... },
        ▶{ ... },
        ▶{ ... },
        ▶{ ... }
      ]
    },
    ▼{
```

Obrázek 7.6: Surová JSON struktura stromové hierarchie souborů vrácená API

Kapitola 8

Evaluace

V této kapitole je probráno zda systém splňuje požadavky primárně co se výkonu týče.

8.1 Výkonnostní měření synchronizace

Měření výkonu synchronizace dat pro repozitář `rust-lang/rust` probíhalo na lokální pracovní stanici. Pro zajištění reprodukovatelnosti experimentů je níže uvedena hardwarová a síťová specifikace testovacího prostředí:

- **Procesor:** AMD Ryzen 5 5600G (6 jader, 12 vláken, základní frekvence 3,9 GHz)
- **Operační paměť:** 32 GB DDR4 (3 600 MHz)
- **Úložiště:** 512 GB NVMe SSD (KXG50ZNV512G, 3200 MBps čtení / 1050 MBps zápis)
- **Připojení k síti:** Symetrické internetové připojení s propustností 300 Mb/s

Časová náročnost synchronizace je úzce spjata s propustností disku a parametry sítě. Získané výsledky proto slouží primárně k relativnímu porovnání v rámci této konkrétní konfigurace a absolutní hodnoty se mohou při použití odlišného hardwaru měnit.

8.1.1 Počáteční inicializace dat

Počáteční synchronizace slouží k prvotnímu stažení všech dat z repozitáře za zvolené časové období. Měření bylo provedeno pro data od 1. 1. 2023, tedy za přibližně 3 roky a 4,5 měsíce. Celková doba počáteční synchronizace činila přibližně 49 minut. Podrobný přehled jednotlivých fází je uveden v Tabulce 8.1.

Za uvedené období bylo staženo přibližně 64 000 PR a issues, k čemuž bylo zapotřebí přibližně 640 API požadavků na GitHub (při maximální velikosti stránky 100 záznamů). Synchronizace dat

Tabulka 8.1: Časy jednotlivých fází počáteční synchronizace (rust-lang/rust, od 2023-01-01)

Fáze	Trvání
Klonování git repozitáře	1 min 20 s
Stahování PR z GitHub API	15 min 35 s
Vložení přispěvatelů z PR do databáze	< 1 s
Zpracování PR a výpočet aktivity souborů	15 min 59 s
Hromadné vložení PR do databáze	2 s
Hromadné vložení aktivity souborů do databáze	15 s
Stahování issues z GitHub API	17 min 17 s
Vložení přispěvatelů z issues do databáze	< 1 s
Vložení issues do databáze	1 s
Synchronizace týmů	< 1 s
Celkem	≈ 49 min

týmů vyžadovala jediný požadavek na API rust týmu (viz Sekce 5.1), který nepodléhá běžnému limitu.

Stahování PR a issues by bylo teoreticky možné urychlit paralelizací požadavků. Vzhledem k limitům počtu požadavků (viz Sekce 7.3.2) a riziku dočasného zablokování přístupového tokenu bylo zvoleno sekvenční zpracování. Počáteční synchronizace se navíc v ideálním případě provede pouze jednou při prvním nasazení, takže tato volba nemá zásadní vliv na běžný provoz systému.

8.1.2 Inkrementální aktualizace

Inkrementální aktualizace je realizována pomocí timestampu poslední již uložené události v databázi. Systém si při zahájení synchronizace načte nejnovější známý timestamp a následně volá výpisy pull requestů a issues seřazené podle času poslední změny v sestupném pořadí z GitHub API. Zpracovávány jsou pouze ty stránky výsledků, jejichž položky mohou ještě obsahovat záznamy novější než uložené mez. Při dosažení starších položek se procházení ukončí. Detailní historie událostí a změn štítků je potom stahována už jen pro takto vybrané změněné pull requesty a issues, přičemž i zde lze načítání ukončit po dosažení již dříve uložených událostí.

Po dokončení počáteční synchronizace systém provádí inkrementální aktualizace, při nichž jsou z GitHub API staženy pouze záznamy změněné od posledního běhu. Výpis 8.1 zachycuje log inkrementální aktualizace v situaci, kdy nebyly nalezeny žádné nové záznamy.

Celý cyklus inkrementální aktualizace trval přibližně 6,3 sekundy. Největší část vykazovalo dotazování GitHub API, konkrétně přibližně 3,7 sekundy pro PR a 2,5 sekundy pro issues. V případě nových záznamů přibývá průměrně 1,5 dodatečného API požadavku na každý nový PR nebo issue, a to z důvodu stahování kompletní historie událostí a štítků.

Po dokončení každého synchronizačního cyklu jsou aktualizovaná data okamžitě dostupná v dashboardu. Maximální prodleva mezi vznikem změny v repozitáři a jejím zobrazením odpovídá

```
Getting pull requests from github: took: 3.653 seconds
Inserting contributors from pull requests: took: 0.000 seconds
Inserting pull requests to database: took: 0.000 seconds
Getting issues from github: took: 2.473 seconds
Inserting contributors from issues to database: took: 0.000 seconds
Inserting issues to database: took: 0.000 seconds
Upserting users from rust teams: took: 0.069 seconds
Overall getting resources: took: 6.312 seconds
```

Výpis 8.1: Log inkrementální aktualizace bez nových záznamů

nakonfigurovanému intervalu mezi synchronizacemi. Při výchozím nastavení intervalu na 2 minuty je tato prodleva v nejhorším případě 2 minuty.

8.1.3 Spotřeba API požadavků

GitHub omezuje počet API požadavků na 5 000 za hodinu pro autentizované požadavky [13] (viz Podsekke 3.3.3). Při standardním provozu spotřebuje systém minimálně 60 požadavků za hodinu, přičemž se vychází z přibližně 2 požadavků na jeden synchronizační cyklus (jeden pro PR, jeden pro issues) při intervalu 2 minuty (30 cyklů za hodinu). K tomu je nutné připočítat průměrně 1,5 požadavku za každý nový PR nebo issue potřebný k synchronizaci historie. Pro repozitář `rust-lang/rust` většinou přibývá v závislosti na denní době 0 až 5 nových záznamů za hodinu.

Celková průměrná spotřeba při normálním provozu za hodinu je tedy:

$$30 \times 2 + 5 \times 1,5 = 67,5 \text{ požadavků/hod}$$

Tato hodnota představuje přibližně 1,4 % stanoveného limitu, takže systém může bez problémů fungovat i s výrazně kratšími synchronizačními intervaly nebo může být provozováno více instancí souběžně pro různé repozitáře se stejným přístupovým tokenem.

8.2 Měření výkonu analytických dotazů

V rámci evaluace bylo měřeno, jaký vliv mají databázové indexy na dobu vykonávání analytických dotazů. Databáze obsahovala data repozitáře `rust-lang/rust`: přibližně 200 000 záznamů issues a PR, 564 000 záznamů v tabulce `issue_event_history`, 355 000 v `issue_labels_history` a 525 000 v `file_activity`. Měření bylo provedeno pomocí `EXPLAIN ANALYZE` v PostgreSQL.

8.2.1 Testované dotazy

Byly testovány čtyři dotazy (S1–S4) pokrývající typické analytické scénáře. Dotazy jsou uvedeny v Příloze C. Stručný popis každého dotazu:

- S1** Počet PR ve stavu `S-waiting-on-review` k zadanému datu. Dotaz využívá dvě Common Table Expression (CTE) s `DISTINCT ON`: jednu pro poslední label-event a druhou pro poslední stavovou změnu, následované `LEFT JOIN` a filtrací.
- S2** Kumulativní počet sloučených PR k zadanému datu pomocí `COUNT(DISTINCT issue)` s filtrem `event = 'merged'` nad tabulkou `issue_event_history`.
- S3** Soubory upravené přispěvateli z daného týmu v zadaném časovém rozsahu, agregace přes `file_activity` a `contributors_teams`.
- S4** Denní vývoj počtu PR v daném stavu v časovém rozsahu. Nejsložitější dotaz využívá delta-event přístup: nejprve sestaví intervaly otevřenosti a aktivity štítku a vypočítá jejich průnik (operátory `&&` a `*`). Intervaly jsou následně rozloženy na delta-události (+1/-1) a nad nimi je spočítán kumulativní součet. Výsledná složitost je $O(\text{events} + \text{days})$ místo naivního range-joinu.

8.2.2 Vliv indexů

Pro optimalizaci byly vytvořeny částečné a pokrývající indexy. Úplné definice indexů jsou uvedeny v Příloze C. Stručně:

- `idx_pr_event_hist_state_changes` – částečný index na `issue_event_history` s podmínkou `WHERE is_pr = true AND event IN ('closed', 'merged', 'reopened')` a klauzulí `INCLUDE (event)`.
- `idx_pr_labels_repo_label_issue_ts` – pokrývající index na `issue_labels_history` se sloupci `(repository, label, issue, timestamp DESC) INCLUDE (action)`.
- `idx_file_activity_repo_contrib_ts` – pokrývající index na `file_activity` se sloupci `(repository, contributor_id, timestamp DESC) INCLUDE (file_path, issue)`.

Částečné indexy s podmínkou `WHERE is_pr = true` redukovaly velikost indexu přibližně na polovinu, protože tabulka obsahuje záznamy jak pro issues, tak pro PR. Pokrývající indexy s klauzulí `INCLUDE` umožnily Index-Only Scan bez přístupu k datovým stránkám (`Heap Fetches: 0`).

8.2.3 Výsledky měření

Výsledky měření jsou shrnuty v Tabulce 8.2, plné výstupy `EXPLAIN (ANALYZE, BUFFERS)` jsou k dispozici v příloze C.3.

Nejvýraznější přínos měly indexy u S1 a S2. Bez indexů musel plánovač provést `Seq Scan` celé tabulky a výsledky seřadit na disku (`external merge Disk`) (viz Výpis C.11), (viz Výpis C.13).

Se zavedením pokrývajících indexů přešel plán na `Index Only Scan` (viz Výpis C.12), (viz Výpis C.14) a třídění na disku zcela odpadlo. Výsledkem je $3,7\times$, resp. $7,0\times$ kratší čas vykonávání.

U S3 byl `Parallel Seq Scan` přes celou tabulku `file_activity` nahrazen `Nested Loop` (viz Výpis C.15), (viz Výpis C.16). Ten pro každého člena týmu načítá záznamy přímo z pokrývajícího indexu, tudíž odpadá přístup k datovým stránkám (`Heap Fetches: 0`). Zrychlení je v porovnání s S1 a S2 menší ($1,2\times$), protože agregace a třídění výsledků tvoří větší podíl celkového času.

S4 zaznamenal zlepšení $1,5\times$ (viz Výpis C.17), (viz Výpis C.18). Hlavní náklady tvoří `Merge Join` při výpočtu průniku intervalů (`in_state_periods`), který indexovat nelze. Přesto indexy přinesly úsporu: sekvenční průchody oběma tabulkami byly nahrazeny `Index Only Scan` a `Bitmap Heap Scan` a řazení pro okenní funkce přešlo z external merge na quicksort v paměti.

Tabulka 8.2: Srovnání výkonu analytických dotazů před a po indexaci (průměr z 5 opakování)

ID	Popis dotazu	Cena (rel. jednotky)		Doba běhu (ms)		Zrychl.
		bez idx	s idx	bez idx	s idx	
S1	Počet PR ve stavu (label)	22 258	5424	74	20	$3,7\times$
S2	Počet sloučených PR	9598	852	19	3	$7,0\times$
S3	Soubory upravené týmem	16 770	2653	51	41	$1,2\times$
S4	Vývoj počtu PR v čase	24 958	15 120	111	75	$1,5\times$

Kapitola 9

Závěr

Cílem této práce bylo vytvořit nástroj pro sledování aktivity rozsáhlých GitHub repozitářů, konkrétně repozitáře překladače jazyka Rust (`rust-lang/rust`). Motivací byl fakt, že běžně dostupné nástroje nedokážou pokrýt potřeby masivních open source projektů, jejichž vývojový cyklus je řízen složitou organizační strukturou a stavovými automaty nad štítky.

V teoretické části byl nastudován datový model GitHub repozitářů a dostupná API, od REST přes GraphQL až po Webhooky. Tato rešerše poskytla základ pro návrh systému.

Na základě získaných znalostí byla navržena struktura relační databáze pro agregaci metadat repozitáře z různých zdrojů: GitHub API, lokální Git historie a konfiguračních souborů týmů. Výsledný systém byl implementován v jazyce Rust, který umožňuje jednoduché znovuvyužití jako knihovny v rámci jiných nástrojů.

Nad navrženou databází byla implementována sada analytických dotazů pro získání užitečných informací o repozitáři. Dotazy umožňují zjistit například to, jak dlouho čekají PR na code review, kteří přispěvatelé pracují v konkrétních oblastech kódu nebo jaký je vývoj počtu otevřených PR v čase. Výkonnostní evaluace prokázala, že s vhodně navrženými databázovými indexy lze tyto dotazy provádět efektivně i nad stovkami tisíc záznamů.

Součástí implementace je rovněž interaktivní webové rozhraní, které výsledky analytických dotazů vizualizuje formou přehledných grafů a tabulek.

Nástroj byl otestován na datech repozitáře překladače Rustu, což je netriviální projekt s více než 200 000 issues a PR. Praktické vyhodnocení ukázalo, že nástroj dokáže identifikovat reálné bottlenecky ve schvalovacím procesu, například odlišit čas čekání na autora od čekání na recenzenta, což není pomocí standardního rozhraní GitHub možné. Všechny body zadání tak byly splněny.

Budoucí rozvoj

Nabízí se několik směrů pro budoucí rozšíření nástroje. V první řadě by bylo možné doplnit další typy analytických dotazů, například analýzu průměrné doby od otevření PR po jeho sloučení v závislosti

na týmu nebo komponentě, identifikaci souborů nejčastěji stojících za konflikty při slučování nebo sledování dlouhodobých trendů v aktivitě jednotlivých kontributorů. Druhým přirozeným rozšířením je obohacení sledovaných dat: v současnosti databáze neobsahuje například komentáře k PR, výsledky CI nebo závislosti mezi issues. Zahrnutí těchto informací by umožnilo hlubší analýzy a otevřelo cestu k prediktivním modelům, například k odhadování rizika zanesení chyby při sloučení konkrétního PR.

Dalším prostorem pro optimalizaci je proces inkrementální synchronizace PR a issues (viz Podsekce 6.2.5). Vzhledem k omezením aktuálně využíteho REST API, které při dotazování na issues vrací sdílený seznam obou entit, dochází k nadbytečnému stahování a následnému zahazování dat. Přejchod na GraphQL API pro tuto specifickou část synchronizace by umožnil striktně oddělené dotazování obou datových typů, čímž by se eliminovala nutnost klientské filtrace a výrazně zefektivnilo stránkování.

Literatura

1. ERNST, Michael D. *Version control concepts and best practices* [online]. University of Washington, 2024-04-08. [cit. 2026-03-07]. Dostupné z: <https://homes.cs.washington.edu/~mernst/advice/version-control.html>. [Online; přístup 7. března 2026].
2. SINK, Eric. *Version Control by Example* [online]. Pyrenean Gold Press, 2011 [cit. 2026-03-07]. ISBN 978-0983507901. Dostupné z: <https://ericsink.com/vcbe/index.html>. [Online verze; přístup 7. března 2026].
3. COLLINS-SUSSMAN, Ben; FITZPATRICK, Brian W.; PILATO, C. Michael. *Version Control with Subversion: For Subversion 1.7* [online]. 2011. [cit. 2026-03-07]. Dostupné z: <https://svnbook.red-bean.com/en/1.7/svn-book.pdf>. [Online verze knihy; přístup 7. března 2026].
4. CHACON, Scott; STRAUB, Ben. *Pro Git* [online]. 2. vyd. Apress, 2014 [cit. 2026-03-07]. ISBN 978-1484200773. Dostupné z: <https://git-scm.com/book/en/v2>. [Online verze; přístup 7. března 2026].
5. POLICY SIMULATION LIBRARY. *Git and GitHub Use, Collaboration, and Workflow: Introduction* [online]. Policy Simulation Library, 2023. [cit. 2026-03-09]. Dostupné z: <https://pslmodels.github.io/Git-Tutorial/content/intro.html>. [Online; přístup 9. března 2026].
6. GITHUB, INC. *About repositories* [online]. GitHub, Inc., 2026. [cit. 2026-03-12]. Dostupné z: <https://docs.github.com/en/repositories/creating-and-managing-repositories/about-repositories>. [Online; přístup 12. března 2026].
7. GITHUB, INC. *About pull requests* [online]. GitHub, Inc., 2026. [cit. 2026-03-12]. Dostupné z: <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests>. [Online; přístup 12. března 2026].
8. GITHUB, INC. *About issues* [online]. GitHub, Inc., 2026. [cit. 2026-03-12]. Dostupné z: <https://docs.github.com/en/issues/tracking-your-work-with-issues/learning-about-issues/about-issues>. [Online; přístup 12. března 2026].

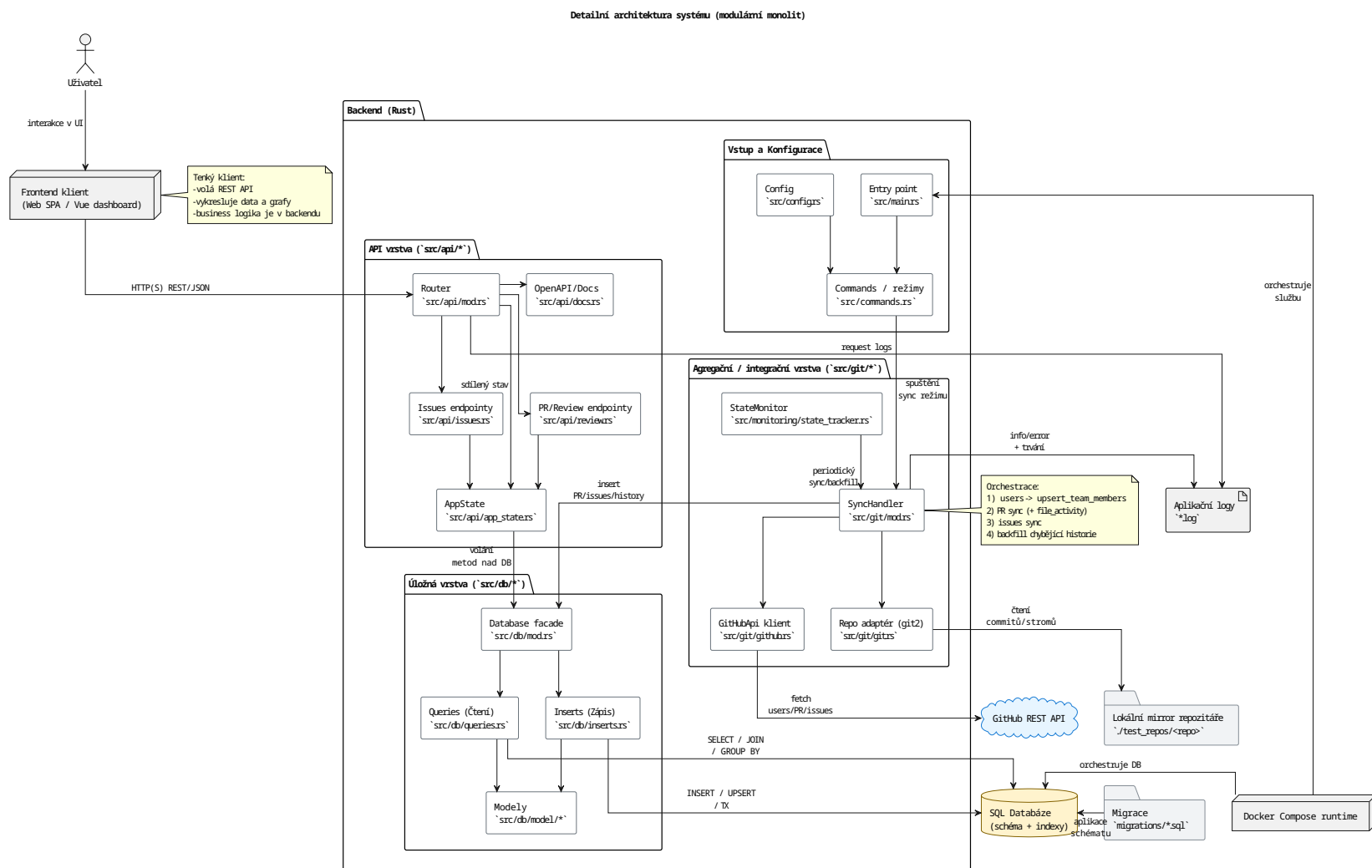
9. GITHUB, INC. *Understanding GitHub Actions* [online]. GitHub, Inc., 2026. [cit. 2026-03-12]. Dostupné z: <https://docs.github.com/en/actions/about-github-actions/understanding-github-actions>. [Online; přístup 12. března 2026].
10. GITHUB, INC. *Getting started with the REST API* [online]. GitHub, Inc., 2026. [cit. 2026-03-09]. Dostupné z: <https://docs.github.com/en/rest/using-the-rest-api/getting-started-with-the-rest-api>. [Online; přístup 9. března 2026].
11. GITHUB, INC. *API Versions* [online]. GitHub, Inc., 2026. [cit. 2026-03-09]. Dostupné z: <https://docs.github.com/en/rest/about-the-rest-api/api-versions>. [Online; přístup 9. března 2026].
12. GITHUB, INC. *Forming calls with GraphQL* [online]. GitHub, Inc., 2026. [cit. 2026-03-09]. Dostupné z: <https://docs.github.com/en/graphql/guides/forming-calls-with-graphql>. [Online; přístup 9. března 2026].
13. GITHUB, INC. *Rate limits for the REST API* [online]. GitHub, Inc., 2026. [cit. 2026-04-13]. Dostupné z: <https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api>. [Online; přístup 13. dubna 2026].
14. GITHUB, INC. *Rate limits and query limits for the GraphQL API* [online]. GitHub, Inc., 2026. [cit. 2026-04-13]. Dostupné z: <https://docs.github.com/en/graphql/overview/rate-limits-and-query-limits-for-the-graphql-api>. [Online; přístup 13. dubna 2026].
15. GITHUB, INC. *About webhooks* [online]. GitHub, Inc., 2026. [cit. 2026-03-19]. Dostupné z: <https://docs.github.com/en/webhooks/about-webhooks>. [Online; přístup 19. března 2026].
16. STENBERG, Daniel. *curl — command line tool and library for transferring data with URLs* [online]. curl, 2026. [cit. 2026-04-26]. Dostupné z: <https://curl.se/>. [Online; přístup 26. dubna 2026].
17. RUST FOUNDATION. *The Rust Foundation* [online]. Rust Foundation, 2026. [cit. 2026-04-26]. Dostupné z: <https://rustfoundation.org/>.
18. RUST PROJECT DEVELOPERS. *Rust Programming Language* [online]. Rust Foundation, 2026. [cit. 2026-04-26]. Dostupné z: <https://www.rust-lang.org/>.
19. RUST TEAM. *Rust Team Documentation* [online]. GitHub, 2026. [cit. 2026-03-22]. Dostupné z: <https://github.com/rust-lang/team/blob/main/docs/>. Repozitář rust-lang/team.
20. THE RUST PROJECT DEVELOPERS. *Issue triaging* [online]. Rust Forge, 2026. [cit. 2026-04-26]. Dostupné z: <https://forge.rust-lang.org/release/issue-triaging.html>. [Online; přístup 26. dubna 2026].

21. THE RUST INFRASTRUCTURE TEAM. *rust-lang/bors: Rust implementation of bors used for various Rust components* [online]. GitHub, Inc., 2026. [cit. 2026-04-28]. Dostupné z: <https://github.com/rust-lang/bors>. [Online; přístup 28. dubna 2026].
22. THE RUST INFRASTRUCTURE TEAM. *Bors — Rust Forge* [online]. The Rust Project, 2026. [cit. 2026-04-28]. Dostupné z: <https://forge.rust-lang.org/infra/docs/bors.html>. [Online; přístup 28. dubna 2026].
23. THE RUST PROJECT DEVELOPERS. *Contributing to Rust* [online]. The Rust Project. [cit. 2026-04-03]. Dostupné z: <https://rustc-dev-guide.rust-lang.org/contributing.html>.
24. SQLITE CONSORTIUM. *About SQLite* [online]. SQLite Consortium, 2025. [cit. 2026-04-11]. Dostupné z: <https://www.sqlite.org/about.html>. [Online; přístup 11. dubna 2026].
25. SQLITE CONSORTIUM. *Appropriate Uses For SQLite* [online]. SQLite Consortium, 2025. [cit. 2026-04-11]. Dostupné z: <https://www.sqlite.org/whentouse.html>. [Online; přístup 11. dubna 2026].
26. SQLITE CONSORTIUM. *Write-Ahead Logging* [online]. SQLite Consortium, 2025. [cit. 2026-04-12]. Dostupné z: <https://www.sqlite.org/wal.html>. [Online; přístup 12. dubna 2026].
27. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *About PostgreSQL* [online]. The PostgreSQL Global Development Group, 2025. [cit. 2026-04-11]. Dostupné z: <https://www.postgresql.org/about/>. [Online; přístup 11. dubna 2026].
28. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *SELECT — klauzule DISTINCT ON* [online]. The PostgreSQL Global Development Group, 2025. [cit. 2026-04-12]. Dostupné z: <https://www.postgresql.org/docs/current/sql-select.html>. [Online; přístup 12. dubna 2026].
29. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *Window Functions* [online]. The PostgreSQL Global Development Group, 2025. [cit. 2026-04-12]. Dostupné z: <https://www.postgresql.org/docs/current/tutorial-window.html>. [Online; přístup 12. dubna 2026].
30. KLABNIK, Steve; NICHOLS, Carol. *The Rust Programming Language*. 1. vyd. No Starch Press, 2018. ISBN 978-1593278281.
31. ALEX CRICHTON AND CONTRIBUTORS. *git2 — Struct Diff* [online]. docs.rs, 2024. [cit. 2026-04-28]. Dostupné z: <https://docs.rs/git2/latest/git2/struct.Diff.html>. [Online; přístup 28. dubna 2026].

Příloha A

Detailní architektura systému

Detailní architektura systému je uvedena na Obrázku A.1.



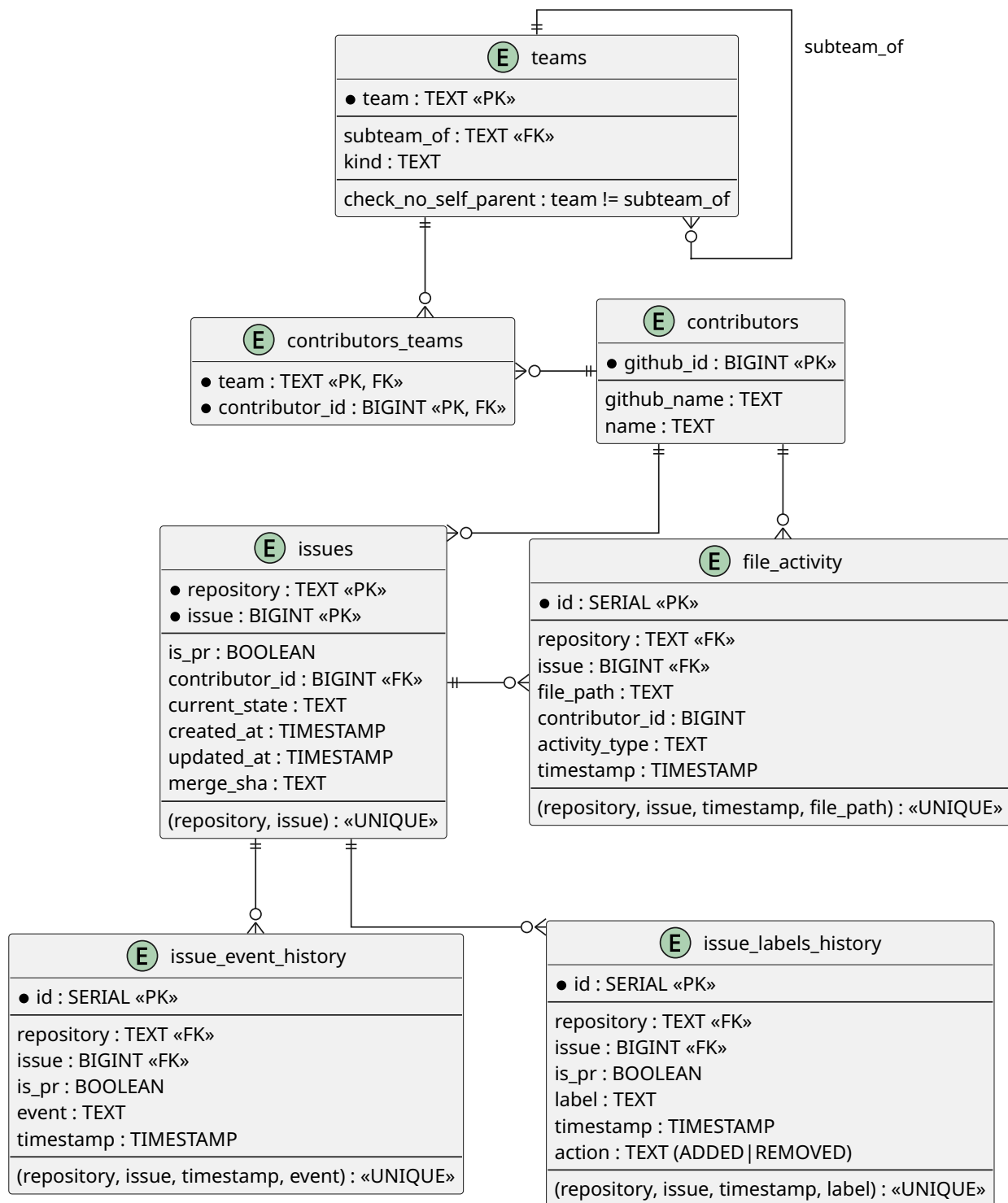
Obrázek A.1: Detailní architektura systému

Příloha B

Podrobný ER diagram databáze

Podrobný ER diagram struktury databáze je uveden na Obrázku B.1.

ER diagram databáze



Obrázek B.1: Podrobný ER diagram databáze

Příloha C

Analytické SQL dotazy a indexy

Tato příloha obsahuje zdrojový kód analytických SQL dotazů použitých při evaluaci v Sekci 8.2 a definice databázových indexů pro jednotlivé tabulky.

C.1 Analytické dotazy

S1 — Počet PR ve stavu (štítek) k zadanému datu

Dotaz pomocí dvou CTE s `DISTINCT ON` určí, kolik PR je v daném stavu ke zvolenému datu. První CTE `latest_labels` zachová pro každý PR nejnovější labelovou událost pro cílový štítek `S-*`, druhá CTE `latest_state` nejnovější stavovou změnu (uzavření, sloučení nebo znovuootevření). Výsledný počet zahrnuje pouze PR, jejichž poslední labelová událost je `ADDED` a které jsou ke zvolenému datu stále otevřené.

Výpis C.1: S1 — Počet PR ve stavu (štítek) k zadanému datu

```
1 WITH latest_labels AS (  
2     -- Pro každý PR zachová nejnovější label-event pro cílový S-* label k T  
3     SELECT DISTINCT ON (issue) issue, action  
4     FROM issue_labels_history  
5     WHERE repository = 'rust-lang/rust'  
6           AND is_pr   = true  
7           AND label    = 'S-waiting-on-review' -- parametr: $3  
8           AND timestamp <= '2026-01-01'      -- parametr: $2  
9     ORDER BY issue, timestamp DESC  
10 ),  
11 latest_state AS (  
12     -- Pro každý PR zachová nejnovější stavovou změnu (closed/merged/reopened) k T  
13     SELECT DISTINCT ON (issue) issue, event  
14     FROM issue_event_history  
15     WHERE repository = 'rust-lang/rust' -- parametr: $1  
16           AND is_pr   = true  
17           AND event    IN ('closed', 'merged', 'reopened')  
18           AND timestamp <= '2026-01-01' -- parametr: $2
```

```

19     ORDER BY issue, timestamp DESC
20 )
21 SELECT COUNT(*)
22 FROM latest_labels ll
23 LEFT JOIN latest_state ls ON ls.issue = ll.issue
24 WHERE ll.action = 'ADDED'
25     -- PR musí být k T otevřen (žádná zavírací událost, nebo poslední byla 'reopened')
26     AND (ls.issue IS NULL OR ls.event = 'reopened');

```

S2 — Kumulativní počet sloučených PR k zadanému datu

Na rozdíl od S1 nevyžaduje tento dotaz sledování historických přechodů stavů. Stav „merged“ je trvalý — jakmile je PR sloučen, nelze jej vrátit do předchozího stavu. Stačí proto spočítat počet unikátních PR, u nichž existuje záznam události `merged` v tabulce `issue_event_history` s časovým razítkem před zadaným datem. Výsledkem je `COUNT(DISTINCT issue)` s indexovatelným filtrem `event = 'merged'` a časovým řezem.

Výpis C.2: S2 — Kumulativní počet sloučených PR k zadanému datu

```

1 SELECT COUNT(DISTINCT issue)
2 FROM issue_event_history
3 WHERE repository = 'rust-lang/rust'    -- parametr: $1
4     AND is_pr    = true
5     AND event    = 'merged'
6     AND timestamp <= '2026-01-01';    -- parametr: $2

```

S3 — Soubory upravené přispěvateli týmu v zadaném období

Dotaz agreguje soubory upravené přispěvateli daného týmu (`contributors_teams`) v zadaném časovém rozsahu. Výsledek je seřazen sestupně podle počtu úprav.

Výpis C.3: S3 — Soubory upravené přispěvateli týmu

```

1 SELECT fa.file_path, count(*) AS editions
2 FROM file_activity fa
3 JOIN contributors_teams ct
4     ON ct.contributor_id = fa.contributor_id
5     AND ct.team          = 'compiler'    -- parametr: $4
6 WHERE fa.repository      = 'rust-lang/rust'    -- parametr: $1
7     AND fa.timestamp BETWEEN '2025-06-01'    -- parametr: $2
8     AND '2026-01-01'    -- parametr: $3
9 GROUP BY fa.file_path
10 ORDER BY editions DESC;

```

S4 — Denní vývoj počtu PR v daném stavu v časovém rozsahu

Nejsložitější z testovaných dotazů využívá delta-event přístup. CTE `all_transitions` sestaví pomocí `UNION ALL` seznam všech relevantních událostí (vznik PR, uzavření, sloučení, znovuotevření). CTE `ordered_transitions` přiřadí každé události čas následující události přes okenní funkci `LEAD()`, čímž vzniknou neprotínající se intervaly otevřenosti (`open_periods`). Stejným způsobem jsou konstruovány intervaly aktivního štítku (`label_active_periods`). Průnik obou typů intervalů operátorem `&&` a `*` dá intervaly, kdy je PR v cílovém stavu (`in_state_periods`). Tyto intervaly jsou rozloženy na delta-události (+1 na začátku, -1 na konci), agregovány po dnech a pak je nad nimi spočítán kumulativní součet. Složitost je $O(\text{events} + \text{days})$ místo naivního $O(\text{days} \times \text{events})$.

Výpis C.4: S4 — Denní vývoj počtu PR v daném stavu

```
1  explain (ANALYZE, BUFFERS)
2  WITH all_transitions AS (SELECT issue, timestamp, event AS event_type
3                           FROM issue_event_history
4                           WHERE repository = 'rust-lang/rust' -- parametr: $1
5                                AND is_pr = true
6                                AND event IN ('created', 'closed', 'merged', 'reopened')),
7     ordered_transitions AS (SELECT issue,
8                                timestamp,
9                                event_type,
10                               -- LEAD() vrátí timestamp následující události toho PR = konec
11                               -- intervalu
12                               LEAD(timestamp) OVER (PARTITION BY issue ORDER BY timestamp) AS
13                               -- next_ts
14                               next_ts
15                               FROM all_transitions),
16     open_periods AS (SELECT issue,
17                           -- Interval otevřenosti PR: [vznik/znovuotevření, následující událost)
18                           tsrange(timestamp, COALESCE(next_ts, 'infinity'::timestamp),
19                                   '[]') AS period
20                           FROM ordered_transitions
21                           WHERE event_type IN ('created', 'reopened')),
22     label_transitions AS (SELECT issue,
23                              timestamp,
24                              action,
25                              LEAD(timestamp) OVER (PARTITION BY issue ORDER BY timestamp) AS
26                              -- next_ts
27                              next_ts
28                              FROM issue_labels_history
29                              WHERE repository = 'rust-lang/rust' -- parametr: $1
30                                   AND is_pr = true
31                                   AND label = 'S-waiting-on-review'), -- parametr: $4
32     -- Interval aktivity labelu: [ADDED, následující label-event)
33     label_active_periods AS (SELECT issue,
34                              tsrange(timestamp, COALESCE(next_ts, 'infinity'::timestamp),
35                                      '[]') AS period
36                              FROM label_transitions
37                              WHERE action = 'ADDED'),
38     in_state_periods AS (SELECT lap.issue,
39                           -- Průnik intervalu otevřenosti a aktivity labelu = skutečný
40                           -- interval "ve stavu"
41                           lap.period * op.period AS valid_period
```

```

36         FROM label_active_periods lap
37         JOIN open_periods op
38         ON lap.issue = op.issue
39         AND lap.period && op.period -- && = neprázdný průnik
40         WHERE NOT isempty(lap.period * op.period)),
41 -- Každý interval převeden na delta-events: +1 při vstupu do stavu, -1 při výstupu.
42 -- Protože LEAD() generuje nepřekrývající se intervaly na PR, běžící SUM
43 -- je ekvivalentní COUNT(DISTINCT issue) v libovolném časovém bodě.
44     period_deltas AS (SELECT lower(valid_period)::date AS event_date, 1 AS delta
45         FROM in_state_periods
46         UNION ALL
47         SELECT upper(valid_period)::date AS event_date, -1 AS delta
48         FROM in_state_periods
49         WHERE NOT upper_inf(valid_period)),
50     daily_deltas AS (SELECT event_date, SUM(delta) AS daily_change
51         FROM period_deltas
52         GROUP BY event_date),
53 -- Kumulativní hodnota před začátkem sledovaného rozsahu (základ pro běžící součet)
54     base AS (SELECT COALESCE(SUM(daily_change), 0) AS cnt
55         FROM daily_deltas
56         WHERE event_date < '2025-06-01'::date),          -- parametr: $2
57     date_series AS (SELECT d::date AS day
58         FROM generate_series('2025-12-01'::timestamp, -- parametr: $2
59             '2026-01-01'::timestamp, -- parametr: $3
60             '1 day'::interval) d)
61 SELECT ds.day                                     AS date,
62     ((SELECT cnt FROM base)
63      + COALESCE(SUM(dd.daily_change) OVER (ORDER BY ds.day), 0))::bigint AS count
64 FROM date_series ds
65     LEFT JOIN daily_deltas dd ON dd.event_date = ds.day
66 ORDER BY ds.day

```

C.2 Databázové indexy

Níže jsou uvedeny definice všech databázových indexů. Indexy jsou rozděleny podle tabulky, ke které náleží.

Tabulka `issues`

Výpis C.5: Indexy tabulky `issues`

```

1  -- DASHBOARD: "Show me Open ISSUES for this repo, sorted by date"
2  CREATE INDEX idx_issues_dashboard_state
3      ON issues (repository, current_state, edited_at DESC)
4      WHERE is_pr = false;
5
6  -- DASHBOARD: "Show me Open PRS for this repo, sorted by date"
7  CREATE INDEX idx_prs_dashboard_state
8      ON issues (repository, current_state, edited_at DESC)
9      WHERE is_pr = true;
10

```

```

11 -- LOOKUP: "Find specific ISSUE by ID"
12 CREATE INDEX idx_issues_lookup
13     ON issues (repository, issue)
14     WHERE is_pr = false;
15
16 -- LOOKUP: "Find specific PR by ID"
17 CREATE INDEX idx_prs_lookup
18     ON issues (repository, issue)
19     WHERE is_pr = true;
20
21 -- "Show me all ISSUES created by this user"
22 CREATE INDEX idx_issues_contributor_only
23     ON issues (contributor_id)
24     WHERE is_pr = false;
25
26 -- "Show me all PRS created by this user"
27 CREATE INDEX idx_prs_contributor_only
28     ON issues (contributor_id)
29     WHERE is_pr = true;
30
31 -- Oldest-PR / date-range queries
32 CREATE INDEX idx_issues_created_at
33     ON issues (repository, created_at)
34     WHERE is_pr = true;
35

```

Tabulka issue_event_history

Výpis C.6: Indexy tabulky issue_event_history

```

1 -- HISTORY: "Show timeline for a specific ISSUE"
2 CREATE INDEX idx_issues_history_lookup
3     ON issue_event_history (repository, issue, timestamp DESC)
4     WHERE is_pr = false;
5
6 -- HISTORY: "Show timeline for a specific PR"
7 CREATE INDEX idx_prs_history_lookup
8     ON issue_event_history (repository, issue, timestamp DESC)
9     WHERE is_pr = true;
10
11 -- index for queries like "Find all PR events of type 'closed' in a repository"
12 CREATE INDEX idx_issue_event_history_repo_evt_pr_only
13     ON issue_event_history (repository, event)
14     WHERE is_pr = true;
15
16 -- PR state-change events (closed/merged/reopened)
17 -- Supports DISTINCT ON (issue) ORDER BY issue, timestamp DESC
18 -- and LEAD() OVER (PARTITION BY issue ORDER BY timestamp)
19 -- INCLUDE (event) enables Index-Only Scans.
20 CREATE INDEX idx_pr_event_hist_state_changes
21     ON issue_event_history (repository, issue, timestamp DESC)
22     INCLUDE (event)
23     WHERE is_pr = true AND event IN ('closed', 'merged', 'reopened');

```

```

24
25 -- Merged-only events for cumulative merge counts & MIN(timestamp)
26 CREATE INDEX idx_pr_event_hist_merged
27     ON issue_event_history (repository, issue, timestamp)
28     WHERE is_pr = true AND event = 'merged';
29
30 -- MAX(timestamp) per repository - Index Scan Backward + LIMIT 1
31 CREATE INDEX idx_event_hist_repo_ts
32     ON issue_event_history (repository, timestamp DESC);
33

```

Tabulka `issue_labels_history`

Výpis C.7: Indexy tabulky `issue_labels_history`

```

1  -- LOOKUP: "Show labels for a specific ISSUE"
2  CREATE INDEX idx_issue_labels_lookup
3      ON issue_labels_history (repository, issue)
4      WHERE is_pr = false;
5
6  -- LOOKUP: "Show labels for a specific PR"
7  CREATE INDEX idx_pr_labels_lookup
8      ON issue_labels_history (repository, issue)
9      WHERE is_pr = true;
10
11 -- SEARCH: "Find all ISSUES with label 'bug'"
12 CREATE INDEX idx_issue_labels_name
13     ON issue_labels_history (repository, label)
14     WHERE is_pr = false;
15
16 -- SEARCH: "Find all PRS with label 'bug'"
17 CREATE INDEX idx_pr_labels_name
18     ON issue_labels_history (repository, label)
19     WHERE is_pr = true;
20
21 -- UPSERT OPTIMIZATION (Finding the latest label state) - split for faster writes/reads
22 CREATE INDEX idx_issue_labels_upsert_latest
23     ON issue_labels_history (repository, issue, label, timestamp DESC, action)
24     WHERE is_pr = false;
25
26 CREATE INDEX idx_pr_labels_upsert_latest
27     ON issue_labels_history (repository, issue, label, timestamp DESC, action)
28     WHERE is_pr = true;
29
30 -- PR label state queries (S-waiting-on-review, etc.)
31 -- Supports DISTINCT ON (issue), LEAD window, label IN filters
32 -- INCLUDE (action) enables Index-Only Scans.
33 CREATE INDEX idx_pr_labels_repo_label_issue_ts
34     ON issue_labels_history (repository, label, issue, timestamp DESC)
35     INCLUDE (action)
36     WHERE is_pr = true;
37

```

Tabulka `file_activity`

Výpis C.8: Indexy tabulky `file_activity`

```
1  -- FILES: "Show changed files for this PR"
2  CREATE INDEX idx_pr_file_activity_lookup
3      ON file_activity (repository, issue);
4
5  -- FILE HISTORY: "Show history of 'src/main.rs'"
6  CREATE INDEX idx_pr_file_activity_path
7      ON file_activity (repository, file_path varchar_pattern_ops, timestamp DESC);
8
9  -- CONTRIBUTOR: "Show files touched by this user"
10 CREATE INDEX idx_pr_file_activity_contributor
11     ON file_activity (contributor_id);
12
13 -- Contributor-based file activity queries
14 -- INCLUDE (file_path, issue) enables Index-Only Scans.
15 CREATE INDEX idx_file_activity_repo_contrib_ts
16     ON file_activity (repository, contributor_id, timestamp DESC)
17     INCLUDE (file_path, issue);
18
```

Tabulka `contributors`

Výpis C.9: Indexy tabulky `contributors`

```
1  -- USER LOOKUP: Users exist independently of issues/PRs
2  CREATE INDEX idx_contributors_name ON contributors (github_name);
3
```

Tabulka `contributors_teams`

Výpis C.10: Indexy tabulky `contributors_teams`

```
1  -- TEAM MEMBERSHIP: lookup by team
2  CREATE INDEX idx_contributors_teams_team ON contributors_teams (team);
3
```

C.3 Výstupy EXPLAIN ANALYZE

Níže jsou uvedeny výstupy příkazu `EXPLAIN (ANALYZE, BUFFERS)` pro každý testovaný dotaz, vždy bez indexů a poté se zapnutými indexy. Měření proběhlo na databázi repozitáře `rust-lang/rust` s při-

bližně 200 000 PR, 564 000 záznamy v `issue_event_history`, 355 000 v `issue_labels_history` a 525 000 v `file_activity`.

S1 — bez indexů

Bez indexů provádí plánovač dva sekvenční průchody (`Seq Scan`) nad celými tabulkami `issue_labels_history` a `issue_event_history`. Výsledky jsou následně řazeny na disku operací `Sort (external merge Disk)`, protože data se nevejdou do operační paměti. Odhadované náklady dosahují 22 258 při celkové době vykonávání 73,5 ms.

Výpis C.11: S1: EXPLAIN ANALYZE bez indexů (Execution Time: 73,5 ms)

```
1  Aggregate  (cost=22257.88..22257.89 rows=1 width=8) (actual time=72.484..72.996 rows=1 loops=1)
2    " Buffers: shared hit=6491 read=449, temp read=175 written=176"
3    -> Merge Right Join  (cost=18293.92..22257.88 rows=1 width=0) (actual time=60.980..72.973
4      ↳ rows=281 loops=1)
5      Merge Cond: (issue_event_history.issue = ll.issue)
6      Filter: ((issue_event_history.issue IS NULL) OR (issue_event_history.event =
7        ↳ 'reopened'::text))
8      Rows Removed by Filter: 1050
9      " Buffers: shared hit=6491 read=449, temp read=175 written=176"
10     -> Unique  (cost=9505.97..12889.50 rows=14233 width=25) (actual time=22.171..28.224
11       ↳ rows=14978 loops=1)
12       Buffers: shared hit=4014 read=449
13       -> Gather Merge  (cost=9505.97..12818.39 rows=28441 width=25) (actual
14         time=22.170..26.285 rows=27640 loops=1)
15         Workers Planned: 2
16         Workers Launched: 2
17         Buffers: shared hit=4014 read=449
18         -> Sort  (cost=8505.95..8535.57 rows=11850 width=25) (actual
19           time=18.947..19.633 rows=9213 loops=3)
20           ↳ time=18.947..19.633 rows=9213 loops=3)
21           Sort Key: issue_event_history.issue, issue_event_history."timestamp"
22           ↳ DESC"
23           Sort Method: quicksort  Memory: 1232kB
24           Buffers: shared hit=4014 read=449
25           Worker 0:  Sort Method: quicksort  Memory: 1066kB
26           Worker 1:  Sort Method: quicksort  Memory: 823kB
27           -> Parallel Seq Scan on issue_event_history  (cost=0.00..7704.14
28             ↳ rows=11850 width=25) (actual time=0.446..16.220 rows=9213 loops=3)
29             Filter: (is_pr AND ("timestamp" <= '2026-01-01
30               ↳ 00:00:00'::timestamp without time zone) AND (repository = 'rust-lang/rust'::text) AND (event =
31               ↳ ANY ('{closed,merged,reopened}'::text[])))"
32             Rows Removed by Filter: 132915
33             Buffers: shared hit=3924 read=449
34             -> Materialize  (cost=8787.95..9189.36 rows=90 width=8) (actual time=36.422..43.717
35               ↳ rows=1331 loops=1)
36               Buffers: shared hit=2477, temp read=175 written=176"
37               -> Subquery Scan on ll  (cost=8787.95..9189.14 rows=90 width=8) (actual
38                 time=36.407..43.567 rows=1331 loops=1)
39                 Filter: (ll.action = 'ADDED'::text)
40                 Rows Removed by Filter: 14145
41                 Buffers: shared hit=2477, temp read=175 written=176"
```

```

30         -> Unique (cost=8787.95..8963.62 rows=18041 width=22) (actual
        ↪ time=36.402..42.635 rows=15476 loops=1)
31     "
        Buffers: shared hit=2477, temp read=175 written=176"
32     -> Sort (cost=8787.95..8875.79 rows=35135 width=22) (actual
        ↪ time=36.401..39.739 rows=42010 loops=1)
33     "
        Sort Key: issue_labels_history.issue,
        ↪ issue_labels_history."timestamp" DESC"
34         Sort Method: external merge Disk: 1400kB
35     "
        Buffers: shared hit=2477, temp read=175 written=176"
36     -> Seq Scan on issue_labels_history (cost=0.00..6135.15
        ↪ rows=35135 width=22) (actual time=0.009..19.753 rows=42010
        ↪ loops=1)
37     "
        Filter: (is_pr AND ("timestamp" <= '2026-01-01
        ↪ 00:00:00'::timestamp without time zone) AND (repository = 'rust-lang/rust'::text) AND (label =
        ↪ 'S-waiting-on-review'::text))"
38         Rows Removed by Filter: 167027
39         Buffers: shared hit=2477
40 Planning Time: 0.314 ms
41 Execution Time: 73.506 ms

```

S1 — s indexy

Hlavní změny oproti variantě bez indexů: oba `Seq Scan` + `Sort (external merge Disk)` nahrazeny dvěma `Index Only Scan`, řazení probíhá přímo po indexu.

Výpis C.12: S1: EXPLAIN ANALYZE s indexy (Execution Time: 20,0 ms)

```

1  Aggregate (cost=5424.47..5424.48 rows=1 width=8) (actual time=19.867..19.869 rows=1 loops=1)
2    Buffers: shared hit=2480
3    -> Merge Right Join (cost=0.83..5424.47 rows=1 width=0) (actual time=3.603..19.845 rows=281
        ↪ loops=1)
4        Merge Cond: (issue_event_history.issue = ll.issue)
5        Filter: ((issue_event_history.issue IS NULL) OR (issue_event_history.event =
        ↪ 'reopened'::text))
6        Rows Removed by Filter: 1050
7        Buffers: shared hit=2480
8    -> Result (cost=0.41..1926.41 rows=14233 width=25) (actual time=0.045..6.680 rows=14978
        ↪ loops=1)
9        Buffers: shared hit=624
10       -> Unique (cost=0.41..1926.41 rows=14233 width=25) (actual time=0.043..5.456
        ↪ rows=14978 loops=1)
11           Buffers: shared hit=624
12       -> Index Only Scan using idx_pr_event_hist_state_changes on
        ↪ issue_event_history (cost=0.41..1855.31 rows=28441 width=25) (actual
        ↪ time=0.042..3.226 rows=27640 loops=1)
13       "
        Index Cond: ((repository = 'rust-lang/rust'::text) AND ("timestamp" <=
        ↪ '2026-01-01 00:00:00'::timestamp without time zone))"
14           Heap Fetches: 766
15           Buffers: shared hit=624
16       -> Materialize (cost=0.42..3319.03 rows=90 width=8) (actual time=0.052..12.336 rows=1331
        ↪ loops=1)
17           Buffers: shared hit=1856

```

```

18      -> Subquery Scan on ll (cost=0.42..3318.81 rows=90 width=8) (actual
    ↪ time=0.035..12.136 rows=1331 loops=1)
19          Filter: (ll.action = 'ADDED'::text)
20          Rows Removed by Filter: 14145
21          Buffers: shared hit=1856
22      -> Result (cost=0.42..3093.30 rows=18041 width=22) (actual time=0.031..11.472
    ↪ rows=15476 loops=1)
23          Buffers: shared hit=1856
24      -> Unique (cost=0.42..3093.30 rows=18041 width=22) (actual
    ↪ time=0.031..10.227 rows=15476 loops=1)
25          Buffers: shared hit=1856
26      -> Index Only Scan using idx_pr_labels_repo_label_issue_ts on
    ↪ issue_labels_history (cost=0.42..3005.46 rows=35135 width=22)
    ↪ (actual time=0.030..6.995 rows=42010 loops=1)
27      "          Index Cond: ((repository = 'rust-lang/rust'::text) AND
    ↪ (label = 'S-waiting-on-review'::text) AND ("timestamp" <= '2026-01-01 00:00:00'::timestamp
    ↪ without time zone))"
28          Heap Fetches: 3749
29          Buffers: shared hit=1856
30      Planning Time: 0.324 ms
31      Execution Time: 19.974 ms

```

S2 — bez indexů

Bez indexů plánovač zvolí **Parallel Seq Scan** přes celou tabulku **issue_event_history** a záznamy filtruje průběžně. Paralelizace sice snižuje elapsed time, avšak všechny záznamy jsou čteny bez ohledu na hodnotu sloupce **event**. Odhadované náklady jsou 9 598 při době vykonávání 18,8 ms.

Výpis C.13: S2: EXPLAIN ANALYZE bez indexů (Execution Time: 18,8 ms)

```

1  Aggregate (cost=9598.08..9598.09 rows=1 width=8) (actual time=16.731..18.758 rows=1 loops=1)
2    Buffers: shared hit=4020 read=353
3    -> Gather (cost=1000.00..9570.86 rows=10888 width=8) (actual time=1.602..16.672 rows=11826
    ↪ loops=1)
4        Workers Planned: 2
5        Workers Launched: 2
6        Buffers: shared hit=4020 read=353
7    -> Parallel Seq Scan on issue_event_history (cost=0.00..7482.06 rows=4537 width=8)
    ↪ (actual time=0.457..12.078 rows=3942 loops=3)
8    "      Filter: (is_pr AND ("timestamp" <= '2026-01-01 00:00:00'::timestamp without time
    ↪ zone) AND (repository = 'rust-lang/rust'::text) AND (event = 'merged'::text))"
9        Rows Removed by Filter: 138187
10       Buffers: shared hit=4020 read=353
11      Planning Time: 0.176 ms
12      Execution Time: 18.834 ms

```

S2 — s indexy

Parallel Seq Scan nahrazen přímým **Index Only Scan using idx_pr_event_hist_merged**. Paralelní scan zaniká, plán se zjednodušuje na jediný uzel.

Výpis C.14: S2: EXPLAIN ANALYZE s indexy (Execution Time: 2,7 ms)

```
1 Aggregate (cost=851.56..851.57 rows=1 width=8) (actual time=2.668..2.669 rows=1 loops=1)
2   Buffers: shared hit=383
3   -> Index Only Scan using idx_pr_event_hist_merged on issue_event_history (cost=0.29..824.34
4     ↳ rows=10888 width=8) (actual time=0.059..1.577 rows=11826 loops=1)
5     "      Index Cond: ((repository = 'rust-lang/rust'::text) AND ("timestamp" <= '2026-01-01
6     ↳ 00:00:00'::timestamp without time zone))"
7     Heap Fetches: 316
8     Buffers: shared hit=383
9 Planning Time: 0.115 ms
10 Execution Time: 2.700 ms
```

S3 — bez indexů

Bez indexů provede plánovač `Parallel Seq Scan` přes celou tabulku `file_activity` a záznamy nesouvisející s daným týmem jsou čteny a až poté filtrovány. Tabulka `contributors_teams` je procházena sekvenčně a výsledky jsou spojeny operací `Hash Join`. Odhadované náklady jsou 16 770 při době vykonávání 50,6 ms.

Výpis C.15: S3: EXPLAIN ANALYZE bez indexů (Execution Time: 50,6 ms)

```
1 Sort (cost=16769.91..16819.97 rows=20021 width=56) (actual time=46.547..48.687 rows=20892 loops=1)
2   Sort Key: (count(*)) DESC
3   Sort Method: quicksort  Memory: 3246kB
4   Buffers: shared hit=82 read=8288
5   -> Finalize HashAggregate (cost=15139.28..15339.49 rows=20021 width=56) (actual
6     ↳ time=40.064..42.969 rows=20892 loops=1)
7     Group Key: fa.file_path
8     Batches: 1  Memory Usage: 3857kB
9     Buffers: shared hit=82 read=8288
10    -> Gather (cost=13304.04..15055.86 rows=16684 width=56) (actual time=27.680..31.502
11      ↳ rows=29861 loops=1)
12      Workers Planned: 2
13      Workers Launched: 2
14      Buffers: shared hit=82 read=8288
15      -> Partial HashAggregate (cost=12304.04..12387.46 rows=8342 width=56) (actual
16        ↳ time=24.377..25.939 rows=9954 loops=3)
17        Group Key: fa.file_path
18        Batches: 1  Memory Usage: 2449kB
19        Buffers: shared hit=82 read=8288
20        Worker 0:  Batches: 1  Memory Usage: 1681kB
21        Worker 1:  Batches: 1  Memory Usage: 1681kB
22        -> Hash Join (cost=34.59..12262.33 rows=8342 width=48) (actual
23          ↳ time=0.183..19.576 rows=16701 loops=3)
24          Hash Cond: (fa.contributor_id = ct.contributor_id)
25          Buffers: shared hit=82 read=8288
26          -> Parallel Seq Scan on file_activity fa (cost=0.00..12126.85
27            ↳ rows=37952 width=56) (actual time=0.021..16.745 rows=30536 loops=3)
```

```

23 "          Filter: ((""timestamp"" >= '2025-06-01 00:00:00'::timestamp
↳ without time zone) AND ((""timestamp"" <= '2026-04-04 00:00:00'::timestamp without time zone)
↳ AND (repository = 'rust-lang/rust'::text))"
24          Rows Removed by Filter: 144909
25          Buffers: shared hit=1 read=8288
26      -> Hash (cost=33.67..33.67 rows=73 width=8) (actual time=0.098..0.099
↳ rows=73 loops=3)
27          Buckets: 1024 Batches: 1 Memory Usage: 11kB
28          Buffers: shared hit=51
29      -> Seq Scan on contributors_teams ct (cost=0.00..33.67 rows=73
↳ width=8) (actual time=0.036..0.082 rows=73 loops=3)
30          Filter: (team = 'compiler'::text)
31          Rows Removed by Filter: 1261
32          Buffers: shared hit=51
33 Planning:
34     Buffers: shared hit=191 read=10 dirtied=1
35 Planning Time: 1.721 ms
36 Execution Time: 50.628 ms

```

S3 — s indexy

Parallel Seq Scan na **file_activity** nahrazen **Nested Loop** s dvěma **Index Only Scan**: nejprve jsou načtení členové týmu z primárního klíče **contributors_teams** (73 řádků), poté jsou pro každého přispěvatele data čtena přímo z pokrývajícího indexu **idx_file_activity_repo_contrib_ts** s **Heap Fetches: 0**. Zrychlení je způsobeno především eliminací paralelizace a přeskočením desítek tisíc nesouvisejících řádků.

Výpis C.16: S3: EXPLAIN ANALYZE s indexy (Execution Time: 41,0 ms)

```

1 Sort (cost=2652.81..2702.86 rows=20021 width=56) (actual time=37.377..39.194 rows=20892 loops=1)
2   Sort Key: (count(*)) DESC
3   Sort Method: quicksort Memory: 3246kB
4   Buffers: shared hit=114 read=819
5   -> HashAggregate (cost=1022.17..1222.38 rows=20021 width=56) (actual time=30.170..33.664
↳ rows=20892 loops=1)
6     Group Key: fa.file_path
7     Batches: 1 Memory Usage: 3857kB
8     Buffers: shared hit=114 read=819
9     -> Nested Loop (cost=0.70..922.07 rows=20021 width=48) (actual time=0.112..15.052
↳ rows=50104 loops=1)
10       Buffers: shared hit=114 read=819
11       -> Index Only Scan using contributors_teams_pkey on contributors_teams ct
↳ (cost=0.28..5.55 rows=73 width=8) (actual time=0.042..0.063 rows=73 loops=1)
12         Index Cond: (team = 'compiler'::text)
13         Heap Fetches: 0
14         Buffers: shared hit=4
15       -> Index Only Scan using idx_file_activity_repo_contrib_ts on file_activity fa
↳ (cost=0.42..11.38 rows=118 width=56) (actual time=0.014..0.164 rows=686 loops=73)
16 "         Index Cond: ((repository = 'rust-lang/rust'::text) AND (contributor_id =
↳ ct.contributor_id) AND (""timestamp"" >= '2025-06-01 00:00:00'::timestamp without time zone)
↳ AND (""timestamp"" <= '2026-04-04 00:00:00'::timestamp without time zone))"

```

```

17           Heap Fetches: 0
18           Buffers: shared hit=110 read=819
19 Planning:
20   Buffers: shared hit=8
21 Planning Time: 0.312 ms
22 Execution Time: 40.965 ms

```

S4 — bez indexů

Bez indexů jsou v plánu dva sekvenční průchody: `Seq Scan` nad `issue_labels_history` a `Parallel Seq Scan` nad `issue_event_history` s filtrací `is_pr` a příslušného repozitáře. Řazení mezivýsledků pro okenní funkce tabulky `issue_labels_history` probíhá operací `Sort (external merge Disk: 1 664 kB)`, protože data se nevejdou do paměti. Odhadované náklady jsou 24958 a doba vykonávání 111,4 ms.

Výpis C.17: S4: EXPLAIN ANALYZE bez indexů (Execution Time: 111,4 ms)

```

1 WindowAgg (cost=24925.41..24958.09 rows=1000 width=12) (actual time=110.496..110.606 rows=32 loops=1)
2   " Buffers: shared hit=6940, temp read=208 written=209"
3   CTE in_state_periods
4     -> Merge Join (cost=18341.59..24865.36 rows=1 width=40) (actual time=63.673..109.308 rows=810 loops=1)
5       Merge Cond: (label_transitions.issue = ordered_transitions.issue)
6       "
7         Join Filter: ((tsrange(label_transitions."timestamp", COALESCE(label_transitions.next_ts,
8         ↪ 'infinity'::timestamp without time zone), '[]'::text) && tsrange(ordered_transitions."timestamp",
9         ↪ COALESCE(ordered_transitions.next_ts, 'infinity'::timestamp without time zone), '[]'::text)) AND (NOT
10        ↪ isempty((tsrange(label_transitions."timestamp", COALESCE(label_transitions.next_ts, 'infinity'::timestamp
11        ↪ without time zone), '[]'::text) * tsrange(ordered_transitions."timestamp",
12        ↪ COALESCE(ordered_transitions.next_ts, 'infinity'::timestamp without time zone), '[]'::text))))"
13       Rows Removed by Join Filter: 806
14       "
15         Buffers: shared hit=6940, temp read=208 written=209"
16       -> Subquery Scan on label_transitions (cost=8863.39..10238.30 rows=30888 width=24) (actual
17       ↪ time=38.908..63.224 rows=26440 loops=1)
18         Filter: (label_transitions.action = 'ADDED'::text)
19         Rows Removed by Filter: 25008
20         "
21         Buffers: shared hit=2477, temp read=208 written=209"
22         -> WindowAgg (cost=8863.39..9709.49 rows=42305 width=30) (actual time=38.906..60.271
23         ↪ rows=51448 loops=1)
24         "
25         Buffers: shared hit=2477, temp read=208 written=209"
26         -> Sort (cost=8863.39..8969.15 rows=42305 width=22) (actual time=38.896..42.829
27         ↪ rows=51448 loops=1)
28         "
29         Sort Key: issue_labels_history.issue, issue_labels_history."timestamp"
30         Sort Method: external merge  Disk: 1664kB
31         "
32         Buffers: shared hit=2477, temp read=208 written=209"
33         -> Seq Scan on issue_labels_history (cost=0.00..5612.56 rows=42305 width=22)
34         ↪ (actual time=0.007..19.173 rows=51448 loops=1)
35         Filter: (is_pr AND (repository = 'rust-lang/rust'::text) AND (label =
36         ↪ 'S-waiting-on-review'::text))
37         Rows Removed by Filter: 157589
38         Buffers: shared hit=2477
39         -> Materialize (cost=9478.21..14547.13 rows=62 width=24) (actual time=24.735..43.901 rows=1645
40         ↪ loops=1)
41         Buffers: shared hit=4463
42         -> Subquery Scan on ordered_transitions (cost=9478.21..14546.97 rows=62 width=24) (actual
43         ↪ time=24.730..43.700 rows=671 loops=1)
44         Filter: (ordered_transitions.event_type = ANY ('{created,reopened}'::text[]))"
45         Rows Removed by Filter: 33132
46         Buffers: shared hit=4463

```

```

29         -> WindowAgg (cost=9478.21..14114.39 rows=34607 width=33) (actual time=24.648..41.527
        ↪ rows=33803 loops=1)
30             Buffers: shared hit=4463
31         -> Gather Merge (cost=9478.21..13508.76 rows=34607 width=25) (actual
        ↪ time=24.632..29.223 rows=33803 loops=1)
32             Workers Planned: 2
33             Workers Launched: 2
34             Buffers: shared hit=4463
35         -> Sort (cost=8478.18..8514.23 rows=14420 width=25) (actual
        ↪ time=21.574..22.454 rows=11268 loops=3)
36             Sort Key: issue_event_history.issue,
        ↪ issue_event_history."timestamp"
37             Sort Method: quicksort Memory: 1514kB
38             Buffers: shared hit=4463
39             Worker 0: Sort Method: quicksort Memory: 1212kB
40             Worker 1: Sort Method: quicksort Memory: 1068kB
41         -> Parallel Seq Scan on issue_event_history (cost=0.00..7482.06
        ↪ rows=14420 width=25) (actual time=0.257..18.223 rows=11268 loops=3)
42             Filter: (is_pr AND (repository = 'rust-lang/rust'::text) AND
        ↪ (event = ANY ('{created,closed,merged,reopened}'::text[])))
43             Rows Removed by Filter: 130861
44             Buffers: shared hit=4373
45     CTE daily_deltas
46     -> HashAggregate (cost=0.08..0.10 rows=2 width=12) (actual time=110.152..110.196 rows=557 loops=1)
47         Group Key: ((lower(in_state_periods.valid_period))::date)
48         Batches: 1 Memory Usage: 129kB
49         Buffers: shared hit=4401, temp read=208 written=209"
50     -> Append (cost=0.00..0.07 rows=2 width=8) (actual time=63.677..109.733 rows=1620 loops=1)
51         Buffers: shared hit=4401, temp read=208 written=209"
52     -> CTE Scan on in_state_periods (cost=0.00..0.03 rows=1 width=8) (actual time=63.676..109.478
        ↪ rows=810 loops=1)
53         Buffers: shared hit=4401, temp read=208 written=209"
54     -> CTE Scan on in_state_periods in_state_periods_1 (cost=0.00..0.03 rows=1 width=8) (actual
        ↪ time=0.003..0.152 rows=810 loops=1)
55         Filter: (NOT upper_inf(valid_period))
56     InitPlan 3 (returns $3)
57     -> Aggregate (cost=0.05..0.06 rows=1 width=32) (actual time=0.049..0.049 rows=1 loops=1)
58     -> CTE Scan on daily_deltas (cost=0.00..0.04 rows=1 width=8) (actual time=0.001..0.032 rows=321
        ↪ loops=1)
59         Filter: (event_date < '2025-06-01'::date)
60         Rows Removed by Filter: 236
61     -> Merge Left Join (cost=59.88..70.06 rows=1000 width=12) (actual time=110.439..110.448 rows=32 loops=1)
62         Merge Cond: (((d.d)::date) = dd.event_date)
63         Buffers: shared hit=4401, temp read=208 written=209"
64     -> Sort (cost=59.83..62.33 rows=1000 width=8) (actual time=0.027..0.029 rows=32 loops=1)
65         Sort Key: ((d.d)::date)
66         Sort Method: quicksort Memory: 26kB
67     -> Function Scan on generate_series d (cost=0.00..10.00 rows=1000 width=8) (actual
        ↪ time=0.014..0.016 rows=32 loops=1)
68     -> Sort (cost=0.05..0.06 rows=2 width=12) (actual time=110.372..110.386 rows=459 loops=1)
69         Sort Key: dd.event_date
70         Sort Method: quicksort Memory: 51kB
71         Buffers: shared hit=4401, temp read=208 written=209"
72     -> CTE Scan on daily_deltas dd (cost=0.00..0.04 rows=2 width=12) (actual time=110.153..110.291
        ↪ rows=557 loops=1)
73         Buffers: shared hit=4401, temp read=208 written=209"
74 Planning Time: 0.611 ms
75 Execution Time: 111.363 ms

```

S4 — s indexy

Oproti variantě bez indexů jsou hlavní změny: `Seq Scan` na `issue_labels_history` nahrazen `Index Only Scan using idx_pr_labels_repo_label_issue_ts` s `Incremental Sort` (quick-sort v paměti místo external merge na disku). `Parallel Seq Scan` na `issue_event_history` nahrazen `Bitmap Heap Scan` za použití indexu `idx_issue_event_history_repo_evt_pr_only`, čímž odpadá paralelní průchod celou tabulkou. Dominantní náklady zůstávají v `Merge Join` pro výpočet průniku intervalů, který není přímo indexovatelný.

Výpis C.18: S4: EXPLAIN ANALYZE s indexy (Execution Time: 74,6 ms)

```
1 WindowAgg (cost=15086.91..15119.58 rows=1000 width=12) (actual time=73.821..73.848 rows=32 loops=1)
2   Buffers: shared hit=5926
3   CTE in_state_periods
4     -> Merge Join (cost=8068.80..15026.85 rows=1 width=40) (actual time=16.909..72.533 rows=810 loops=1)
5       Merge Cond: (label_transitions.issue = ordered_transitions.issue)
6       "
7         Join Filter: ((tsrange(label_transitions."timestamp", COALESCE(label_transitions.next_ts,
8         ↪ 'infinity'::timestamp without time zone), '[]'::text) && tsrange(ordered_transitions."timestamp",
9         ↪ COALESCE(ordered_transitions.next_ts, 'infinity'::timestamp without time zone), '[]'::text)) AND (NOT
10        ↪ isempty((tsrange(label_transitions."timestamp", COALESCE(label_transitions.next_ts, 'infinity'::timestamp
11        ↪ without time zone), '[]'::text) * tsrange(ordered_transitions."timestamp",
12        ↪ COALESCE(ordered_transitions.next_ts, 'infinity'::timestamp without time zone), '[]'::text))))))"
13       Rows Removed by Join Filter: 806
14       Buffers: shared hit=5926
15     -> Subquery Scan on label_transitions (cost=0.60..5757.67 rows=30888 width=24) (actual
16     ↪ time=0.060..36.455 rows=26440 loops=1)
17       Filter: (label_transitions.action = 'ADDED'::text)
18       Rows Removed by Filter: 25008
19       Buffers: shared hit=2112
20     -> WindowAgg (cost=0.60..5228.86 rows=42305 width=30) (actual time=0.059..33.384 rows=51448
21     ↪ loops=1)
22       Buffers: shared hit=2112
23     -> Incremental Sort (cost=0.60..4488.52 rows=42305 width=22) (actual time=0.051..16.041
24     ↪ rows=51448 loops=1)
25       "
26         Sort Key: issue_labels_history.issue, issue_labels_history."timestamp"
27         Presorted Key: issue_labels_history.issue
28         Full-sort Groups: 1543  Sort Method: quicksort  Average Memory: 27kB  Peak Memory:
29         ↪ 27kB
30       Buffers: shared hit=2112
31     -> Index Only Scan using idx_pr_labels_repo_label_issue_ts on issue_labels_history
32     ↪ (cost=0.42..2975.90 rows=42305 width=22) (actual time=0.032..8.224 rows=51448
33     ↪ loops=1)
34       Index Cond: ((repository = 'rust-lang/rust'::text) AND (label =
35       ↪ 'S-waiting-on-review'::text))
36       Heap Fetches: 4281
37       Buffers: shared hit=2112
38     -> Materialize (cost=8068.20..9193.08 rows=62 width=24) (actual time=16.822..33.894 rows=1645
39     ↪ loops=1)
40       Buffers: shared hit=3814
41     -> Subquery Scan on ordered_transitions (cost=8068.20..9192.93 rows=62 width=24) (actual
42     ↪ time=16.812..33.704 rows=671 loops=1)
43       Filter: (ordered_transitions.event_type = ANY ('{created,reopened}'::text[]))"
44       Rows Removed by Filter: 33132
45       Buffers: shared hit=3814
46     -> WindowAgg (cost=8068.20..8760.34 rows=34607 width=33) (actual time=16.743..31.517
47     ↪ rows=33803 loops=1)
48       Buffers: shared hit=3814
49     -> Sort (cost=8068.20..8154.72 rows=34607 width=25) (actual time=16.726..19.297
50     ↪ rows=33803 loops=1)
51       "
52         Sort Key: issue_event_history.issue, issue_event_history."timestamp"
53
```

```

34             Sort Method: quicksort Memory: 3587kB
35             Buffers: shared hit=3814
36             -> Bitmap Heap Scan on issue_event_history (cost=480.42..5459.04 rows=34607
               ↳ width=25) (actual time=1.526..8.569 rows=33803 loops=1)
37             "
               Recheck Cond: ((repository = 'rust-lang/rust'::text) AND (event = ANY
               ↳ ('{created,closed,merged,reopened}'::text[])) AND is_pr)"
38             Heap Blocks: exact=3773
39             Buffers: shared hit=3814
40             -> Bitmap Index Scan on idx_issue_event_history_repo_evt_pr_only
               ↳ (cost=0.00..471.77 rows=34607 width=0) (actual time=1.152..1.152
               ↳ rows=33803 loops=1)
41             "
               Index Cond: ((repository = 'rust-lang/rust'::text) AND (event =
               ↳ ANY ('{created,closed,merged,reopened}'::text[])))"
42             Buffers: shared hit=41
43     CTE daily_deltas
44     -> HashAggregate (cost=0.08..0.10 rows=2 width=12) (actual time=73.489..73.534 rows=557 loops=1)
45         Group Key: ((lower(in_state_periods.valid_period)::date)
46         Batches: 1 Memory Usage: 129kB
47         Buffers: shared hit=5926
48     -> Append (cost=0.00..0.07 rows=2 width=8) (actual time=16.913..73.026 rows=1620 loops=1)
49         Buffers: shared hit=5926
50     -> CTE Scan on in_state_periods (cost=0.00..0.03 rows=1 width=8) (actual time=16.912..72.787
51         ↳ rows=810 loops=1)
52         Buffers: shared hit=5926
53     -> CTE Scan on in_state_periods in_state_periods_1 (cost=0.00..0.03 rows=1 width=8) (actual
54         time=0.003..0.143 rows=810 loops=1)
55         Filter: (NOT upper_inf(valid_period))
56     InitPlan 3 (returns $2)
57     -> Aggregate (cost=0.05..0.06 rows=1 width=32) (actual time=0.049..0.049 rows=1 loops=1)
58     -> CTE Scan on daily_deltas (cost=0.00..0.04 rows=1 width=8) (actual time=0.000..0.033 rows=321
59         ↳ loops=1)
60         Filter: (event_date < '2025-06-01'::date)
61         Rows Removed by Filter: 236
62     -> Merge Left Join (cost=59.88..70.06 rows=1000 width=12) (actual time=73.763..73.771 rows=32 loops=1)
63         Merge Cond: (((d.d)::date) = dd.event_date)
64         Buffers: shared hit=5926
65     -> Sort (cost=59.83..62.33 rows=1000 width=8) (actual time=0.027..0.029 rows=32 loops=1)
66         Sort Key: ((d.d)::date)
67         Sort Method: quicksort Memory: 26kB
68     -> Function Scan on generate_series d (cost=0.00..10.00 rows=1000 width=8) (actual
69         time=0.013..0.015 rows=32 loops=1)
70     -> Sort (cost=0.05..0.06 rows=2 width=12) (actual time=73.701..73.714 rows=459 loops=1)
71         Sort Key: dd.event_date
72         Sort Method: quicksort Memory: 51kB
73         Buffers: shared hit=5926
74     -> CTE Scan on daily_deltas dd (cost=0.00..0.04 rows=2 width=12) (actual time=73.491..73.621
75         ↳ rows=557 loops=1)
76         Buffers: shared hit=5926
77     Planning Time: 0.612 ms
78     Execution Time: 74.555 ms

```
