Analýza služeb kompilovaných v režimu Ahead-of-Time a Just-In-Time na platformě .NET

Bc. Noe Švanda

Diplomová práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Noe Švanda**
Osobní číslo:	**A22497**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Analýza služeb kompilovaných v režimu Ahead-of-Time a Just-In-Time na platformě .NET**
Téma práce anglicky:	**Analysis of Services Compiled in Ahead-of-Time and Just-in-Time Modes on the .NET Platform**

Zásady pro vypracování

1. Proveďte rozbor režimů kompilace v dotNET, zvolte dvě kompilační metody.
2. Vytvořte mikroslužby ve frameworku dotnet s využitím těchto kompilačních metod.
3. Popište konfigurace a nasazení ve vybrané platformě.
4. Připravte monitorovací a vizualizační nástroje pro srovnání výkonu kompilací.
5. Navrhněte testovací scénáře nad aplikačním stackem pro obě kompilační metody.
6. Srovnejte výsledky testování a tyto prezentujte v interaktivní podobě.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. KOKOSA, Konrad. Pro .NET memory management: for better code, performance and scalability. For professionals by professionals. New York: Apress, [2018]. ISBN 978-1484240267.
2. RICHTER, Jeffrey. CLR via C#: the common language runtime for .NET programmers. [4th ed.]. Redmond, Wash.: Microsoft Press, [2012]. ISBN 978-0735667457.
3. RICHARDSON, Chris. Microservices patterns: with examples in Java. Sebastopol, Calif.: O'Reilly Media, [2018]. ISBN 978-1617294549.
4. NICKOLOFF, James, KUENZIL, Steffen. Docker in action. 2nd ed. Greenwich, CT: Manning Publications, [2019]. ISBN 978-1617294761.
5. GARRISON, Josh, NOVA, Kelsey. Cloud native infrastructure: designing, building, and running scalable microservices applications. 1st ed. Sebastopol, Calif.: O'Reilly Media, [2017]. ISBN 978-1491984307.
6. GAMMELGAARD, Christian Horsdal. Microservices for .NET developers: a hands-on guide to building and deploying microservices-based applications using .NET Core. 2nd ed. Apress, [2021]. ISBN 978-1617297922.
7. LOCK, Andrew. ASP.NET Core in action. Greenwich. 2nd ed. CT: Manning Publications, [2021]. ISBN 978-1617298301.

Vedoucí diplomové práce: **doc. Ing. Petr Šilhavý, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **5. listopadu 2023**
Termín odevzdání diplomové práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomové práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne .....

podpis studenta

ABSTRAKT

Tato diplomová práce analyzuje možnosti kompilace služeb v režimu Ahead-of-Time a Just-In-Time na platformě .NET. Zaměřuje se na srovnání vývojového procesu, výstupu a výkonu služeb kompilovaných v obou režimech. Za tímto účelem je vytvořena aplikace pro testování scénářů a monitorování výsledků. Výsledkem práce je analýza vývojového postupu, programového výstupu a výkonnostních dat služeb. Na jejich základě jsou porovnány výhody a nevýhody obou režimů kompilace a analyzována jejich použitelnost.

Klíčová slova: .NET, kompilace, Ahead-of-Time, Just-In-Time, služby, microservice architektura, kontejner, telemetrie, metriky, testování

ABSTRACT

This thesis analyses the possibilities of compiling services in Ahead-of-Time and Just-In-Time mode on the .NET platform. It focuses on comparing the development process, output and performance of services compiled in both modes. To this end, an application is created to test the scenarios and monitor the results. The result of the work is an analysis of the development process, program output and service performance data. Based on these, the advantages and disadvantages of both compilation modes are compared and their applicability is analyzed.

Keywords: .NET, compilation, Ahead-of-Time, Just-In-Time, services, microservice architecture, container, telemetry, metrics, testing

Děkuji svému vedoucímu práce, doc. Ing. Petru Šilhavému, Ph.D., za jeho cenné rady, trpělivost a ochotu věnovat mi svůj čas. Dále bych chtěl poděkovat své rodině a přátelům za podporu a pochopení během mého studia.

OBSAH

ÚVOD

Programovací jazyky jsou základním kamenem softwarovévo vývoje respektive celého moderního světa v období informací. Představují způsob, kterým vývojář komunikuje s virtuálním prostředím operačního systému a následně hardwarovým rozhraním. Vývoj výkonu hardwaru, znalostí vývojářů a požadavků na vyvíjené systémy byl hnacím strojem technologického rozvoje. Postupným vývojem přicházeli další a další variace programovacích jazyků, některé rozdílné inkrementálně, jiné zcela diametrálně. Významným mezníkem v přístupu k tvorbě a běhu strojového kódu je vznik virtuálních strojů, které umožňují běh kódu nezávisle na HW. Tento přístup umožňuje vývojářům psát kód v jazyce, který je jim přirozený a následně jej spouštět na různých platformách.

.NET je platforma od společnosti Microsoft, která umožňuje vytvářet kód určený pro následnou kompilaci za běhu (Just-in-Time, dále JIT) a spuštění pomocí tzv. běhového prostředí (Common Language Runtime, dále CLR), jenž operuje jako virtuální stroj. Jedná se o relativně vyvinutou a zkušenou platformu s využitím v mnoha projektech a firmách. Přesto právě na této platformě byla dodatečně vyvinuta možnost pro PC platformy kompilovat do nativního kódu (Ahead-of-Time, dále AOT), který je spouštěn přímo na operačním systému ve specifické architektuře. Tato funkce přichází do období rozmachu vývoje a migrace nativních cloudových řešení. Ty charakterizuje snaha dodávat pouze nezbytnou část infrastruktury a zpoplatnit reálnou dobu běhu systému s režií. Právě v prostředí cloudu mají nastávat situace, kdy bude využití služeb zkompilovaných do nativního kódu výhodnější. Jaké jsou však rozdíly vývojového procesu aplikace kompilované AOT? V kterých případech takto vytvořený program exceluje či selhává? A lze kvantifikacovat rozdíly mezi JIT kopilací pro běhové prostředí a nativní AOT kompilací?

Tato práce se zabývá porovnáním vývojového procesu, programového výstupu a výkonu služeb kompilovaných JIT a AOT na platformě .NET. Cílem je zjistit, zda a v jakých případech je možné využít AOT kompilace pro zvýšení výkonu a efektivnější běh aplikací. Výsledkem práce je kvantifikace, respektive srovnání výkonu a chování JIT a AOT kompilace na platformě .NET. Na základě těchto výsledků je možné posoudit a doporučit vhodné případy pro využití AOT kompilace.

I. TEORETICKÁ ČÁST

1 PLATFORMA .NET

Platforma .NET od společnosti Microsoft představuje komplexní sadu nástrojů k vývoji aplikací v podporovaných jazycích. Je multiplatformní a umožňuje vývoj pro operační systémy (dále OS) jako Windows, Linux, macOS, ale i pro mobilní platformy a zařízení Internet of Things (dále IoT). Vývojáři mohou využívat nástroje pro vývoj webových aplikací, desktopových aplikací, mobilních aplikací a dalších. Platforma .NET je postavena na dvou hlavních nástrojích. Prvním z nich je *CLR*, běhové prostředí zodpovídající za běh aplikací. Druhým nástrojem je *.NET CLI* (Command Line Interface, dále CLI), konzolový nástroj, zodpovědný za interakci s dílčími nástroji platformy .NET.

1.1 Historie

Využití runtime prostředí, respektive v originální podobě virtuálního stroje, má historický původ. V dřívějších dobách byli programátoři limitování nutností kompilace kódu do nativní reprezentace přímo pro architekturu systému. Kód vytvořen pro jednu konkrétní architekturu se zpravidla neobešel bez modifikací, pokud měl fungovat i na odlišné architektuře.

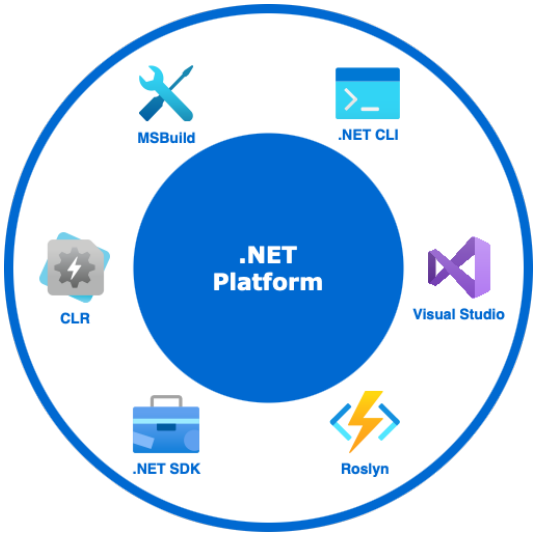
V průběhu 90. let 20. století představila společnost Sun-Microsystems virtuální stroj Java Virtual Machine (dále JVM). Jedná se o komponentu runtime prostředí Javy, která zprostředkovává spuštění specifického kódu, správu paměti, vytváření tříd, typů a další. Kompilací Javy do tzv. bytecode (Intermediate Language, dále IL), tedy provedením mezikroku v procesu transformace zdrojového kódu do strojového kódu, je získána reprezentace programu, jenž běží na každém zařízení s implementovaným JVM. V rámci JVM dochází k finálním krokům mezi které patří interpretace (JIT kompilace) bytecode do nativního kódu pro cílovou architekturu systému.

Microsoft v reakci na JVM vydal v roce 2000 první .NET Framework, který umožňoval spouštět kód v jazyce C# na operačním systému Windows. Cílem prvních verzí .NET Framework nebylo primárně umožnit vývoj pro různé zařízení a operační systémy, ale zprostředkovat lepší nástroje pro vývoj aplikací. [1] V roce 2014 byla vydána první multiplatformní verze platformy .NET. Ta nese název .NET Core a umožňovala spouštět kód v jazyce C# na operačních systémech Windows, Linux a macOS.

1.2 Architektura

Platforma .NET je postavena na několika klíčových komponentách, které zajišťují běh aplikací a poskytují nástroje pro vývoj aplikací. Mezi nejdůležitější komponenty patří:

- **Common Language Runtime** - CLR je základním kamenem .NET a poskytuje běhové prostředí pro spouštění aplikací na platformě. Překládá IL do nativního



Obrázek 1.1 .NET platforma

kódu, spravuje alokaci paměti a garbage collection (dále GC), zajišťuje zpracování výjimek (exceptions). CLR také kontroluje datové typy, interoperabilitu a zprostředkovává služby nezbytné pro spouštění nejrůznějších aplikací .NET. [2]

- **.NET CLI** - Všestranný nástroj pro vývoj, kompilaci a nasazení aplikací .NET prostřednictvím rozhraní příkazové řádky. Podporuje širokou škálu operací, od vytváření projektů a správy závislostí až po testování a publikování aplikací. [3] Prostředí .NET CLI je multiplatformní a dodává sjednocené rozhraní uživatelských nástrojů pro vývoj aplikací .NET.

- **Microsoft Build** - Microsoft Build (dále MSBuild) je engine používaný v platformě .NET, který umožňuje sestavovat aplikace a vytvářet balíčky pro nasazení. [4] Tento nástroj používá k organizaci a řízení procesu sestavení projektový soubor *csproj* na bázi Extensible Markup Language (dále XML). Pomocí něj je zajištěna kontrola nad kompilací a průběhem sestavení. V rámci procesu sestavení lze doplnit vlastní úlohy a cíle kompilace, což poskytuje flexibilitu sestavení pro komplexní procesy ve velkých projektech.

- **Nástroje .NET Software Development Kit** (dále SDK) - Soubor nástrojů a knihoven podporujících vývoj, debugování a testování aplikací .NET. Zahrnují různé CLI a GUI nástroje, které pomáhají vývojářům spravovat práci s kódem, optimalizovat výkon a zajistit kvalitní výstup programu v platformě .NET. [5]

- **Roslyn** - Roslyn je sada kompilátorů platformy .NET, která poskytuje bohaté Application Programming Interface (dále API) pro analýzu kódu. [6] Umožňuje vývojářům používat implementace kompilátorů jazyka C# a VB.NET jako

služby. Roslyn zlepšuje výkonnost vývojářů poskytnutím funkcí jako je refaktoring, generování kódu a sémantická analýza.

- **NuGet** - Správce balíčků pro platformu .NET. dodává standardizovanou metodu správy externích knihoven, na nichž závisí aplikace v .NET. [7] Zjednodušuje proces inkorporace knihoven, systémových i třetích stran, do projektu. Rovněž spravuje závislosti, čímž zajišťuje, že projekty zůstávají aktuální a kompatibilní. Tento nástroj je téměř nezbytný pro vývoj na platformě .NET, neboť umožňuje modulární vývoj softwaru (dále SW).

1.3 Frameworky

Platforma .NET poskytuje mnoho frameworků a technologií pro vývoj aplikací. Jednotlivé frameworky plní různé role a poskytují různé úrovně funkcionality pro vývoj aplikací. [5] Následující seznam obsahuje některé z nejpoužívanějších frameworků v rámci platformy .NET:

- **.NET** - Hlavní framework platformy .NET pro vývoj SW. Podporuje tvorbu a provoz moderních aplikací a služeb. [5] Původně známý jako .NET Framework a primárně zaměřen na prostředí Windows, s příchodem .NET Core a novějších verzí se vyvinul v modulární platformu s open-source zdrojovým kódem známou jednoduše jako .NET. Umožňuje vývojářům vytvářet aplikace, které jsou škálovatelné, výkonné a multiplatformní.

- **ASP.NET** - Robustní framework pro vytváření webových aplikací a služeb. Je součástí ekosystému .NET a navržený tak, aby umožňoval vývoj vysoce výkonných, dynamických webových stránek, API a webových aplikací v reálném čase. [8] ASP.NET podporuje jak webové formuláře, tak architekturu Model-View-Controller (dále MVC). S uvedením ASP.NET Core byl framework přepracován pro cloudové nasazení, škálevatolnost, vývoj napříč platformami a vysoký výkon. Poskytuje komplexní základ pro vývoj moderních webových aplikací, které lze spustit jak na Linuxu, Windows tak macOS. ASP.NET Core také představuje Blazor, který umožňuje vývojářům používat C# při vývoji webu, což dále zvyšuje všestrannost platfomy. [9] Vývojářům, kteří chtějí využít .NET pro vývoj webu, poskytuje ASP.NET komplexní a flexibilní sadu nástrojů pro vytváření všech řešení, od malých webů až po složité webové aplikace.

- **MAUI** - Moderní specializovaný framework pro vývoj aplikací napříč platformami v rámci ekosystému .NET. Umožňuje vývojářům vytvářet aplikace pro Android, iOS, macOS a Windows z jedné kódové báze. Zakládá na populárních konceptech z Xamarin.Forms a zároveň rozšiřuje jeho možnosti na desktopové

aplikace. .NET MAUI zjednodušuje vývojový proces tím, že poskytuje jednotnou sadu nástrojů pro vývoj uživatelského rozhraní na všech platformách s možností přístupu k funkcím specifickým pro platformu v případě potřeby. [10] Podporuje moderní vývojové vzory a nástroje, včetně Model-View-ViewModel (dále MVVM), datové vazby a asynchronní programování, což usnadňuje vytváření rozsáhlých aplikací. Předchůdcem MAUI je framework Xamarin, který sloužil pro vytváření mobilních aplikací na platformě .NET.

- **Blazor** - Specializovaný framework v rámci platformy .NET, jenž zprostředkovává vývojářům tvorbu interaktivních webových aplikací pomocí C# namísto JavaScript. Blazor může běžet na serveru (varianta Blazor Server), kde zpracovává požadavky a komunikuje s uživatelským rozhraním pomocí knihovny SignalR, která zabezpečuje websocket komunikaci. Nebo také jako WebAssembly (dále WASM) aplikace (varianta Blazor WASM), kdy dochází k přeložení C# kódu do nativního kódu WASM a je spouštěn přímo ve webovém prohlížeči vedle tradičních webových technologií, jako jsou Hypertext Markup Language (dále HTML) a Cascading Style Sheets (dále CSS). [5] Umožňuje vývojářům využít znalosti .NET pro tvorbu sofistikovaných webových aplikací, běžících na straně klienta v prohlížeči. Architektura Blazor je založená na komponentách a propaguje opětovné využití kódu pro tvorbu uživatelského rozhraní. Zároveň Blazor podporuje modulární vývojový přístup a poskytuje široké možnosti vývoje webových aplikací v ekosystému .NET.

1.4 Nástroje

Platforma .NET zprostředkovává rozsáhlou sadu nástrojů za účelem tvorby, sestavení a spuštění aplikace. V této kapitole budou uvedeny příklady těch nejvýznamnějších.

1.4.1 Knihovny

Knihovny představují soubor funkcí a tříd, které mohou být použity při vývoji ve více aplikacích. Typicky tvoří obecnou a logicky oddělenou část funkcionality aplikace. Umožňují distribuovat běžnou funkcionalitu napříč různými projekty. [5] Knihovny v .NET mohou být tvořeny binárnimi Dynamic Link Library (dále DLL) soubory nebo referencované jako samostatný projekt pomocí cesty adresáři. K distribuci knihoven obecně dochází pomocí balíčků NuGet. Využitím NuGet jsou knihovny zabaleny a sdíleny přes internetová úložiště. Běžnou praxí tvůrce platformy, programovacího jazyka nebo frameworku je poskytnutí sad knihoven, které usnadňují vývoj aplikací. Zároveň tyto knihovny zpravidla implementují nejběžnější funkcionality, které mohou programátoři vyžadovat. Typicky se jedná o přístup k souborovému systému, síti, databázím,

grafickému rozhraní a další. Následující seznam obsahuje některé z nejběžněji používaných knihoven v .NET:

- **System** - Poskytuje základní třídy, typy a rozhraní, které umožňují a podporují širokou škálu operací na úrovni systému, jako jsou vstupy a výstupy (Input/Output, dále IO), využití vláken, kolekcí a další. Je nezbytná prakticky pro každou aplikaci .NET.
- **System.IO** - Dodáva funkcionalitu čtení z datových proudů, souborů a zápis do nich. Zprostředkovává rozhraní pro práci se souborovým systémem.
- **System.Net** - Obsahuje třídy a abstrakce pro síťovou komunikaci a elektronickou poštu.
- **System.Data** - Zajišťuje přístup k datovým zdrojům, jako jsou databáze nebo XML soubory. Obsahuje knihovnu ADO.NET pro přístup k vybraným databázovým implementacím.
- **System.Collections** - Knihovnu tvoří rozhraní a třídy, které definují různé kolekce objektů, jako jsou seznamy, fronty, bitová pole, hašovací tabulky a slovníky.
- **System.Linq** - Zaštiťuje dotazování nad kolekcemi objektů pomocí Language Integrated Query (dále LINQ).
- **System.Threading** - Umožňuje správu vláken, synchronizační primitiva nebo například thread pool. Podporuje vývoj paralelizovaných aplikacích.
- **System.Security** - Spravuje ověřování, autorizaci a šifrování, a je základem pro vývoj bezpečných aplikací.
- **Entity Framework** - Object-Relational Mapping (dále ORM) framework, který umožňuje vývojářům pracovat s databázemi pomocí objektově orientovaného přístupu. Poskytuje abstrakci nad databázovými systémy a umožňuje vývojářům pracovat s daty pomocí objektů aplikace.

Kromě knihoven poskytovaných společností Microsoft existuje mnoho knihoven třetích stran. Za vývojem těchto knihoven mohou stát vývojařské komunity nebo být vydány velkými společnostmi. Běžně tyto knihovny navazují na sadu funkcí poskytovaných Microsoftem a rozšiřují je o další novou funkcionalitu, nebo portují známé existující projekty do platformy .NET. Mezi příklady nejznámějších knihoven třetích stran v .NET patří Dapper, AutoMapper, Newtonsoft.Json a další.

1.4.2 Integrované vývojové prostředí

Neméně důležitým prvkem vývoje aplikací je integrované vývojové prostředí (Integrated Development Environment, dále IDE). I když není povinné, pro spoustu vývojářů je jeho použití při tvorbě aplikací neodmyslitelné. IDE je nástroj, který zprostředkovává vývoj aplikací, správu projektů, debugování a další. IDE poskytuje uživatelské rozhraní, které umožňuje vývojářům vytvářet, upravovat a testovat kód. Dodává vizuální nástroje pro sestavení, testování a publikaci aplikace. Umožňuje provádět různorodé operace v kódu, jako je refaktorování, hledání chyb a ladění. Jedním z nejpoužívanějších IDE je Visual Studio, vyvíjené společností Microsoft. Visual Studio poskytuje prvotřídní podporu pro vývoj na platformě .NET. Mezi další populární IDE patří Visual Studio Code a JetBrains Rider.

1.4.3 Balíčky

Pro jednoduchou distribuci knihoven, jak systémových tak třetích stran, je využíván nástroj pro správu balíčků NuGet. Projekt, jenž má být distribuován je buďto opatřen atributem *<PackageOnBuild>* a sestaven nebo je využito příkazu *dotnet pack*. Takto vytvořené balíčky lze distribuovat např. přes NuGet.org, což je veřejný repozitář knihoven, který je dostupný pro všechny vývojáře. Možná je také implementace vlastního řešení pro sdílení balíčků. Distribuované knihovny jsou jednoduše importovatelné do projektu a umožňují snadnou správu závislostí.

1.4.4 Dokumentace

Dokumentace je důležitou součástí vývoje aplikací. Poskytuje informace o tom, jak používat nástroje a technologie, které jsou součástí platformy .NET. Obsahuje informace o API, knihovnách, nástrojích a dalších součástech platformy .NET. Oficiální dokumentace je dostupná online na oficiálních webových stránkách platformy .NET. Obsahuje podrobné informace o mnoha aspektech vývoje aplikací, jako je popis frameworků, tříd, rozhrání, ale i články a návody pro vývojáře. K nalezení je na webu `https://docs.microsoft.com/en-us/dotnet/`.

1.4.5 Jazyky

Aplikace je reprezentována zdrojovým kódem, který má v případě platformy .NET podobu jednoho z (nebo i kombinaci) podporovaných jazyků. Zdaleka nejčastěji využívaným je C#. Představuje všestranný, objektově orientovaný jazyk navržený tak, aby umožnil vývojářům jednoduše a srozumitelně vyvíjet širokou škálu aplikací. [5] Jedná se o staticky typovaný jazyk s širokou škálou moderních funkcí a každoročně podléhá vydání nové verze. Funkčně zaměřené programování je umožňěno pomocí ja-

zyka F#. Ten je primárně vhodný pro vědecké a datově náročné aplikace. [5] Zakládá na silném typování, umožňuje stručný, robustní a výkonný kód. Dalším podporovaným jazykem je VB.NET. Jedná se o moderní verzi jazyka VB, která je implementována na platformě .NET. VB je historický programovací jazyk vyvinutý společností Microsoft. Představený v roce 1991 a navržen s cílem uživatelské přívětivosti. Byl založen na jazyce BASIC a jeho drag-and-drop rozhraní umožňovalo snadné vytváření GUI. Díky tomu zpřístupnil programování širšímu okruhu lidí. V rámci VB byl kladen důraz na rychlý vývoj aplikací (RAD).

1.5 Aplikační struktura

Základním stavebním prvkem aplikace v .NET je projektový soubor. Jedná se o soubor na bázi XML disponující příponou *.csproj*. V rámci něj dochází ke konfiguraci a deklaraci, jak bude .NET CLI, respektive nástroje sestavení, s projektem pracovat. Zároveň jsou zde definované závislosti na další projekty a knihovny. [11] Mezi základní charakteristiky běžně určené v projektovém souboru patří verze .NET, verze projektu/assembly, seznam závislostí, konfigurace pro buildování, testování a publikaci. Pro tvorbu složitějších aplikací je možné využít více projektových souborů. Tyto souboury jsou seskupeny pomocí speciálního solution souboru. Jedná se o soubor sloužící ke kontejnerizaci a provázání veškerých projektových a pomocných souborů, jako jsou konfigurace setavení, pomocné skripty a další. Disponuje příponou *.sln*. Mezi další běžně používané konfigurační soubory patří následující:

- **appsettings.json** - obsahuje nastavení aplikace
- **launchsettings.json** - deklaruje konfiguraci pro spuštění aplikace
- **Directory.Build.props** - zprostředkovává globální nastavení atributů pro všechny projekty v solution
- **Directory.Build.targets** - obsahuje globální nastavení cílů kompilace pro všechny projekty v solution
- **NuGet.config** - nastavení pro balíčkovací správce NuGet

Dalším příkladem projektového souboru je *.fsproj* pro F# projekty, *.vbproj* pro VB projekty a *.nuspec* pro balíčkovací soubory NuGet. Speciálně pro IDE Visual Studio je využíván soubor *.dcsproj*, který obsahuje nastavení pro spustění a debugování aplikace spuštěné v Docker kontejneru.

1.6 Kompilace zdrojového kódu

Kompilace je proces transformace zdrojového kódu do jiné podoby. Kód je zpravidla kompilován do podoby bližší cílové architektuře, ať je touto architekturou OS, případně konkrétní hardware (dále HW), nebo runtime prostředí (virtuální stroj). [2] V rámci .NET jsou k dispozici dva hlavní režimy kompilace zdrojového kódu pro PC platformy: kompilace pro běhové prostředí (CLR) a kompilace do nativního kódu přímo pro cílovou architekturu (native AOT).

Cílem kompilace je převést zdrojový kód do podoby, kterou je možné spustit na cílovém zařízení. Platforma .NET podporuje zacílit na různá zařízení. Typickým cílem kompilace jsou platformy PC. Proces probíhá následujícím způsobem. V případě aplikace běžící v CLR, je kód kompilován do jazyka IL a poté spouštěn prostřednictvím běhového prostředí, přičemž je za běhu převeden na nativní kód. [2] Pro situace, kdy CLR není k dispozici nebo nemůže být využito, lze použít kompilace AOT pro vygenerování nativního kódu. [14] Pro zacílení na mobilní platformy je využito frameworku .NET MAUI ke kompilaci nativní aplikace pro speciální běhové prostředí v Android nebo iOS. [10]

1.6.1 Obecné postupy kompilace

Proces kompilace zahrnuje několik charakteristických kroků. V první řadě dochází k parsování zdrojového kódu. Při tomto kroku je analyzována syntaxe kódu a je vytvořena reprezentace programu pomocí Abstract Syntax Tree (dále AST). Následuje krok sémantické analýzy programu. V tento moment je analyzována logika a význam kódu. Dochází ke kontrole typů, vyhodnocení symbolů a další kontrole pravidel jazyka C#. Kompilace pokračuje krokem přeložení programu do IL. Kód IL je sada instrukcí nezávislá na procesoru, která je spustitelná v běhové prostředí CLR. [2] V neposlední řadě je IL spolu s metadaty o programu sestaven a zabalen do jednotky nasazení, tzv. *assembly*. Metadata mají význam v identifikaci, verzování a zabezpečení programu. V rámci kompilace je možno spustit dodatečné procesy. Jedním z volitelných procesů kompilace je linkování. Při něm je využit nástroj IL Linker jenž po procesu sestavení aplikace propojí více assembly tak aby vytvořily jeden spustitelný soubor nebo assembly. [12] To zahrnuje řešení odkazů mezi různými dll a integraci všech požadovaných zdrojů. Moderních verze .NET umožňují pro kompilaci nastavit proces odebrání nepoužívaného kódu. Tento proces se nazývá *trimming* neboli ořezávání a snižuje velikost aplikace tím, že vylučuje nepotřebné knihovny a větve kódu. [5] Je součástí rozsáhlé strategie *Tree shaking*, která optimalizuje výstupní aplikaci.

1.6.2 Kompilace pro CLR

CLR je zodpovědný za interpretaci IL kódu a jeho dodatečnou kompilaci do nativního kódu. Kompilace do nativního kódu je prováděna JIT za běhu aplikace, což zajišťuje, že kód je optimalizován pro konkrétní architekturu systému. V případě jazyka C# na platformě Windows slouží ke kompilaci spustitelný soubor *csc.exe*. Assembly typicky disponují příponou *.dll*, případně jsou zabaleny do spustilného souboru dle cílové platformy a výstupu. Takovýto výstup je následně připraven buďto ke spuštění za pomocí CLR nebo pro referenci a využití při tvorbě jiného .NET programu. [2] Speciálním případem kompilace pro CLR jsou aplikace R2R. Zdrojový kód je při sestavení zkompilován do podoby nativního kódu pomocí nástroje crossgen, čímž vzniknou sestavy Ready To Run (dále R2R). Za běhu se sestavy R2R načtou a spustí s minimální kompilací JIT, protože většina kódu je již v nativní podobě. CLR může přesto JIT kompilovat některé části kódu, které nelze staticky zkompilovat předem. [13] Využití je v aplikacích, které potřebují zkrátit dobu spouštění, ale zachovat určitou funkcionalitu nebo úroveň optimalizace poskytovanou JIT kompilací.

Mezi hlavní výhody kompilace pro CLR, respektive JIT kompilace se řadí zprostředkování následujícího:

- **Reflexe** - CLR umožňuje využívat reflexi. Tato funkcionalita poskytuje možnost získat informace o kódu za běhu aplikace. Za pomocí reflexe lze vytvářet aplikace, které jsou schopny měnit své chování za běhu.
- **Dynamické načítání** - CLR zprostřekovává možnost dynamicky načítat knihovny za běhu aplikace. Kód spuštěné aplikace je tedy možno rozšířit o další assembly. Aplikace tak mají možnost rozšířit své chování i po spuštění.
- **Větší bezpečnost** - Díky CLR je zajištěno, že aplikace nemůže přistupovat k paměti, která jí nebyla přidělena. Zabezpečuje tak, aby případný škodlivý program se nedostal k paměti jiných aplikací.
- **Správa paměti** - CLR zodpovídá za správu paměti pomocí GC. Zajišťuje, že paměť je uvolněna vždy, když ji aplikace již nepotřebuje. Tím je zabráněno tzv. memory leaks, tedy situacím, kdy alokovaná paměť programu není uvolněna.
- **Větší přenositelnost** - CLR umožňuje spuštění aplikace na všech OS, na kterých je dostupné běhové prostředí CLR.

Zatímco za nevýhody CLR se dá považovat:

- **Výkonnost** - I když určité optimalizace jsou prováděny pro konkrétní systém a architekturu, spousta dodatečný operací probíhá JIT za běhu aplikace. Tyto operace přidávají dodatečnou výkonnostní režii. Dalším výkonnostním měřítkem je rychlost startu aplikace, která je pro CLR vyšší než v případě aplikací v nativním kódu.

- **Velikost aplikace** - Přítomnost CLR nehraje zásádní roli v případě monolitických aplikací, ale v případě mikroslužeb je nutné CLR přidat ke každé službě. Tímto se zvyšuje velikost jedné aplikační instance.

1.6.3 Kompilace do nativního kódu

Přímá nativní AOT kompilace je proces, při kterém je kód kompilován do nativního kódu cílové architektury. Děje se tak v procesu sestavení programu ze zdrojového kódu. V případě platformy .NET je tato funkcionalita dostupná při použití jazyka C# a speciálních projektových atributů. Obecný postup nativní AOT kompilace sestává ze standartní kompilace a vytvoření IL kódu. Následuje přeložení IL a tvorba ILc souborů (c symbolizuje příponu souborů jazyka C). Ty jsou v následné fázi *IlcCompile* kompilovány pomocí překladače jazyka C. Výstupní soubory jsou prolinkovány ve fázi *LinkNative*. Konečným výstupem nativní AOT kompilace je spustitelný soubor. Tento soubor nabývá formátu dle cílové architektury, jenž byla definována v procesu sestavení. Takto vytvořený soubor je možné spustit přímo v OS bez potřeby CLR.

Tato funkcionalita prošla několika iteracemi. První možnosti sestavení aplikace v nativním kódu na .NET platformě byly aplikace Universal Windows Platform. Jednalo se o aplikace využívající specifické rozhraní, nativní pro vybrané produkty Microsoft. S verzí frameworku .NET 7 byly rozšířeny možnosti sestavení aplikace do podoby nativního kódu i pro další architektury a typy aplikací. [14] Tato nová funkcionalita získala vyráznější podporu v roce 2023 s vydáním frameworku .NET 8. Filozofie společnosti Microsoft je, že vývojáři by měli mít možnost využít AOT kompilace na platformě .NET, pokud je to pro daný scénář vhodné. Scénáře kladoucí takovéto požadavky se vyskytují především v cloudovém nasazení, respektivně při implementaci cloudové infrastruky s využitím platformy .NET.

Mezi hlavní výhody nativní AOT kompilace patří:

- **Nezávislost na CLR** - AOT kompilace umožňuje vytvořit aplikaci, která je schopna běžet bez nutnosti běhového prostředí. To znamená, že jediný soubor obsahuje všechny potřebné knihovny a závislosti.

- **Rychlejší start aplikace** - Aplikace zkompilované do nativního kódu cílové architektury se spouští daleko rychleji než aplikace využívající běhové prostředí.

- **Rychlá odpověď aplikace** - Díky tomu, že aplikace musí mít všechny typy a funkcionality vygenerovány ve chvíli kompilace, je rychlost první odpovědi aplikace vyšší než v případě aplikace využívající běhové prostředí a JIT kompilace.

Následující jsou slabé stránky aplikací kompilovaných do nativního kódu v .NET:

- **Absence dynamického načítání** - Neumožňuje dynamicky načítat assembly za běhu aplikace.
- **Bez generování kódu za běhu** - Nelze využít knihovnu System.Reflection.Emit pro generování kódu za běhu aplikace pomocí reflexí.
- **Vyžaduje trimming** - Trimming vyžaduje, aby veškeré nepřímo používané části byly explicitně deklarovány, jinak budou vyřazeny z výsledného kódu.
- **Připojení běhových knihoven** - Veškeré potřebné knihovny jsou součástí výsledného aplikačního souboru. To zvyšuje velikost samoteného programu ve srovnání s aplikacemi závislými na runtime protředí.
- **Kompatibilita knihoven s AOT** - Né všechny knihovny aplikují na svá rozhraní atributy signalizující nekompatibilitu s nativní AOT kompilací. To platí obzvlášť pro knihovny třetích stran. Využití těchto knihoven může bez varování selhat při kompilaci nebo za běhu AOT aplikace.
- **Cross platform sestavení** - .NET SDK umožňuje vytvářet nativní aplikace pouze pro OS stejný jako je ten, na kterém sestavení aplikace probíhá.

1.7 Běh kódu

Běh kódu na konkrétním zařízení vyžaduje aby nabýval kompatibilní podoby. To znamená, že jeho reprezentaci OS nativně rozumí a je zacílen na správnou architekturu. V případě nativní AOT kompilace v .NET tento kód získáme již při sestavení aplikace. Při využití kompilace do IL je nutné nativní kód získat JIT kompilací v CLR. Výsledná nativní reprezentace se v obou případech spouští zavoláním vstupní metody programu.

CLR poskytuje spravované prostředí pro spouštění aplikací .NET. Podporuje více programovacích jazyků, včetně jazyků C#, VB.NET a F#, a umožňuje jejich bezproblémovou spolupráci. Spravuje paměť prostřednictvím automatického GC, který přiděluje a sbírá paměť využitou objekty. [2] Tím pomáhá předcházet memory leaks a optimalizuje využití prostředků. CLR poskytuje komplexní model zabezpečení, který pomáhá chránit aplikace před neoprávněným přístupem, poškozením dat a dalšími bezpečnostními hrozbami. Vynucuje zásady zabezpečení, jako je zabezpečení přístupu ke kódu

(Code Access Security, dále CAS) a využití rolí při spuštění aplikace. [2] Zajišťuje, aby kód byl spouštěn s příslušnými oprávněními na základě svého původu a úrovně důvěryhodnosti, čímž chrání citlivá data a systémové prostředky. Důležitou funkcí pro stabilitu systému je zpracování výjimek. To v CLR zahrnuje detekci, propagaci a zpracování chyb nebo stavů, které mohou nastat během provádění programu. Mechanismus vyjímek umožňuje elegantně řešit neočekávané situace a zachovat stabilitu aplikace. CLR zprostředkovává dynamické generování typů za běhu, což aplikacím umožňuje za běhu dynamicky vytvářet a manipulovat typy dle potřeby. [2] Jednou z nejdůležitějších funkcí CLR představuje reflexe. Díky ní je umožněna validace a manipulace typů, attributů a metadat načtených z assembly za běhu. Reflexe dovoluje dotazovat se a upravovat typy a jejich attributy za běhu, dynamicky volat metody a přistupovat k informacím o metadatech. Díky tomu mají aplikace .NET využívající běhové prostředí výrazné možnosti introspekce a přizpůsobení. CLR obsahuje nástroje pro ladění a profilování, které vývojářům pomáhají debugovat a identifikovat problémy s aplikací. Aby mohl být kód z IL reprezentace spuštěn na systému, respektive HW stroje, musí být dodatečně JIT kompilován. Za tímto účelem existuje v CLR několik technik, které mají využití v specifických scénářích.

Oproti CLR, běh nativního kódu je závislý na konkrétní architektuře systému, pro kterou jsou nativní programové soubory vytvořeny. Nepodléhá další úpravě ze strany .NET nástrojů. Spuštění probíhá nativním příkazem v OS, který je schopen interpretovat a spustit daný soubor

1.8 Tvorba programu v .NET

Následující část popisuje obecnou koncepci a strukturu projektu aplikace v .NET. Součástí je postup pro tvorbu a vydání projektu. Blížší pozornost bude věnována tvorbě projektu využívajícího nativní AOT kompilace.

1.8.1 Obecný postup

Vytvoření aplikace v .NET zahrnuje několik kroků, které jsou obecně platné pro všechny typy aplikací. V první řádě je potřeba připravit vývojové prostředí, nainstalovat sadu nástrojů .NET SDK a provést počáteční konfiguraci. Následuje vytvoření projektu. Pomocí příkazu *dotnet new* nebo skrze GUI IDE je vytvořen nový projekt a solution soubor. Součástí je výběr typu projektu, jazyka, frameworku a dalších konfiguračních parametrů. Primární fází je samotná tvorba programu, jenž sestává z tvorby kódu, testování a ladění. [15] S procesem programování je spojena správa závislostí. Pomocí nástroje NuGet se referencují balíčky a knihovny, které dodají projektu vyžadovanou funkcionalitu. Následuná kompilace aplikace probíhá pomocí příkazu *dotnet build*, který

převede kód v podporovaném jazyce do IL. V případě AOT dochází k dodatečné kompilaci do nativního kódu dle cílové architektury. Finálním krokem je použití příkazu *dotnet publish* a vydání aplikace ve specifické konfiguraci.

1.8.2 Tvorba nativního programu

Pro tvorbu nativního programu v .NET je nutné využít speciálního atributu *PublishAoT* v projektovém souboru. Tento atribut je zodpovědný za konfiguraci projektu pro nativní AOT kompilaci. Pokud je v projektu zapnut atribut EmitCompilerGeneratedFiles, kompilátor generuje soubory, které obsahují podrobné informace o stavu zkompilované aplikace. Ty zahrnují i meziprodukty nebo výpisy strojového kódu, které jsou cenné pro procest ladění a analýzy výstupního produktu. [16] Pomáhají při zacílení na kompilaci do nativního AOT lépe pochopit, jak je vysokoúrovňový kód překládán do strojového kódu. Deklarace unmanaged nebo také nespravovaného rozhraní zahrnuje definování způsobů jakým jednotlivá rozhraní komunikují. Spravované rozhraní představuje zdrojový kód vytvářené aplikace .NET. Nespravované rozhraní jsou funkce operující mimo .NET aplikaci, například v C/C++ knihovně. [1] Pomocí deklarací je specifikován způsob volání. U nativních AOT aplikací je důležité, aby tato rozhraní byla přesně definována a dodržována, protože jakýkoli nesoulad nebo chyba v deklaraci může vést k chybám za běhu, které se hůře diagnostikují a opravují kvůli absenci runtime prostředí a dynamickým funkcím. V rámci nativní AOT kompilace dochází automaticky k ořezávání kódu a jeho linkování do jednoho spustitelného souboru.

1.8.3 Přehled podpory

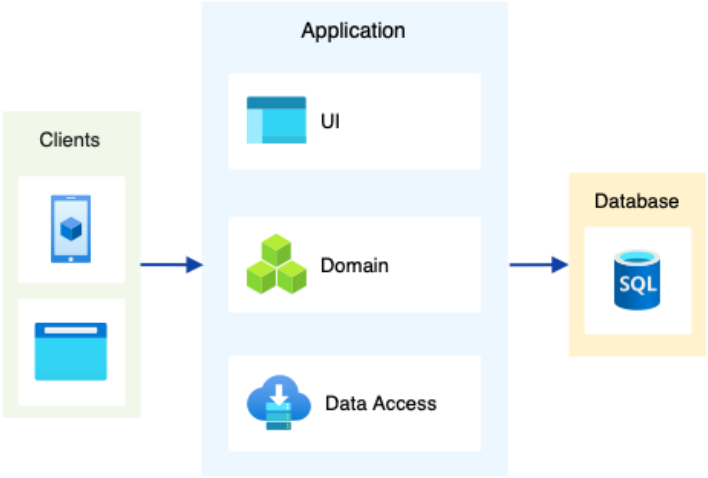
Následující přehled představuje rozsah funkcionality implementované v rámci .NET frameworku k datu vydání verze 8.0.4. [9]

- **REST minimal API** - Tvorba minimalistých služeb implementujících REST API.
- **gRPC API** - Komunikace mezi službami pomocí protokolu gRPC.
- **JSON Web Token Authentication** - Autentizace a autorizace za pomocí JSON Web Token (dále JWT) tokenů.
- **CORS** - Konfigurace Cross-Origin Resource Sharing.
- **HealthChecks** - Monitorování stavu aplikace.
- **HttpLogging** - Logování HTTP požadavků.
- **Localization** - Lokalizace aplikace.

- **OutputCaching** - Cachování výstupu.
- **RateLimiting** - Omezení počtu požadavků.
- **RequestDecompression** - Dekomprese požadavků.
- **ResponseCaching** - Cachování odpovědí.
- **ResponseCompression** - Komprese odpovědí.
- **Rewrite** - Přepisování adresy Uniform Resource Locator (dále URL).
- **StaticFiles** - Poskytování statických souborů.
- **WebSockets** - Komunikace pomocí WebSockets.

2 MICROSERVICE ARCHITEKTURA

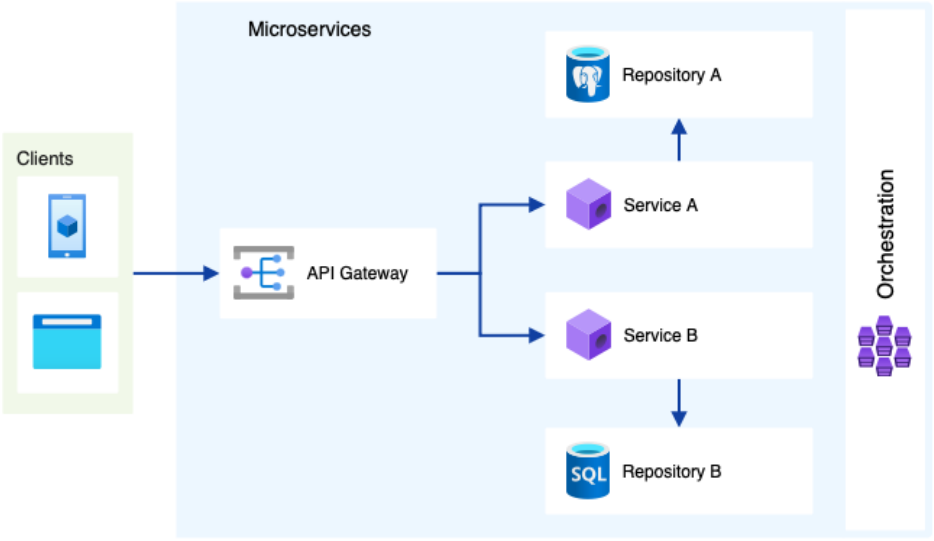
Při vývoji softwaru je možné aplikovat různé architektury a návrhové vzory. Za základní a vysoce rozšířenou architekturu lze považovat monolitickou architekturu. Aplikace využívající monolitickou architekturu sestávají z jedné kódové báze v níž se spojuje řešení veškeré doménové problematiky. [7] Jsou založeny na principu, že celá aplikace je spuštěna jako jeden proces. Obecně obsahují určitou formu logického členění, například na jednotlivé doménové moduly nebo služby, které seskupují související části aplikace. [7]



Obrázek 2.1 Monolitická architektura

Monolitická architektura je jednoduchá na vývoj, nasazení a testování. Při správném návrhu může poskytovat i jednoduchou možnost škálování, kdy aplikace je nasazena ve více identických instancích. [17] Jelikož i při logickém oddělení funkcionality je celá aplikace spjata pevnými vazbami, jakýkoliv zásah do jedné části aplikace může mít nepředvídatelné důsledky na ostatní části. Jednotná kódová báze zase může představovat limitující faktor pro rychlost vývoje, automatizace testovaní a nasazování. Samotné škálování může být vysoce neefiktivní a limitující, pokud je potřeba škálovat pouze určité části aplikace. [17]

Naproti tomu stojí architektura microservice. Ta je založena na principu rozdělení aplikace do samostatných služeb. Každá z těchto služeb je zodpovědná za určitou část funkcionality aplikace. Služby jsou navzájem nezávislé a komunikují mezi sebou pomocí definovaných rozhraní. [17] Tím je zajištěno, že každá služba může být vyvíjena, testována, nasazována a škálována nezávisle na ostatních. Tato architektura umožňuje vývojářům pracovat na menších a jednodušších částech aplikace, což zvyšuje produktivitu a umožňuje rychlejší iterace vývoje. Díky nezávislosti služeb je také možné dosáhnout vyšší odolnosti a škálovatelnosti aplikace.



Obrázek 2.2 Microservice architektura

2.1 Historie

Původ microservice architektury nelze přesně definovat, důležitý moment však nastal v roce 2011, kdy Martin Fowler publikoval článek *Microservices* na svém blogu. Článek je k nalezení na adrese `https://martinfowler.com/articles/microservices.html`. V tomto článku spolu s Jamesem Lewisem popsal způsob jakým lze tuto architekturu využít, její výhody a nevýhody. Dalším momentem, kdy microservice architektura nabyla popularity, bylo vydání knihy *Building Microservices* od Sama Newmana v roce 2015. Tato kniha popisuje způsob, jakým je možné využít microservice architekturu v praxi. Opravdový přelom přišel postupně, nástupem a popularizací virtualizace a kontejnerizace v průběhu let 2013 až 2015. Pomocí těchto technik lze vytvářet a spouštět mikroslužby v izolovaných prostředích. V rámci kontejnerů jsou mikroslužby nezávislé na operačním systému a hardwaru, na kterém je kontejner spouštěn. Nejdůležitější v tomto ohledu je nepochybně projekt Docker, který byl vydán v roce 2013. [18] Díky Dockeru bylo možno jednoduše definovat, vytvářet a spouštět kontejnerizované aplikace.

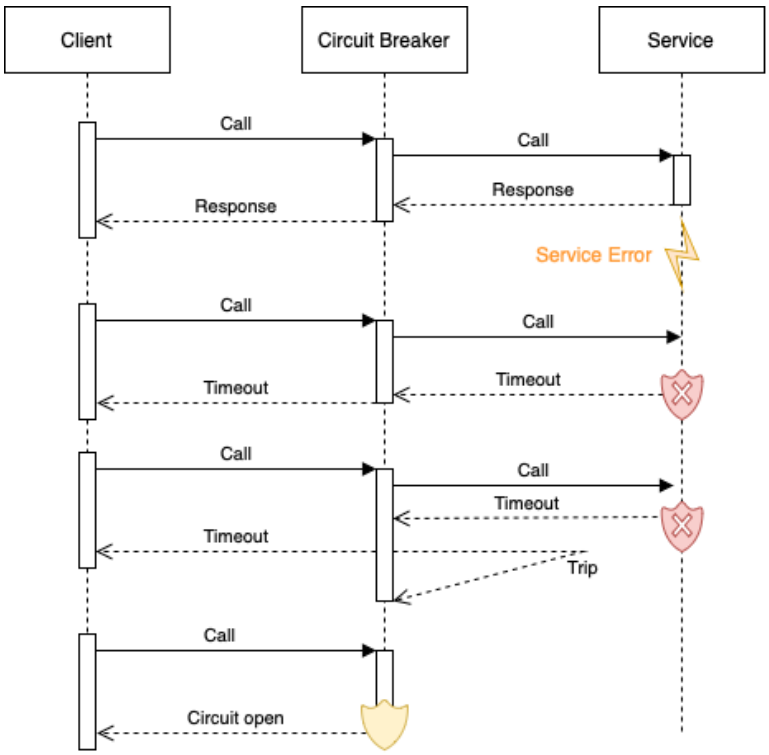
2.2 Základní principy

Microservice ahitektura stojí na několika základních principech. Tyto principy nejsou jen teoretické, mají přímý dopad na to, jak jsou služby vyvíjeny, nasazovány a udržovány. Jejich dodržení je klíčové k úspěšné implementaci architektury. [17]

- **Decentralizace** - Definuje, že každá služba zodpovídá za určitou část funkcionality aplikace. Služby jsou navzájem nezávislé a komunikují mezi sebou po síti pomocí deklarovaných rozhraní. Každá služba je samostatně nasaditelná a zod-

povídá za svá data. [19] Tím je zajištěno, že každá služba může být vyvíjena, nasazována a škálována nezávisle na ostatních.

- **Odolnost** - Odolnost nebo také robustnost microservice architektury je definována schopností systému zůstat v provozu i přes výskyt chyb v jeho dílčích částech. To znamená, že pokud jedna služba selže, zbytek systému může pokračovat v provozu. Toho je dosaženo použitím specifických vzorů. *Circuit Breaker* představuje jeden z těchto vzorů. Zajišťuje aby služby mimo provoz nebyly zbytečně zatíženy požadavky. Komunikace v tomto vzoru probíhá za pomocí stavového automatu, jenž za splnění určitých kritérií (například konkrétního počtu neúspěšných požadavků) se sepne do stavu *Open* a na určitou dobu přestane zasílat další požadavky. [19]



Obrázek 2.3 Circuit Breaker vzor

- **Kontejnerizace** - Kontejnerizace představuje proces zabalení služby spolu s veškerými závislostmi jako je OS, prostředí a konfigurace. Kontejner tvoří základní spustitelnou jednotku microservice architektury. Je založen na minimalistickém obrazu OS k němuž jsou dodány potřebné nástroje, knihovny a samotná služba. Takto vytvořený kontejner je virtualizován, tedy spuštěn jako samostatný virtuální OS v rámci hostitelského OS. [18] Moderní metody kontejnerizace zakládájí na technologiích jako je například Docker nebo Podman. Tyto nástroje poskytují ekosystém pro kompletní proces tvorby, sdílení a nasazení kontejnerů. Hlavní

výhodou kontejnerizace je umožňění běhu aplikace v odděleném prostředí s vybranou konfigurací při co nejmenší režii.

- **Orchestrace** - Rozšiřováním počtu služeb respektive kontejnerů se jejich správa stává složitou. Nástroje pro orchestraci pomáhají automatizovat nasazení, škálování a správu kontejnerů. Mezi oblíbené orchestrační nástroje patří Kubernetes, Docker Swarm a Marathon. [7] Zodpovědností těchto nástrojů je řešit problémy jako vyhledávání služeb, vyvažování zátěže, přidělování prostředků a škálování na základě vytížení služeb.

- **Škálování** - Architektura microservice zvyšuje škálovatelnost aplikace a umožňuje ji provádět jen v rámci konkrétní části, na úrovni dílčích služeb. Škálování probíhá typicky vytvořením několika instancí stejné služby a nastavením služby v roli tzv. *Load Balancer*. Ta distribuuje požadavky na jednotlivé služby podle definovaných pravidel. Typicky se cíl požadavku určuje dle hashe IP adresy požadavku nebo na základě váhy či doby odpovědi instance. [19] Pomocí těchto technik je v aplikaci dosaženo zlepšení schopnosti zvládat velké objemy požadavků za využití co nejmenšího množství systémových prostředků.

2.3 Komponenty

Architektura microservice rozděluje aplikaci do menších, nezávislých služeb, z nichž každá plní samostatnou funkci. Pro zajištění běhu takto distribuovaného systému je doporučeno využít vybraných komponent. Jedná se o technologie nebo služby plnící specifickou roli podporující microservice architekturu. [7] Využití těchto komponent není nutně povinné, nicméně s rostoucím rozsahem aplikace může být jejich zapojení kritické pro udržitelný provoz. Následující část se zabývá těmito klíčovými architektonickými komponentami.

2.3.1 Obecné komponenty

- **API Gateway** - Brána, která slouží jako vstupní bod pro komunikaci s mikroslužbami. Zajišťuje směrování požadavků, autentizaci, autorizaci, zabezpečení a další funkce, které jsou společné pro všechny služby. API Gateway může také poskytovat další funkce, jako jsou cachování, transformace zpráv a řízení toku dat. Tím zjednodušuje a centralizuje správu komunikace mezi klienty a mikroslužbami.

- **Service Discovery** - Mechanismus, který umožňuje mikroslužbám dynamicky najít a komunikovat s ostatními službami v systému. To je důležité pro dynamické

škálování, nasazování a správu služeb. Service Discovery může být implementován pomocí centrálního registru služeb nebo distribuovaného protokolu. [17]

- **Load Balancer** - Služba rozděluje požadavky mezi několik instancí stejné služby, aby se zajistila rovnoměrná zátěž a zvýšila se odolnost proti chybám. Load Balancer může být implementován jako hardwarové zařízení nebo softwarová služba, která poskytuje rozhraní pro konfiguraci a správu zátěže.

2.3.2 Komunikační systémy

Mikroslužby spolu komunikují skrze rozhraní prostřednictvím vybraných protokolů, nástrojů a vzorů. Mezi nejčastěji využívané patří:

- **REpresentational State Transfer** (dále REST) - Představuje vysoce rozšířenou možnost komunikace mezi mikroslužbami. Využívají se při ní standardní metody protokolu Hypertext Transfer Protocol (dále HTTP) k provádění operací na rozhraní identifikovaným prostřednictvím adresy URL. [20] Díky bezstavové povaze je rozhraní REST vysoce škálovatelné a vhodné pro veřejně přístupné služby. Má širokou podporu na různých platformách a v různých jazycích, což pomáhá zajistit interoperabilitu v rozmanitém ekosystému mikroslužeb.

- **Remote Procedure Call** (dále RPC) - Komunikační metoda používaná v distribuovaných systémech, včetně mikroslužeb. Definuje formu komunikace, kdy procedura volaná programem se spustí v jiném adresním prostoru. V případě microservice architektury je běžné použití, kdy služba takto spouští metodu z jiné služby. [21] Tato technika abstrahuje složitost síťové komunikace do jednoduchosti volání lokální funkce. Mezi běžné implementace RPC patří generic RPC (dále gRPC), Thrift anebo Apache Avro.

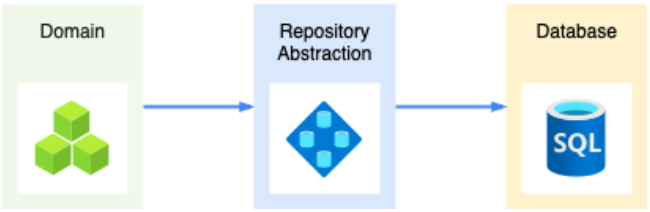
- **Message Broker** - Jedná se o komunikační vzor kdy broker - prostředník, spravuje asynchronní komunikaci mezi mikroslužbami pomocí front zpráv. Tato metoda odděluje mikroslužby tím, že jim umožňuje publikovat zprávy do fronty, aniž by znaly podrobnosti o tom, které služby je budou konzumovat. [7] Mezi běžné zprostředkovatele zpráv patří RabbitMQ, Apache Kafka a AWS SQS. Tento komunikační vzor zvyšuje odolnost proti chybám, protože broker zpráv může zajistit, že zprávy nebudou ztraceny při přenosu i když je konzumující služba dočasně nedostupná.

2.3.3 Databáze

V microservice architektuře si každá služba obvykle spravuje vlastní databázi podle vzoru *Database per Service*. Tato izolace umožňuje službám být volně provázané a

nezávisle nasaditelné, přičemž každé databázové schéma je přizpůsobeno konkrétním potřebám služby. [7] V závislosti na případu použití mohou služby používat různé typy databází. Structured Query Language (dále SQL) pro transakční data vyžadující silnou konzistenci a vlastnosti Atomocity Consistency Isolation Durability (dále ACID). Nebo Not only SQL (dále NoSQL) pro flexibilnější možnosti ukládání dat, které nabývají velkých objemů, nejsou definovány schématy nebo mají specifickou vazbu, například na čas.

Různorodost databázových technologií přináší výzvy, jako je jednotný přístup k datovým zdrojům. Ten je řešen pomocí vzoru *Repository*. Tento vzor využívá vytvoření obecného rozhraní, které definuje standartní operace pro přístup k datům. Pro každou databázovou technologii a její specifické rozhraní je vytvořena implementace tohoto rozhraní. [20] Služby následně pracují objekty, jako by byly součástí kolekce v paměti služby, tedy nezávisle na využitou databázovou technologii.



Obrázek 2.4 Repository vzor

Další častou problematikou je udržování konzistence dat v transakcích probíhajících napříč více službami. K řešení se využívájí specifické vzory, jako je například *Saga*. Saga je vzor, který řeší distribuované transakce formou lokálních transakcí v jednotlivých službách. Pokud při některé z dílčích transakcí dojde k chybě, je vyvolaná série kompenzačních transakcí, jenž vrátí systém do původního stavu. Řízení událostí může být vedeno formou choreografie, kdy jednotlivé služby zodpovídají za publikování událostí spouštějících následující transakci. Alternativně může být využitý orchestrátor, který distribuovanou transakci řídí. [22]

2.3.4 Bezpečnost

Bezpečnost v architektuře microservice je velmi důležitá, protože distribuovaná povaha těchto systémů přináší mnoho zranitelných míst. Bezpečnostní prvky se zaměřují na ochranu dat při přenosu i v klidovém stavu a zajišťují, že k službám a datům mají přístup pouze oprávněné subjekty. Mezi klíčové strategie patří implementace API Gateway s vestavěnými bezpečnostními prvky, jako je centrální logování, autentizace a autorizace. [7] Zásadní význam mají systémy správy identit a přístupu (Identity and Access Management, dále IAM), často integrované s tokeny Open Authorization (dále OAuth) a JWT pro správu identit uživatelů a řízení přístupu na základě definovaných

zásad. Zajištění šifrované komunikace mezi službami pomocí protokolů, jako je Transport Layer Security (dále TLS), chrání před odposlechem a manipulací. [17] Zásadní jsou také účinné strategie logování, auditování a monitorování, které poskytují možnost odhalovat bezpečnostní hrozby a reagovat na ně v reálném čase. Každá z těchto složek hraje klíčovou roli při vytváření bezpečné microservice aplikace a společně poskytují robustní obranné mechanismy proti interním i externím bezpečnostním rizikům.

2.4 Testování

Testování mikroslužeb je klíčové pro zajištění kvality a spolehlivosti systému. Mikroslužby lze testovat na několika následujících úrovních: [17] [20]

- **Jednotkové testy** - Také nazývané *Unit testy*, testují jednotlivé komponenty služby, jako jsou třídy, metody a funkce. Cílem je ověřit, že jednotlivé části fungují správně a splňují požadavky.
- **Integrační testy** - Ověřují komunikaci mezi službami a testují, zda jsou data přenášena a zpracovávána správně.
- **End-to-end testy** - Testují celý systém z pohledu uživatele. Cílem je ověřit, že systém funguje správně a splňuje požadavky a uživatelské scénáře.
- **Smoke testy** - Testují základní funkce systému, aby se ověřilo, že je správně sestaven, dokáže se spustit a provést základní operace.
- **Load testy** - Ověřují výkonnost systému za zátěžových podmínek. Cílem je zjistit, zda je systém schopen zvládnout požadavky uživatelů a odpovídat i při zátěži.
- **Penetrační testy** - Testují bezpečnost systému a identifikují potenciální bezpečnostní chyby. Cílem je odhalit slabá místa v systému a zlepšit jeho odolnost proti útokům.

Automatizované testování je klíčové pro rychlé a spolehlivé nasazení. Pomáhá odhalit chyby a problémy v raných fázích vývoje a minimalizuje riziko selhání v produkci. Testování microservice architektury je však složitější než testování monolitických aplikací, protože služby jsou nasazeny nezávisle. Automatizace pomáhá zjednodušit komplexní testovací strategie a zajišťuje, že jednotlivé části aplikace jsou spolehlivé.

2.5 Výhody a nevýhody

Mezi hlavní výhody microservice architektury lze zařadit následující:

- **Přizpůsobitelnost** - Mikroslužby umožňují rychlé, modulární a spolehlivé poskytování rozsáhlých a komplexních aplikací. Týmy mohou aktualizovat určité oblasti aplikace, aniž by to mělo dopad na celý systém, což umožňuje rychlejší iterace vývoje produktu.
- **Škálovatelnost** - Mikroslužby lze škálovat nezávisle, což umožňuje přesnější přidělování zdrojů na základě aktuálního stavu systému. Tím je řešena problematika proměnlivého a nesouměrného zatížení aplikace.
- **Odolnost** - Decentralizovaná povaha služeb pomáhá izolovat selhání na jedinou službu nebo jejich malou skupinu, čímž zabraňuje pádu celé aplikace.
- **Technologická rozmanitost** - Týmy si mohou vybrat nejlepší nástroje pro svoji práci. Podle potřeby lze používat různé programovací jazyky, databáze nebo jiné nástroje pro implementaci mikroslužby. Tato volnost vede k optimalizovaným řešením specifickým pro daný problém.
- **Flexibilita nasazení** - Mikroslužby lze nasazovat nezávisle, což je ideální pro praktiky Continuous Integration a Continuous Delivery (dále CI/CD). Umožňují průběžné aktualizace aplikace při minimalizaci prodlevy a rizika.
- **Modularita** - Microservice architektura zvyšuje modularitu, což usnadňuje vývoj, testování a údržbu aplikací. Jednotlivé vývojové týmy se mohou zaměřit na konkrétní doménovou logiku, což zvyšuje produktivitu a kvalitu výstupu. Rovněž umožňuje geograficky dislokované nasazení.

Zatímco mezi nevýhody patří:

- **Komplexnost** - Správa více služeb na rozdíl od monolitické aplikace přináší složitost při nasazování, monitorování a řízení komunikace mezi službami.
- **Správa dat** - Konzistence dat mezi službami může být náročná na údržbu, zejména pokud si každá mikroslužba spravuje vlastní databázi. Implementace transakcí napříč rozhraními vyžaduje pečlivou koordinaci.
- **Zpoždění sítě** - Komunikace mezi službami po síti přináší zpoždění, které může ovlivnit výkonnost aplikace. Ke zmírnění tohoto jevu jsou nutné efektivní komunikační protokoly a vzory.
- **Provozní režie** - S počtem služeb roste potřeba správy, orchestrace, monitorování a dalších provozních záležitostí. To vyžaduje použití vhodných technologií a nástrojů. Zároveň jsou tím kladeny požadavky na odborné znalosti vývojářů.

- **Složitost vývoje a testování** - Mikroslužby sice zvyšují flexibilitu vývoje, ale také komplikují testování, zejména pokud jde o testování *end-to-end*, které zahrnuje komunikaci více služeb.
- **Integrace služeb** - Zajištění bezproblémové spolupráce služeb vyžaduje robustní správu API, verzování a strategie zpětné kompatibility. Je nutné definovat jasná rozhraní a kontrakty, aby bylo možné služby snadno integrovat a rozšiřovat.

2.6 Nasazení založené na mikroslužbách

Efektivní nasazení mikroslužeb je klíčové pro využití jejich potenciálních výhod, jako je škálovatelnost, flexibilita a odolnost. Tato část se zabývá různými strategiemi nasazení, které jsou pro mikroslužby obzvláště vhodné, zejména v nativním cloudovém prostředí (cloud-native). Tyto strategie zajišťují, že mikroslužby lze efektivně spravovat a škálovat, dynamicky reagovat na změny zatížení a minimalizovat prostoje. [7]

2.6.1 Strategie

Existuje několik strategií nasazení, které jsou v microservice architektuře aplikovatelné:

- **Jedna služba na hostitele** - Strategie zahrnuje nasazení každé služby na vlastní server, ať už virtuální, nebo fyzický. Tento přístup zjednodušuje ladění a izolaci služeb, ale může vést k nedostatečnému využití zdrojů a vyšším nákladům.
- **Více služeb na jednoho hostitele** - Nasazení více služeb na jednom serveru maximalizuje využití zdrojů a snižuje náklady. Vyžaduje však pečlivou správu, aby nedocházelo ke konfliktům a aby služby při běhu vzájemně nekolidovaly. Zároveň musí být splněn předpoklad, že množina takto nasazených služeb vyžaduje obdobné prostředí pro nasazení.
- **Instance služby na kontejner** - Moderní nasazení mikroslužeb často používají kontejnery (například Docker) pro umístění jednotlivých služeb. Kontejnery poskytují odlehčené a oddělené prostředí pro každou službu, zjednodušují nasazení, škálování a zajišťují, že každá služba má splněny své závislosti bez konfliktů s jinými službami.

2.6.2 Cloud-native nasazení

Microservice architektura je obzvláště vhodná pro nativní cloudová prostředí, která podporují jejich dynamickou povahu. [23] Příklady strategií cloud-native nasazení zahrnují:

- **Infrastruktura jako služba** - Tento styl nasazení zprostředkovává v cloudu potřebnou infrastrukturu jako je výpočetní výkon, úložiště a síťové služby. Následně lze na infrastrukturu nainstalovat nástroje orchestrace jako například Kubernetes pro nasazení kontejnerizovaných služeb. Kubernetes následně řídí životní cyklus služeb od nasazení až po ukončení. Kubernetes se stará o škálování, vyrovnávání zátěže a řeší obnovu po pádu. Využití kontejnerů a orchestrace umožňuje rychlé a spolehlivé nasazení aplikace v microservice architektuře nativně na cloudu při maximální kontrole nad prostředím.

- **Mikroslužby na platformě jako služba** - Platform as a Service (dále PaaS) je typ nasazení poskytovující prostředí, kde lze mikroslužby snadno nasadit, škálovat a spravovat bez nutnosti starat se o základní infrastrukturu. Poskytovatel cloudu je zodpovědný za provoz a správu platformy, což uživatelům umožňuje soustředit se na vývoj aplikací.

- **Serverless** - Bezserverové výpočetní modely umožňují nasazení mikroslužeb jako funkcí (Function as a Service, dále FaaS), které se spouštějí v reakci na události. Poskytovatel cloudu spravuje prostředí, v němž jsou nasazeny a dodává rozhraní pro jejich konfiguraci. Tento model je prezentován jako vysoce škálovatelný a nákladově efektivní, protože zdroje jsou spotřebovávány pouze za běhu aplikačních funkcí.

3 MONITOROVÁNÍ APLIKACE

Monitorování aplikace je proces získávání zpětné vazby. Představuje klíčový aspekt vývoje a provozu software. [24] Umožňuje sledovat stav, výkon a celkové chování aplikace. Zahrnuje shromažďování, analýzu a interpretaci různých typů dat a informací, které zajišťují hladký a efektivní chod aplikací. Umožňuje rychle identifikovat a řešit problémy vzniklé za provozu.

3.1 Cíle monitorování

Cílem monitorování v kontextu mikroslužeb je poskytnout informace v klíčových oblastech aplikace. Základní informací je dostupnost aplikace, případně jednotlivých mikroslužeb. Ta zajišťuje, že lze rychle identifikovat vznik problému a minimalizovat dobu nedostupnosti systému. Výkonnostní metriky, jako je odezva, propustnost a chybovost, pomáhají pochopit, jak aplikace funguje za normálních podmínek i při zátěži. [24] Rovněž poskytují zpětnou vazbu o dopadu různých strategií nasazení a sestavení na výkon. Zahrnují údaje o míře úspěšnosti nasazení a problémy vyplývající z nových nasazení pomocí analýzy chování služeb v produkčním prostředí.

3.2 Druhy dat

Monitorovací data hrají zásadní roli při údržbě a optimalizaci moderních softwarových systémů. Sběrem, analýzou a interpretací různých typů dat lze získat cenné informace o výkonu, stavu a celkovém chování aplikace. Následující část kategorizuje tři základní typy monitorovacích dat: logy, traces a metriky. [25]

3.2.1 Logy

Logy jsou chronologické záznamy o událostech, ke kterým dochází v rámci aplikace. Pomáhají určit hlavní příčiny konkrétních problémů nebo odhalit vzory rozsáhlých chyb. Logy generuje jak aplikace, tak dle použité technologie i prostředí jenž aplikaci spravuje. Ať se jedná o OS nebo běhové prostředí. Logy mohou nabývat různých struktur. Nejjednodušší forma je obyčejný textový řetězec. Některé nástroje však podporují i strukturované logy. To jsou datové záznamy nabývající podoby typicky JSON nebo XML formátu. Je běžné, že konkrétní záznam logu poskytuje pouze část informace o systémové události. A tedy, že celá událost sestává z více záznamů. [25] Logy typicky obsahují data o systémových aktivitách, chybách, zprávách, změnách konfigurace a síťových požadavcích. Analýzou logů mohou vývojáři a správci systému porozumět kontextu aplikace a zajistit soulad s očekávaným chováním. [25]

3.2.2 Traces

Traces trasují cestu požadavku, při průchodu různými částmi distribuované aplikace. Každá trace se skládá z jednoho nebo více segmentů, které zaznamenávají cestu a latenci požadavku napříč různými službami a zdroji. Sledování je zvláště důležité v architektuře mikroslužeb, kde jedna transakce může zahrnovat více volně propojených služeb. Traces poskytují přehled o výkonu a chování jednotlivých služeb, ale také aplikace jako celku. Pomáhají identifikovat úzká místa aplikace a problémy s latencí. [26] Zároveň umožňují pochopit vztahy a závislosti mezi službami, což podporuje efektivnější ladění a optimalizaci aplikace. Obvykle jsou reprezentovány stromovou strukturou, jenž obsahuje hierarchii segmentů. Každý segment udržuje informace o době odezvy, latenci, chybách a dalších metrikách. [26]

3.2.3 Metriky

Metrika představuje konkrétní číslo volitelně doplněné o značky (tagy) sloužící k identifikaci a seskupování. [25] Jedná se o kvantitativní údaj, který reprezentuje informaci o stavu aplikace. Metriky jsou klíčovým prvkem monitorování, protože poskytují objektivní a měřitelné informace o výkonu a chování aplikace. Typicky jsou shromažďovány v pravidelných intervalech v reálném čase. Běžná data, jenž metriky reprezentují, jsou například využití systémových prostředků (CPU, paměť, I/O, ...), doba odezvy, chybovost anebo propustnost. Sledování těchto metrik pomáhá při ladění výkonu a správě aplikace.

3.3 Monitorovací nástroje

Monitorovací data vyžadují nástroje, které umožňují jejich shromažďování, analýzu a vizualizaci. Obvykle se skládají z několika komponent včetně agentů, kolektorů, databází, vizualizačních nástrojů a nástrojů pro analýzu dat. [24] Různé nástroje mohou poskytovat komplexní řešení nebo pouze část funkcionality vyžadované za účelem monitorování systému.

3.4 Sběr dat

Efektivita monitorování aplikací do značné míry závisí na schopnosti shromažďovat relevantní data z různých zdrojů a na schopnosti zprostředkovat tato data do monitorovacích nástrojů. Kolektory jsou nástroje nebo agenti, kteří shromažďují data z různých zdrojů v rámci aplikace a jejího prostředí. [27] Mohou být nasazeny jako součást infrastruktury aplikace nebo mohou být provozovány jako externí služby. Kolektory jsou zodpovědné za shromažďování logů, traces a metrik. Dále zprostředkovávají předávání

těchto dat do monitorovacích nástrojů, kde je následně možné provést analýzu a vizualizaci. Efektivní sběr dat je nezbytný pro monitorování v reálném čase a pro zajištění toho, aby shromážděná data nezaznamenala zkreslený stav a výkon aplikace. Příklad velmi populárního univerzálního kolektoru je OpenTelemetry. Tento nástroj mimo jiné definuje prostor monitorovacích technologií, protokoly a standardy využívané pro sběr a propagaci dat nezávisle na využitých nástrojích. [27]

3.5 Analýza a interpretace

V oblasti monitorování systému je sběr dat pouze prvním krokem. Skutečná hodnota spočívá v tom, jak jsou tato data analyzována a interpretována. Analýza a interpretace transformují nezpracovaná data na praktické poznatky, které umožňují porozumět nejen tomu, co se děje v aplikacích, ale také tomu, proč k těmto událostem dochází. [25] Tyto procesy jsou úzce propojeny. Vizualizace přibližuje pochopení komplexních dat a pomáhá uživatelům monitorovacích nástrojů rozpoznat trendy a anomálie na první pohled. Pokročilé analytické techniky poskytují hlubší porozumění dat, odhalují základní vzorce a předpovídají budoucí trendy, kterými podporují strategické rozhodování. Společně umožňují reagovat na aktuální stav aplikace, proaktivně ji spravovat a optimalizovat budoucí výkon. Grafickým znázorněním datových toků lze snadněji odhalit trendy a vzorce, které nemusí být patrné ze samotných nezpracovaných dat. Vizualizace mohou mít různé formáty, následující jsou příklady běžně používaných vizualizačních komponent:

- **Grafy** - Graficky reprezentují data vůči času nebo jiným škálám. Různými typy jako např. spojnicové grafy, sloupcové grafy a bodové grafy, které mohou zobrazovat změny v čase, distribuci a korelaci.
- **Tabulky** - Prezentují nezpracovaná data zarovnaná do sloupců pro přímé srovnání.
- **Řídicí panely** - Integrují více vizualizací do jediného rozhraní a nabízejí detailní pohled na výkon a stav systému.
- **Heatmapy** - Zobrazují složité vztahy a toky mezi komponentami systému. Barevné škály ukazují intenzitu a frekvenci událostí.

Použitím a kombinací vizualizačních komponent vzniká unikátní pohled na dostupná monitorovací data. Tím je umožněno rychle identifikovat kritická místa systému, jako jsou bottlenecks (úzká místa výkonu) a řešit potenciální problémy dříve, než ovlivní stabilitu systému nebo uživatelský dojem z aplikace. [28] Konečným cílem analýzy a vizualizace dat je podpora informovaného rozhodování. Interpretovaná data poskytují

užitečné poznatky, na základě kterých lze provádět strategická rozhodnutí, optimalizovat výkon a stabilitu systému. [25] Následující seznam uvádí vybrané oblasti, ve kterých můžou informace získané monitorováním aplikací být využity:

- **Přidělování zdrojů** - Úprava přidělování zdrojů na základě informací o výkonu. Za účelem optimalizace zdrojů a výkonu probíhá například škálování aplikace. V reakci na očekávanou zátěž je možné dynamicky přidělit nebo odebrat zdroje.
- **Optimalizace výkonu** - Identifikace a řešení překážek výkonu. Nalezením úzkých míst v aplikaci lze optimalizovat software (dále SW) s cílem zlepšit odezvu a spokojenost uživatelů.
- **Zabezpečení** - Rozpoznání vzorců indikujících bezpečnostní hrozby. Neočekávaným počtem požadavků nebo jiným využitím zdrojů lze identifikovat probíhající útok na aplikaci.
- **Vylepšení služeb** - Využití dat o interakci uživatelů s aplikací. Data popipsující chování uživatelů podporují kroky vedoucí k vylepšení aplikace dle očekávání zákazníků.

3.6 Implementace monitorování

Implementace monitorování aplikace zahrnuje několik kroků, včetně definice klíčových metrik, výběru monitorovacích nástrojů, nasazení kolektorů a vizualizace dat. Zároveň by měly být vytvořeny procesy a postupy pro využití dat a řešení problémů, které byly identifikovány prostřednictvím monitorování. [27] Obecně implementace monitorování zahrnuje následující kroky:

1. **Sběr dat v monitorovaných službách** - Implementace sběru dat zahrnuje inkorporaci funkcionality monitorování a zprostředkování dat v rámci předem definovaného rozhraní. Sběr je realizován zpravidla sérií čítačů a zapisovačů, které jsou využívány k získávání dat z různých zdrojů. Takto sbíraná data jsou kategorizována a označena pro identifikaci. Realizace monitorování je zajištěna buďto použitím existujících implementací v rámci SW knihoven nebo vytvořením vlastní implementace dle potřeb aplikace a monitorovacích protokolů.
2. **Nasazení služeb pro správu a sběr dat** - Je zajištěno pomocí nasazení nástrojů, které jsou schopny zprostředkovat sběr a distribuci telemetrických dat z různých zdrojů. Zároveň mohou také zprostředkovat jejich zpracování a zobrazení. Klíčový je výběr nástrojů s ohledem na kompatibilní rozhraní pro sběr a následnou distribuci dat k vizualizaci. Splněním tohoto je dosaženo, že data mohou být zpracována, uložena a dále zprostředkována.

3. **Vizualizace dat** - Vizualizace dat je implementována nasazením nástrojů, které jsou schopny zobrazit data z dostupných zdrojů v uživatelsky přívětivé podobě. Formát připojení na datové zdroje s monitorovacími daty je definován protokoly služeb spravujících tato data. Vizualiace konkrétních dat je předmětem implementace grafických komponent za pomocí dostupných rozhraní.

Konfigurace monitorování v rámci aplikace obecně zahrnuje zmapování interakcí mezi monitorovanými komponentami a monitorovacími nástroji. To zahrnuje určení, které metriky, logy a traces jsou relevantní na základě architektury aplikace a doménových požadavků. Konfigurace musí zajistit, že shromážděná data budou relevatní a vyvarovat se nadměrné granularitě, která může vést k přetížení systému. [27] Obvykle tento proces zahrnuje nastavení agentů či integrací v rámci aplikace nebo služeb, které efektivně sbírají data a přenáší je dále do nástrojů monitorovacího systému. Konají tak aniž by došlo k narušení výkonu aplikace. Komunikace mezi aplikačními komponentami a monitorovacími nástroji využívá síťové protokoly a metody bezpečného přenosu dat. Proces konfigurace může dále zahrnovat nastavení hraničních hodnot pro výstrahy, definování pravidel retence dat a nastavení parametrů pro automatické reakce na určité typy událostí. Tyto prvky pomáhají udržovat celkový stav a výkon aplikace. [27] Implementace tohoto přístupu zajišťuje, že monitorovací systém poskytuje užitečné informace v souladu s doménovými požadavky a poskytuje komplexní bázi dat pro analýzu a optimalizaci systému.

II. PRAKTICKÁ ČÁST

4 TESTOVACÍ APLIKACE

Praktická část této práce se zaměřuje na vytvoření testovací aplikace postavené na microservice architektuře. Cílem je vytvořit, otestovat a analyzovat služby využívající JIT a nativní AOT kompilace. Rozsah funkcionality a chování aplikace je definováno množinou funkčních a nefunkčních požadavků. Dále jsou vybrány konkrétní technologie a nástroje, jenž v aplikaci doplní .NET služby. Jejich účelem je zprostředkování platformy monitorování, dodání perzistence, hostování webového rozhraní a zprostředkování testování. Samotný postup testování je definován metodikou, která zahrnuje také definici hypotéz a je podrobně popsána v následující kapitole. Tyto hypotézy jsou následně ověřeny v analytické sekci práce. Ověření probíhá v rámci experimentů, které jsou provedeny na testovací aplikaci. Data z testů jsou zprostředkovány pro analýzu a vyhodnocení. Aplikace je nasazena v kontejnerizovaném prostředí a vytvořena s ohledem na rozšiřitelnost pro otestování konkrétní doménové problematiky.

4.1 Požadavky na SW

Na aplikaci jsou pro splnění účelu analýzy vývoje, výstupu a výkonu služeb kladeny přímé i nepřímé požadavky. Je klíčové navrhnout řešení, vybrat technologie a provést implementaci včetně konfigurace s ohledem na tyto požadavky. Následující část této sekce je kategorizována do funkčních a nefunkčních požadavků na SW.

4.1.1 Funkční požadavky

Funkční požadavky definují chování, funkce a vlastnosti, které musí aplikace poskytovat. Přímo souvisejí s doménovými požadavky a zahrnují specifikace, jako je zpracování dat, provádění výpočtů nebo podpora konkrétních procesů. Funkční požadavky popisují očekávané operace systému, včetně vstupů, chování a výstupů. Jsou tak klíčové pro vývoj a testování. V případě testovací aplikace jsou požadavky se zacílením na splnění testovacích scénářů a minimalistickou simulaci scénářů. Následující seznam funkčních požadavků byl definován pro testovací aplikaci.

- **Healthchecks** - Služby musí implementovat na REST API healtcheck endpoint, který bude poskytovat informace o stavu služby. Endpoint musí být dostupný na standardní adrese */health*. Návratová hodnota bude triviálně formou řetězce *Healthy* a vracet HTTP Code 200 v případě, že je služba dostupná. Dostupnost je definována schopností služby přijímat požadavky.
- **SwaggerUI** - Pro vizualizaci a testování REST API služeb musí být implementováno grafické rozhraní SwaggerUI. SwaggerUI musí být dostupné na standardní

adrese */swagger* a musí zobrazovat dostupné endpointy a umožňovat jejich testování. SwaggerUI je implementováno pouze v konfiguraci JIT kompilace a režimu Debug.

- **Perzistence souborů** - Aplikace musí umožňovat ukládání libovolného souboru do perzistentního úložiště. Soubor musí být uložen do PostgreSQL databáze a musí být možné ho následně stáhnout. Pro ukládání a čtení souborů musí být implementováno rozhraní REST API. Specificky pro čtení souborů musí být implementováno i gRPC rozhraní.

- **Generování signálů** - Aplikace musí být schopna generovat náhodné signály. Signál musí obsahovat název, jednotku a hodnotu. Pro získání generovaných signálů musí být implementováno rozhraní REST API.

- **Výpočet n-tého Fibonacciho čísla** - Je požadováno, aby aplikace poskytovala funkcionalitu výpočtu Fibonacciho čísla rekurzivní metodou. Tato neefektivní metoda má za účel vytvořit zátěž na systém. Pro volání výpoču a získání výsledku musí být implementováno rozhraní rozhraní API.

- **Asynchronní komunikace** - Aplikace musí být schopna asynchronně zpracovávat data z jiných služeb. To zahrnuje vyvolání události a její následnou konzumaci vzorem publish - subscribe. Pro implementaci asynchronní komunikace bude využito RabbitMQ. Samotné vyvolání události musí být k dispozici pomocí požadavku na REST API.

- **Sběr telemetrických dat z .NET služeb** - Aplikace musí být schopna sbírat a ukládat telemetrická data z .NET služeb. To zahrnuje metriky, logy a traces. Data musí být dostupná v reálném čase. Veškerá data budou strukturována dle zásad OpenTelemetry a sbírány na gRPC rozhraní této služby. Sbíraná data budou určena podstatou funkcionality služby a doplněna o množinu dostupných a relevantních metrik zprostředkovaných knihovnami OpenTelemetry.

- **Monitorování kontejnerů a systému** - Aplikace musí být schopna sbírat a vizualizovat data o výkonu, škálovatelnosti kontejnerů a hostitelském systému. To zahrnuje sběr a vizualizaci výkonnostních metrik. Data musí být dostupná v reálném čase a musí být perzistentně ukládána.

- **Testování scénářů** - Aplikace musí být schopna provádět testování scénářů, které simulují běh systému a zátěž na mikroslužby. Testovací scénáře musí být jednoduše vytvořitelné pomocí skriptů. Spouštění scénářů nad aplikací musí být možno více způsoby, napřímo pomocí nástroje nativně běžícího na hostitelském

sytému, ale také kontejnerizovaným nasazením nástroje. Testovací scénáře musí být konfigurovatelné a spustitelné v manuálním a automatizovaném režimu.

- **Vizualizace dat** - V rámci aplikace musí být dostupné grafické rozhraní pro vizualizaci metrik a testovacích dat. Vizualizace musí být dostupná v reálném čase a umožnit zobrazení historických dat. Je nutné, aby aplikace podporovala seskupení a filtraci dat podle druhu, značek a času. Zároveň je požadováno, aby uživatelé mohli jednoduše připojit různé zdroje dat a vytvářet vizualizace ve vlastní režii. Přístup k funkcionalitě vizualizace musí být řešen přes webové rozhraní.
- **Směrování** - Přístup k aplikaci bude řešen pomocí reverzní proxy. Ta bude mít vystavené rozhraní z orchestračního nástroje a její indexovací stránka bude odkazovat na vizualizační nástroj monitorovacích dat.
- **Konfigurace aplikace** - V rámci aplikace musí být možnost konfigurovat chování nasazených služeb. To se týká jak konfigurace komunikace mezi službami, tak i konfigurace monitorovacích nástrojů. Nastavení musí být uloženo v konfiguračních souborech ve standardním formátu dle konvencí dané služby nebo nástroje.

4.1.2 Nefunkční požadavky

Nefunkční požadavky specifikují celkové vlastnosti systému. Definují atributy kvality, které musí systém splňovat. Nefunkční požadavky mohou zahrnovat omezení týkající se návrhu a implementace aplikace, jako jsou bezpečnostní standardy, soulad s právními a regulačními směrnicemi, doba odezvy při zpracování dat, kapacita pro souběžné uživatele, integrita dat a mechanismy převzetí služeb při selhání. Mají zásadní význam pro zajištění životaschopnosti a efektivity aplikace v provozním prostředí. Ovlivňují celkový uživatelský dojem, výkonnost systému a splnění regulačních podmínek. Následující seznam nefunkčních požadavků byl definován pro testovací aplikaci.

- **Použitelnost** - Aplikace musí být snadno použitelná a přístupná pro uživatele. To zahrnuje snadnou konfiguraci a nasazení aplikace na specifickém HW a OS. Aplikace musí být dostupná na webovém rozhraní a standardních portech.
- **Udržitelnost** - Aplikace musí být udržitelná a snadno rozšiřitelná. To zahrnuje dodržení praktik čistého kódu a vhodných návrhových vzorů. Implementace služeb musí být založena na principu SOLID a Don't Repeat Yourself (dále DRY). Dodržování SOLID principu zajišťuje testovatelnost, rozšiřitelnost a dlouhodobou udržitelnost aplikace. Zatímco použitím DRY principu je zabráněno tvorbě duplicitního kódu. Vytvořený kód, konfigurace a skripty musí být řádně dokumentovány.

- **Testovatelnost** - Aplikace musí být snadno testovatelná. To zahrnuje zprostředkování nástrojů a API pro možnost definice a konfigurace vlastních testovacích scénářů. Testování musí být automatizovatelné a poskytovat možnost perzistentního ukládání výsledků testů.
- **Přívětivost** - Aplikace musí být přívětivá pro uživatele. To zahrnuje snadnou navigaci, přehlednost a intuitivní ovládání. Vizuální stránka aplikace musí být jasná a přehledná.

4.2 Požadavky na HW

Hardware, na kterém bude aplikace provozována, musí výkonnostně dostačovat pro běh testovacích scénářů a sběr a vizualizaci dat. Týká se to primárně počtu jader, velikosti paměti a rychlosti diskového I/O. Provozované služby mají určitou základní režii, která se musí brát v potaz.

4.3 Organizace a správa zdrojů

Pro správu souborů práce byl zvolen verzovací systém (Source Control Management, dále SCM) Git. Git je open-source nástroj, který umožňuje spravovat a sdílet soubory. Byl zvolen pro svou schopnost vést historii v rámci větví a s ohledem na dostupná serverová úložiště. Za účelem jednoduché organizace souborů bylo zvoleno řešení monorepozitáře. Monorepozitář je repozitář, který obsahuje veškeré soubory projektu, ale také relevantní dokumentaci, obrázky, podpůrné nástroje a zdrojové soubory diplomové práce. Následující struktura adresářů byla zvolena pro organizaci souborů.

- **Documentation** - Zahrnuje podpůrné soubory aplikační dokumentace.
- **Source** - Obsahuje zdrojové soubory aplikace, nasazení a konfigurace.
- **Thesis** - Uchovává text diplomové práce, zdrojové soubory pro vytvoření práce v LaTeX a práci samotnou ve formátu pdf.

Pro sdílení veškerých souborů souvisejících s prací a jejich sdílení byl vybrán GitHub, jakožto server pro hostování repozitáře. GitHub je platforma pro verzování souborů a projektů. Navíc poskytuje dodatečné funkce jako je CI/CD, správa dokumentace a další. Repozitář projektu je veden jako veřejný s licencí MIT a je dostupný na adrese `https://github.com/DonasNave/MasterThesis`.

4.4 Návrh a implementace testovacích služeb

Následující pasáž se zabírá návrhem a implementací testovacích služeb, které budou využity pro analýzu vývoje a výkonu jednotlivých kompilací AOT a JIT v rámci .NET.

Služby jsou implementovány jako mikroslužby a podporují kontejnerizované nasazení v microservice architektuře. Každá služba reprezentuje jednu dílčí funkcionalitu a má definované rozhraní pro komunikaci s ostatními službami. Pro implementaci služeb byla vybrána z podstaty práce technologie .NET, konkrétně jazyk C#. Verze frameworku byla zvolena .NET 8.0 (konkrétně 8.0.4) jakožto jediná verze oficiálně podporující nativní AOT kompilaci pro framework ASP.NET. Jazyk C# je použit ve verzi 12.0.

4.4.1 Architektura

Architektura testovacích služeb byla vytvořena s cílem minimalisticky simulovat scénáře v aplikaci. Pro splnění funkčních požadavků bylo zvoleno následující rozdělení zodpovědnosti .NET služeb:

- **SRS - Signal reading service** - Simuluje roli čtecího zařízení. Generuje signály a poskytuje je ostatním službám.
- **FUS - File Upload Service** - Zprostředkovává datové perzistentní zapisovací zařízení. Čte nebo zapisuje soubory do PostgreSQL databáze.
- **BPS - Business Processing Service** - Reaguje na události publikované v RabbitMQ, do kterého je napojena jako subscriber. Provádí výpočet Fibonacciho čísla.
- **EPS - Event Publishing Service** - Slouží k vyvolání události, která je následně zpracována jinými službami. Je přihlášena do RabbitMQ jako publisher.

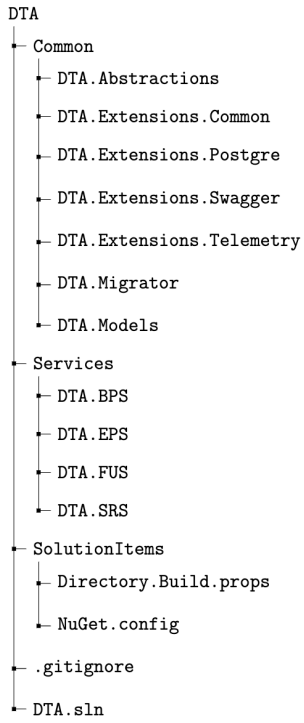
Nativního AOT kompilace kódu je deklarována použitím atributu *PublishAoT* v projektovém souboru. Za účelem zajištění co největší podobnosti služeb zacílených na AOT a JIT kompilaci, bude využito vymezení konstantních hodnot v rámci projektu. Konstanty *JIT* a *AOT* budou využity pro rozlišení chování služeb v rámci obou kompilačních verzích. S použitím direktiv kompilátoru a zmíněných konstant bude v nutných případech docíleno rozdílného volání API při snaze zachovat totožnou funkcionalitu.

4.4.2 Očekávání vývojového procesu

Na základě podporované funkcionality, tak jak je definována týmem .NET a popsána v rámci rešerše, je očekáváno, že vývojový proces bude probíhat bez výrazných problémů a bude možné vytvořit služby, které budou schopny zvládnout definované funkční a nefunkční požadavky. Podpora třetích stran byla předem prozkoumána v rámci dostupných dokumentací vybraných knihoven .NET. Konkrétní podoba a rozsah této podpory budou plně ověřitelné až po implementaci a otestování služeb.

4.4.3 Organizace souborů

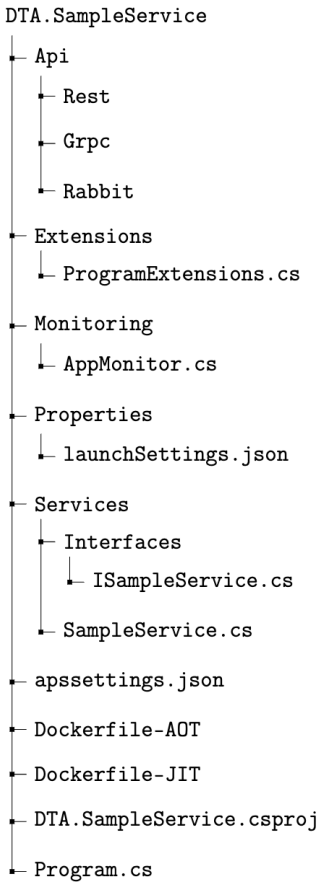
Organizace zdrojových souborů služeb, knihoven a pomocných souborů je řešena v rámci hlavního adresáře *DTA* obsahujícího .NET solution soubor, pomocné soubory a solution složky s konkrétními projekty služeb a knihoven. Následující graf popisuje strukturu adresáře projektu.



Obrázek 4.1 Struktura adresáře projektu

Každá z vyvinutých služeb využívá konkrétní .NET SDK *Microsoft.NET.Sdk.Web*, které umožňuje využít třídu *WebApplication* pro registraci a konfiguraci funkcionality služby a zároveň poskytuje konfigurovatelný Kestrel server pro běh programu. Za účelem zajištění jednotného přístupu k logování, metrikám a konfiguraci byly vytvořeny společné knihovny, které jsou využity ve všech službách. Následující graf vizualizuje ukázkou strukturu adresáře služby. Následný seznam popisuje vybrané složky a soubory.

- **Api** - Obsahuje implementaci rozhraní služby.
- **Extensions** - Implementuje extension metody specifické pro doménu služby.
- **Monitoring** - Obsahuje statickou třídu, která drží reference na počítadla metrik.
- **Service** - Ve složce jsou implementovány služby, které provádějí doménovou logiku služby.



Obrázek 4.2 Struktura adresáře služby

- **Properties** - Drží konfiguraci pro spuštění služby.
- **Program.cs** - Obsahuje definici a konfiguraci služby, včetně jejího vstupního bodu.
- **appsettings.json** - Konfigurační soubor služby.
- **Dockerfile-AOT** - Soubor pro tvorbu Docker obrazu pro AOT kompilaci.
- **Dockerfile-JIT** - Soubor pro tvorbu Docker obrazu pro JIT kompilaci.

Součástí řešení je společná konfigurace, která je využita ve všech službách. Ta je řešena jedna na úrovni solution souboru a Directory.Build.props souboru. Týká se jednotné distribuce projektových atributů pro verzi, kompatibilitu s AOT, vynucení konkrétních pravidel pro kód a analyzéry.

4.4.4 Knihovny třetích stran

Pro implementaci funkcionality aplikace byly využity následující knihovny třetích stran:

- **Npgsql** - Npgsql je open-source ADO.NET provider pro PostgreSQL, který umožňuje komunikaci s PostgreSQL databází. Npgsql poskytuje základní balíček funkcí pro vytvoření připojení na základě standardizovaného řetězce pro připojení. Tento balíček sice není plně kompatibilní s AOT kompilací, funkce které jsou využity v rámci aplikace jsou avšak kompatibilní.
- **Dapper** - ORM knihovna pro .NET, která umožňuje mapovat databázové struktury na C# objekty a provádět dotazy na databázi. Dapper.AOT je dílčí knihovna, která umožňuje provádět dotazy na databázi s podporou AOT kompilace. Toho je zajištěno tím, že Dapper.AOT generuje kód pro mapování objektů v době kompilace. Využívá k tomu interceptorů a generátorů pro zachování totožného API jak v případě kódu pro JIT kompilaci. Samotný balíček Dapper.AOT obsahuje další knihovnu Dapper.Advisor, která pomáhá s analýzou zdrojového kódu včetně dotazů na databázi.
- **OpenTelemetry** - OpenTelemetry zprostředkovává množinu knihoven pro sběr, zpracování a export telemetrických dat. V rámci dodaného API je možno registrovat vlastní metriky, logy a traces, ale také nastavit export vybraných dat systémových knihoven a třetích stran. To se týká vybraných knihoven, které zprostředkovávají vlastní implementaci metrik OpenTelemetry.
- **Grpc** - Knihovny pro implementaci komunikace pomocí protokolu HTTP/2 a gRPC. Konkrétně jsou využity Grpc.AspNetCore v případě serveru, Grpc.Net.Client pro klienta a Google.Protobuf s Grpc.Tools pro generování modelů v přístupu code first.
- **RabbitMQ** - Asynchronní komunikace a implementace publish subscribe vzoru je umožněna knihovnou RabbitMQ.Client. S její pomocí aplikace komunikují s brokerem, vytváří fronty, dochází k přihlášení nebo odběru zpráv a jejich publikování.
- **Swagger** - Grafické rozhraní pro vizualizaci a testování REST API služeb. Swagger je využit pouze v kombinaci konfigurací *JIT Debug*. K tomuto účelou jsou využity knihovny Swashbuckle.AspNetCore a Microsoft.AspNetCore.OpenApi.

4.4.5 Společné knihovny

V rámci zjednodušení tvorby služeb, jednotné implementace a konfigurace, ale také z důvodu zajištění některé základní klíčové funkcionality, byly vytvořeny společné knihovny. Tyto knihovny obsahují společné třídy, rozhraní a konfigurace, které jsou použity ve všech službách. Následující výčet popisuje oblasti funkcionality, které jsou zprostředkovány společnými knihovnami.

- **Perzistence** - Pro implementaci perzistence byla vytvořena pomocná knihovna DTA.Extensions.Postgres, která poskytuje pomocnou funkcionalitu pro zajištění existence databáze pro službu, dle konfigurace v řetězci pro připojení.
- **Migrace** - Zajištění migrace databáze bylo implementováno po vlastní ose minimalistickým migrátorem v knihovně DTA.Migrator. Tato knihovna poskytuje základní funkcionalitu pro vytvoření databáze, vytvoření tabulek a indexů, ale také zajištění migrace dat a verzování změn.
- **Telemetrie** - Knihovna DTA.Extensions.Telemetry zprostředkovává extensions metody pro jednotnou a jednoduchou registraci sběru a export telemetrických dat napříč službami.
- **Modely** - Knihovna DTA.Models obsahuje společné modely, které jsou využity ve službách. Je tím docílena dostupnost datových struktur a rozhraní aplikace napříč všemi službami.
- **Obecná funkcionalita** - Za účelem sjednocení funkcionality využité napříč všemi službami jsou v rámci DTA.Extensions.Common knihovny implementovány specifické extension metody. Poskytnuta je funkcionalita pro extrakci názvů a verzí z metadat služby.

4.4.6 SRS - Signal reading service

Za účelem simulace funkce čtecího zařízení byla vytvořena služba SRS. Tato služba poskytuje základní rozhraní pro získání dat signálu včetně jednotek formou REST API. Pro zjednodušení implementace není využito čtení dat ze skutečného zdroje, ale jsou generována náhodná data. Načež data jsou následně poskytována se zdržením simulujícím čtení dat ze vzdáleného zdroje. Služba poskytuje následující rozhraní

- **GET /api/signals/{int:amount}** - Vygeneruje zadané množství náhodných signálů

4.4.7 FUS - File Upload Service

Služba v systému hraje roli rozhraní k persistentnímu uložišti, v rámci kterého čte a zapisuje data. Jakožto úložiště je využito PostgreSQL databáze. Služba využívá vlastní databázovou instanci a spravuje vlastní tabulky pomocí migrací definovaných SQL skripty. Pro přístup k perzistence dat je využito knihovny knihovny Dapper, která umožňuje mapování databázových struktur na C# objekty a vytváření a provádění dotazů na databázi. SRS poskytuje rozhraní formou REST API pro zápis a čtení dat. Daty je myšlen libovolný soubor v libovolném formátu. Samotná podstata nahraných

dat není pro službu důležitá, ale to že jsou uložena do databáze. Za účelem sehrání testovacích scénářů poskytuje služba také gRPC rozhraní, které je zajištěno na dedikovaném portu. V rámci gRPC komunikace slouží FUS jako server, který zpracovává volání vzdálené procudery. Služba poskytuje následující rozhraní:

- **GET /api/file/download/{int:id}** - Stáhne soubor podle zadaného ID.
- **POST /api/file/upload** - Nahraje soubor do systému. Soubor je předán jako multipart/form-data.
- **gRPC Operation FileServer.GetFile** - Stáhne soubor podle zadaného objektu s ID.

4.4.8 BPS - Business Processing Service

Pro splnění role vytvoření zátěže na systém je vytvořena BPS. Tato služba získává data pomocí volání gRPC, konzumuje jako subsciber událost a provádí náročnou výpočetní operaci, kterou simuluje obtížnou doménovou operaci. Konkrétně implementaváno je neefiktivní rekurzivní výpočet zadaného čísla Fibonacciho posloupnosti. BPS se po spuštění přihlašuje k odběru zpráv na předem definovaný kanál *simulated* na službe RabbitMQ. Po získání zprávy volá vzdálenou proceduru nad FUS. Následně provádí výpočet 40-tého Fibonacciho čísla. Tato hodnota je pevná s ohledem na její jediné využití a zacílení pro scecifický testovací scénář. Služba poskytuje následující rozhraní:

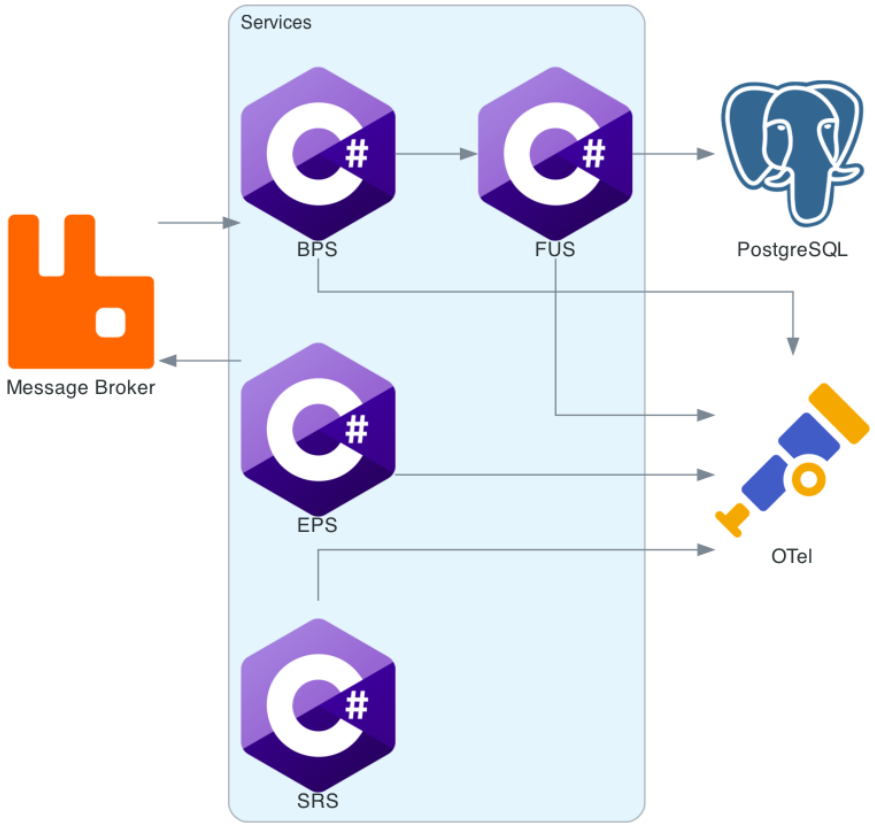
- **GET /api/processFibonacci/{int:degree}** - Vypočítá číslo z Fibonacciho posloupnosti na zadané pozici náročným rekurzivním způsobem.
- **Event subscribed: <queue-name>_simulated** - Přihlášení k odběru zpráv v rámci kanálu na službě RabbitMQ.

4.4.9 EPS - Event Publishing Service

Jednoduchá služba umožňující vyvolat událost ve frontě a docílit spuštění dodatečných operací v aplikaci. V systému simuluje roli uživatele vyvolávajícího událost. Služba poskytuje následující rozhraní:

- **GET api/simulateEvent/{int:id}** - Vyvolá simulovanou událost s daným ID.
- **Event published: <queue-name>_simulated** - Vyvolá údalost se zprávou obsahující identifikator na konfigurovaném kanálu do služby RabbitMQ.

Následující diagram znázorňuje přímé závislosti testovacích služeb na další nástroje.



Obrázek 4.3 Diagram .NET služeb a závislých služeb

4.5 Monitorování aplikace

Za účelem monitorování aplikace byla vybraná množina nástrojů. Tyto nástroje umožňují sběr, perzistenci a vizualizaci metrik, traces a logů. Klíčové bylo zajistit možnost sledovat dění uvnitř aplikace, ale i v rámci hostitelského systému. Následující pasáž se zabývá výběrem a implementací monitorovacích nástrojů.

4.5.1 Grafana observability stack

Pro monitorování aplikace byl zvolen Grafana Observability stack pro jeho pokrytí komplexní škály monitorovacích dat. Zahrnuje specifické nástroje pro sběr, vizualizaci a analýzu dat. Zprostředkovává jednoduchou možnost propojení dílčích nástrojů a konfiguraci datových zdrojů. V neposlední řadě poskytuje rozsáhlé možnosti vizualizace. Následující nástroje jsou součástí Grafana Observability stacku.

- **Grafana** - Grafana je open source webová aplikace pro analýzu a interaktivní vizualizaci dat. Poskytuje možnost sestavit dashboard z komponent jako jsou grafy, tabulky a další. Jedná se o velmi populární technologii v doménách serverové infrastruktury a monitorování. Grafana umožňuje sjednotit monitorovací služby a zobrazit data v reálném čase. Podporuje širokou škálu datových zdrojů, jako

jsou Prometheus, InfluxDB, Tempo, Loki nebo PostgreSQL, což umožňuje jednoduchou konfiguraci a připojení monitorovacích dat aplikace. Zároveň nativně podporuje datový zdroj, kde ukládá data testovací nástroj aplikace. Kombinací dat z různých zdrojů umožňuje vytvářet komplexní pohled na celý systém.

- **Prometheus** - Open-source monitorovací systém. Shromažďuje a ukládá metriky jako time-series data a umožňuje se na ně dotazovat pomocí vlastního výkonného jazyka PromQL. Jeho architektura podporuje více modelů získávání dat, což je využito při napojení více zdrojů v aplikaci. Dále umožňuje přímé stahování metrik z cílových služeb nebo kolektorů, odesílání metrik přes gateway a zprostředkování notifikací.

- **Loki** - Škálovatelný agregátor logů. Na rozdíl od obdobných systémů pro agregaci logů, jenž indexují všechna data, Loki indexuje pouze metadata, přičemž ukládá celá data logu efektivním způsobem. Loki je navržen tak, aby jednoduše spolupracoval s Grafanou a umožňuje rychle vyhledávat a vizualizovat logy.

- **Tempo** - Poskytuje jednoduše ovladatelný open-source nástroj pro sledování distribuovaných požadavků. Tempo podporuje ukládání a načítání traces. Na rozdíl od mnoha jiných nástrojů pro traces, nevyžaduje Tempo žadné předem definované schéma. Je navržen tak, aby se bezproblémově integroval s Prometheus, Grafanou a aby jednoduše přijímal data z OpenTelemetry kolektoru.

- **OpenTelemetry** - Open source kolektor monitorovacích dat. Poskytuje jednotný způsob sběru, zpracování a exportu dat. Je konfigurovatelný a podporuje více pipeline, které mohou upravovat telemetrická data při jejich sběru. Výrazně zjednodušuje instrumentaci služeb, protože umožňuje agregovat a exportovat metriky, traces a logy do různých analytických a monitorovacích nástrojů. Poskytuje podporu pro export dat do Prometheus, Tempo i Loki.

Implementace Grafana Observability stacku je zajištěna pomocí obrazu nazvaného dta-lgtm a sestrojeného po vzoru Grafana LGTM (Loki, Grafana, Tempo a Mimir). Grafana LGTM kombinuje množinu monitorovacích nástrojů v rámci jediného obrazu s předchystanou konfigurací. Tím je odstíněna část konfigurace monitorovacího stacku a zjednodušen proces nasazení. Obraz použitý v rámci práce využívá kombinaci dříve zmíněných technologií (Loki, Tempo, Grafana, Prometheus a OpenTelemetry) zabalených a předkonfigurovaných v rámci Docker obrazu. Tím je v aplikaci zajištěno, že potřebná konfigurace pro vzájemné propojení nástrojů a datových zdrojů je předpřipravena. Stejně tak jsou součástí vytvořeného obrazu vlastní monitorovací dashboardy pro vizualizace.

V rámci aplikace mají jednotlivé služby nastaven export svých logů, traces a metrik do OpenTelemetry, respektive na adekvátní rozhraní dta-lgtm. Služby využívají existujících metrů a logů, ale také vytváří vlastní metriky a logy. Vlastní metriky zahrnují informace o počtu a druhu provedených operací. Z předpřipravených metrik, ať systémových nebo třetích strán jsou využity následující instrumentace:

- **System.Runtime** - Metriky běhového prostředí .NET.
- **System.Net.Http** - Metriky HTTP dotazů.
- **Microsoft.AspNetCore.Hosting** - Metriky hostovacího prostředí ASP.NET Core.
- **Microsoft.AspNetCore.Server.Kestrel** - Metriky serveru Kestrel.
- **Npgsql** - Metriky klientské knihovny PostgreSQL.

4.5.2 Monitorování hostitelského systému

Monitorování hostitelského systému poskytuje pro aplikaci klíčové informace o využití zdrojů a výkonu, jak ze samotného systému, tak i z jednotlivých kontejnerů. Pro monitorování Docker kontejnerů je využit nástroj CAdvisor. CAdvisor je schopen monitorovat kontejnery běžící na Dockeru, Kubernetes nebo jiných kontejnerových platformách. Poskytuje informace o využití procesoru, paměti, sítě a diskového I/O z pohledu hostitelského systému. Dalším využitým nástrojem je NodeExporter. NodeExporter je nástroj pro sběr metrik z hostitelského systému. Obdobně jako CAdvisor poskytuje informace o využití procesoru, paměti, sítě a diskového I/O. Data z obou zmíněných nástrojů jsou exportována do Prometheus, kde jsou následně zpracována a zprostředkována Grafaně.

4.6 Testovací nástroje

Za účelem testování monitorovacího stacku byl vybrán nástroj K6. Jedná se o moderní open-source nástroj pro zátěžové testování. Slouží k vytváření a spouštění výkonnostních testů nad aplikací. Nabízí bohaté API pro těchto testů v jazyce JavaScript. Umožňuje psát komplexní scénáře napodobující reálný provoz systému nebo simulovat hraniční situace. K6 podporuje různé systémové metriky, jako je doba odezvy, propustnost a chybovost. Nabízí široké možnosti rozšíření skrze API, což umožňuje přizpůsobení a integraci s dalšími nástroji pro komplexní sledování výkonu. K6 obsahuje nativní integraci exportu výsledku testování do databáze InfluxDB ve verzi 1. Pro zjednodušení procesu testování a zajištění opakovatelnosti testovacích scénářů byly vytvořeny skripty pro spuštění testů. Tyto skripty zajišťují opakovatelné spouštění testů v rámci Docker

kontejneru s nastavitelnými parametry. Skripty jsou vytvořeny v jazyce Bash a využívají nástroje K6 pro spuštění testů a zpracování výsledků. Zároveň pracují s Docker Compose orchestrací pro spouštění a vypínání služeb dle potřeby testů.

4.7 Nasazení aplikace

Následující pasáž popisuje náležitosti nasazení aplikace, včetně obecného popisu finální struktury nasazení, využitých nástrojů kontejnerizace a orchestrace, konkrétních verzí obrazů a použitou konfiguraci.

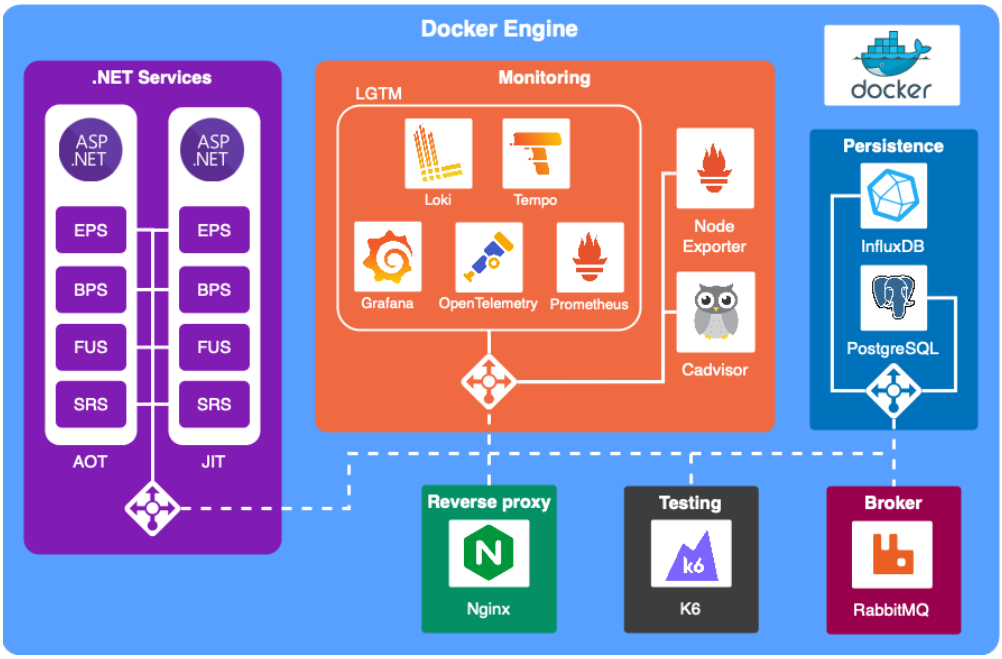
4.7.1 Přehled řešení

Řešení aplikace sestává z následujících sekcí a jednotlivých obrazů služeb a verzí, definovaných v rámci Docker Compose souboru.

- **Testovací služby** - Aplikace obsahuje testovací .NET služby FUS, SRS, BPS a EPS. Tyto služby jsou vytvořeny ve dvou kompilačních verzích - AOT a JIT. Každá služba je vytvořena jako obraz s názvem *dta-<service-name>* a značkou *<compilatioinMode>-latest*.
- **Komunikace** - Komunikační kanál mezi službami je zajištěn pomocí RabbitMQ. Pro RabbitMQ je využit obraz *rabbitmq:3-management-alpine*.
- **Monitorovací nástroje** - Monitorování zajišťuje Grafana Observability stack implementovaný v rámci obrazu *dta-lgtm:latest*, jenž obsahuje OpenTelemetry, Prometheus, Loki, Tempo a Grafanu. Pro měření výkonu hostitelského systému a export těchto dat jsou využity služby NodeExporter a CAdvisor s obrazy *node-exporter:latest* a *cadvisor-arm64:v0.49.1*.
- **Perzistence** - Pro perzistenci dat je využita PostgreSQL databáze. Pro PostgreSQL je využit obraz *postgres:latest*. Ukládání metrik je zajištěno pomocí InfluxDB a obrazu *influxdb:1.8.10*.
- **Směrování** - Funkci reverzní proxy zajišťuje Nginx ve verzi obrazu *nginx:latest*.

4.7.2 Kontejnerizace a orchestrace

Kontejnerizace služeb je zajištěna pomocí nástroje Docker. Docker je open-source platforma poskytující ekosystém pro správu kontejnerů. Jednotlivé služby jsou vytvořeny jako obrazy podle definicí Dockerfile.



Obrázek 4.4 Diagram nasazení aplikace

Orchestrace aplikace je zprostředkována rovněž nástrojem Docker, konkrétně formou *compose* utility. Ta umožňuje jednoduše nasadit a spravovat větší množství služeb. Definice nasazení aplikace je sepsána v souboru *compose.yaml*. V něm lze nalézt následující části:

- **volumes** - V této sekci jsou definovány všechny volume, které jsou využity v rámci stacku. Volume jsou definovány názvem a využity pro ukládání dat služeb. Konkrétně jsou využity úložiště pro data Grafany, OpenTelemetry, Prometheus a InfluxDB.
- **networks** - Konfigurace pro vnitřní síť aplikace, která je využita pro komunikaci mezi službami. Síť je definována názvem a typem. Pro potřeby aplikace se využívá síť s názvem *stack-network* a typ *bridge*, jenž funguje jako síťový most.
- **services** - Definuji sekci pro jednotlivé služby, které jsou součástí stacku. Každá služba je definována názvem služby - *container_name*, názvem obrazu - *image*, v případě lokálně sestavených služeb také definicí sestavení - *build*. Dále je definováno, jaké porty jsou mapovány z kontejneru do hostitelského systému - *ports*, jaké volume jsou připojeny k kontejneru - *volumes*, použité síťové rozhraní - *networks*, závislosti služby - *depends_on* a proměnné prostředí - *environment*. V případech testovaných služeb je uvedena dodatečná konfigurační sekce nasazení - *deploy*, jenž limituje dostupné zdroje paměti a procesoru.

4.7.3 Konfigurace nasazení

Za účelem běhu aplikace je klíčové správné nastavení konfigurace. Konfigurace je řešena na různých úrovních. Základní úroveň představuje soubor *compose.yaml*. V tomto souboru je dílčí konfigurace jednotlivých služeb řešena v rámci konfiguračních souborů, které jsou do služeb připojeny z hostitelského OS, a proměnných prostředí.

Nastavení směrování v rámci stacku je řešeno konfigurací proxy služby Nginx. Za tímto účelem obsahuje dva klíčové soubory, rozcestník v podobně statického *index.html* souboru a konfigurační *nginx.conf* soubor se směrovacími pravidly. Oba zmíněné soubory jsou do služby připojeny formou mapování virualizovaného repozitáře kontejneru. Směrovací pravidla jsou následující:

- / - cesta na statickou hlavní stránku-rozcestník aplikace
- /grafana - směrování na Grafanu

Nastavení telemetrie spočívá v definici rozhraní, nastavení chování služeb a systémů z niž se telemetrická data sbírají, jejich cíl pro zpracování, správu a vizualizaci. Značnou část konfigurace představuje propojení nástrojů aplikace v rámci obrazu dta-lgtm, k čemuž slouží konfigurační soubory. Ty obsahují výchozí minimalistickou konfiguraci pro jednotlivé nástroje. Výstupem dta-lgtm je individuální kontejner a zmíněná konfigurace je z podstaty interní a není potřeba s ní manipulovat s ohledem na požadavky práce. Mezi dodatečné konfigurace telemetrie napříč službami patří:

- **Testované služby** - Veškeré testovací služby mají nastavený endpoint pro export telemetrických dat. Toto nastavení je zprostředkováno proměnnou prostředí *OpenTelemetrySettings_ExporterEndpoint*.
- **LGTM** - Konfigurace pro LGTM je řešena pomocí proměnných prostředí. Je definována adresa a cesta, z které je k dispozici Grafana. Dále je umožněno anonymní přihlášení do Grafany.
- **Cadvisor** - Nastavení služby představuje dodatečné nastavení volumes, které jsou připojeny k kontejneru. Připojením systémových souborů je zajištěno sběr dat o využití systémových zdrojů. Toto nastavení je závislé na OS hostitelského stroje.

Jednotlivé služby mají vlastní dodatečné konfigurace, které jsou řešeny pomocí kombinace dle standardizovaného postupu pro konfiguraci .NET aplikací, a to konfiguračního souboru *appsettings.json* a proměnných prostředí. V první řadě řadě je použita

konfigurace ze souboru, načež je přepsána odpovídajícími hodnotami proměnných prostředí. Každá služba má definovaný specifický prefix pro identifikaci proměnných prostředí. Následující seznam popisuje dodatečné konfigurace jednotlivých služeb.

- **FUS - File Upload Service** - Obsahuje connection string (přístupový řetězec) pro připojení do databáze PostgreSQL.
- **BPS - Batch Processing Service** - Disponuje konfigurací pro připojení k RabbitMQ a konkrétní frontě.
- **EPS - Event Publishing Service** - Drží informaci o rozhraní RabbitMQ, konkrétní frontě a gRPC rozhraní služby FUS.

Služby využité pro perzistentní ukládání dat jsou konfigurovány pomocí proměnných prostředí. Jedná se o následující služby:

- **PostgreSQL** - Jsou definovány údaje pro uživatele databáze a název databáze.
- **InfluxDB** - Proměnné prostředí definují název databáze, uživatelské údaje a povolení přihlášení pomocí http.

Definice uživatelského rozhraní, respektive dostupných dashboardů, je dána při sestavení obrazu LGTM. V rámci něj jsou předdefinovány hodnoty pro připojení zdrojů dat, tj. Prometheus, Loki, Tempo a InfluxDb. Patřičné dashboardy zobrazující relevantní data pro různé scénáře systému byly předem připraveny a jsou k dispozici po otevření Grafany anonymním uživatelem. Následující seznam popisuje klíčové soubory konfigurace uživatelského rozhraní.

- **grafana-dashboards.json** - Definuje dostupné dashboardy v Grafaně. Dashboardy jsou definovány v JSON formátu a obsahují definici panelů, zdrojů dat a dalších parametrů.
- **grafana-datasources.json** - Obsahuje zdroje dat z kterých Grafana, respektive dashboardy, čerpají data.

5 TESTOVÁNÍ SCÉNÁŘŮ

Cílem práce je prouzkomat dopady kompilace AOT a JIT na výkon. Pokouší se určit, která strategie kompilace nabízí lepší výkon při různém zatížení a podmínkách. Za tímto účelem jsou využity testovací scénáře. Obsahem této kapitoly je definování metodiky a náležitostí testování aplikace vytvořené v předešlé kapitole. Jsou zde vyřčeny hypotézy a očekávání na chování aplikace. Následně jsou popsány jednotlivé scénáře, které jsou vytvořeny pro testování .NET služeb za specifických podmínek.

5.1 Metodika testování

Testování je prováděno za účelem získání kvantitativních dat o výkonu služeb. Metriky výkonu jsou shromážděny z dat aplikace, doby odezvy služeb a využití zdrojů během standardizovaných testů. Experimentální nastavení zahruje dvě hlavní součásti, řízení testovacího prostředí a testovacích služeb. Tyto služby jsou sestaveny ze stejné (až na vybrané nekompatibilní API) kódové báze v kompilačních režimech AOT i JIT. Testovací prostředí bylo standardizováno ve všech experimentech, aby bylo zajištěno, že jakékoli pozorované rozdíly ve výkonu lze připsat výhradně metodám kompilace a nikoli variantách HW nebo SW. Testy budou prováděny při identických nastaveních HW s následujícími specifikacemi:

- **Operační systém**: macOS Sonoma 14.4.1
- **Procesor**: Apple M1 8-core CPU (ARM 64bit)
- **Paměť**: 8 GB LPDDR4X

Operační systém a všechny služby na pozadí budou udržovány konzistentní, aby se minimalizovaly vnější vlivy na výsledky výkonu. Testovaným službám budou omezeny zdroje v orchestraci, aby se zabránilo nespravedlivé výhodě služby v jedné kompilaci před druhou. Služby budou připraveny do obrazů a použity v rámci orchestrace pomocí kontejnerů. Jednotlivé obrazy jsou založeny na architektuře ARM 64bit a Linuxových distribucích Alpine. Sběr dat je automatizován pomocí kombinace nástrojů pro monitorování systému a skriptů. K sběru dat hostitelského systému byl použit Cadvisor a Node Explorer. Telemetrie jednotlivých služeb je sbírána implementovanými metry a exportována do kolektoru OpenTelemetry. Testovací data jsou získána exportem výsledků testů do InfluxDB. Statické informace o průběhu a výstupech kopilace jsou získány kombinací nástroje .NET CLI, Docker a souborového systému. K porovnání strategií kompilace při různých podmínkách využívají testovací služby vybrané scénáře testující propustnost a vytížení zdrojů. Úlohy prováděné ve scénářích vystavují služby datovým

transakcím, přístupů k externím zdrojům, komunikací pomocí různých metodik anebo samotné spuštění v rámci orchestrace. Údaje o výkonnosti budou analyzovány pomocí statistických metod, aby se určily významné rozdíly mezi službami kompilovanými v režimu AOT a JIT.

5.1.1 Hypotézy

Na základě předběžného přehledu literatury a teoretických výhod každého kompilačního režimu byly formulovány následující hypotézy:

- **Hypotéza 1**: Vývoj služeb bude možný za pomocí obou kompilačních režimů, s minimálními rozdíly v použitém API a systémových knihovnách. Při použití knihoven 3. stran bude dostatečně jasná kompatibilita mezi oběma režimy.
- **Hypotéza 2**: Kompilace AOT má za následek rychlejší spouštění, ale bude vést k větším binárním velikostem aplikace ve srovnání s kompilací JIT. Výsledný virtualizovaný obraz aplikace bude ovšem mít výrazně nižší velikost kvůli absenci .NET runtime.
- **Hypotéza 3**: Kompilace AOT poskytuje lepší optimalizaci výkonu díky generování typů a funkcí, jež by museli být dodatečně tvořeny za běhu. Tím pádem je očekáváno, že služby kompilované do nativního kódu budou mít nižší řežii procesoru. Přítomnost staticky generovaných typů a funkcí však může způsobit vyšší paměťovou zátěž. Vytížení I/O operací bude srovnatelné.

5.2 Cíle porovnání služeb

Za účelem dosažení cílů této práce a porovnání .NET služeb v kompilačních režimech JIT a AOT jsou definovány cíle porovnání. Identifikují oblasti zájmu, které budou zkoumány a analyzovány v rámci vývoje, výstupu a experimentálního testování. Tyto cíle zahrnují:

- **Zkušenosti s vývojem** - Jedním z klíčových cílů je zachytit a analyzovat dopad různých kompilačních strategií na proces vývoje. To zahrnuje přípravu prostředí, sledování doby sestavení, celkové komplexity integrace a nasazení služeb v rámci architektury microservice. Posouzením těchto faktorů lze poskytnout subjektivní i objektivní náhled na to, jak jednotlivé metody kompilace ovlivňují každodenní fungování vývojářů, včetně potenciálních problémů nebo ztráty efektivity, které přinášejí kompilační režimy JIT nebo AOT.

- **Srovnání výstupů** - Tento cíl se zaměřuje na přímé porovnání programových výstupů metod kompilace JIT a AOT. Konkrétně se bude sledovat velikost vytvořených spustitelných souborů napříč platformami a architekturou. S tím je spojena analýza velikosti obrazů služeb, které hrají primární roli v kontejnerizovaném nasazení. Pochopení těchto výstupů pomůže pochopit vliv platformy a architektury na výsledné binární soubory. Zároveň bude ověřena úspora velikosti obrazů v rámci AOT kompilace.

- **Výkonnostní metriky** - Pro tuto práci je rozhodující porovnání výkonnostních ukazatelů za podobných provozních podmínek. Mezi sledované metriky patří doba odezvy, propustnost (počet požadavků, které je služba schopna zpracovat za vybrané scénáře) a chybovost. Hodnota doby odezvy bude ve specifických případech měřena včetně startu kontejneru. Tyto ukazatele poskytnou kvantitativní základ pro porovnání výkonu kompilací JIT a AOT při zvládání reálné provozní zátěže.

Dosažení těchto cílů stojí na dodržení metodiky a korektním analyzování aplikace. Je klíčové získat dostatečné poznatky, zkušenosti, výstupy a monitorovací data. Na jejich základě budou podpořena informovaná rozhodnutí týkající se optimálního využití kompilací JIT a AOT při vývoji mikroslužeb.

5.3 Definice scénářů

Scénáře jsou vytvořeny jako množina JavaScript souborů splňujících požadavky API nástroje K6. Každý scénář je definován přes jeden nebo více scriptových souborů. Tyto soubory obsahují kroky, které mají být provedeny, a data, která mají být použita. Pro sjednocení obecných nastavení jsou vytvořený konfigurační soubory, které jsou využity ve více scénářích. V rámci scénářů jsou využíváni Virtual Users (dále VUs), což jsou virtuální uživatelé nástroje K6, jenž vykonávají funkci definovanou v testovém skriptu. Konfigurace VUs je specifická pro každý scénář a je součástí skriptu. Pro zjednodušené a automatizované spuštění testovacích scénářů jsou definovány runner skripty, které zajišťují spuštění testů spolu se správou orchestrace.

Pro dodatečnou identifikaci dat jednotlivých scénářů je užito InfluxDB tagů, které jsou přidány k jednotlivým voláním v testech. Tím je zajištěno, že data z jednotlivých scénářů jsou jednoznačně identifikována a lze je následně zpracovat. Tagy jsou definovány následovně:

- **dta_service** - Značka pro identifikaci služby, která je testována. Má standardní formát hodnot *Služba-Kompilační režim*, kdy služba může nabývat hodnot *SRS, FUS, BPS, EPS* a kompilační režim nabývá hodnot *JIT, AOT*.

- **test_scenario** - Značka pro identifikaci scénáře, který je testován. Má standardní formát hodnot *scenario + číslo*.
- **test_id** - Identifikátor konkrétního testovacího scénáře. Nabývá libovolné hodnoty a slouží pro identifikaci konkrétních instancí, tedy spuštění testovacího scénáře. Výchozí hodnota je časová značka UNIX timestamp ve formátu vteřin.

5.4 Popis scénářů

Následující sekce obsahuje popis scénářů, které byly vytvořeny pro testování výkonu a škálovatelnosti mikroslužeb kompilovaných JIT a AOT. Ke každému scénáři patří odpovídající sada souborů scriptů a konfigurací. Rovněž každý scénář disponuje vlastním interaktivním dashboardem v Grafaně, který umožňuje sledovat výsledky testů v reálném čase.

5.4.1 Scénář 1 - Výkonnost komunikace

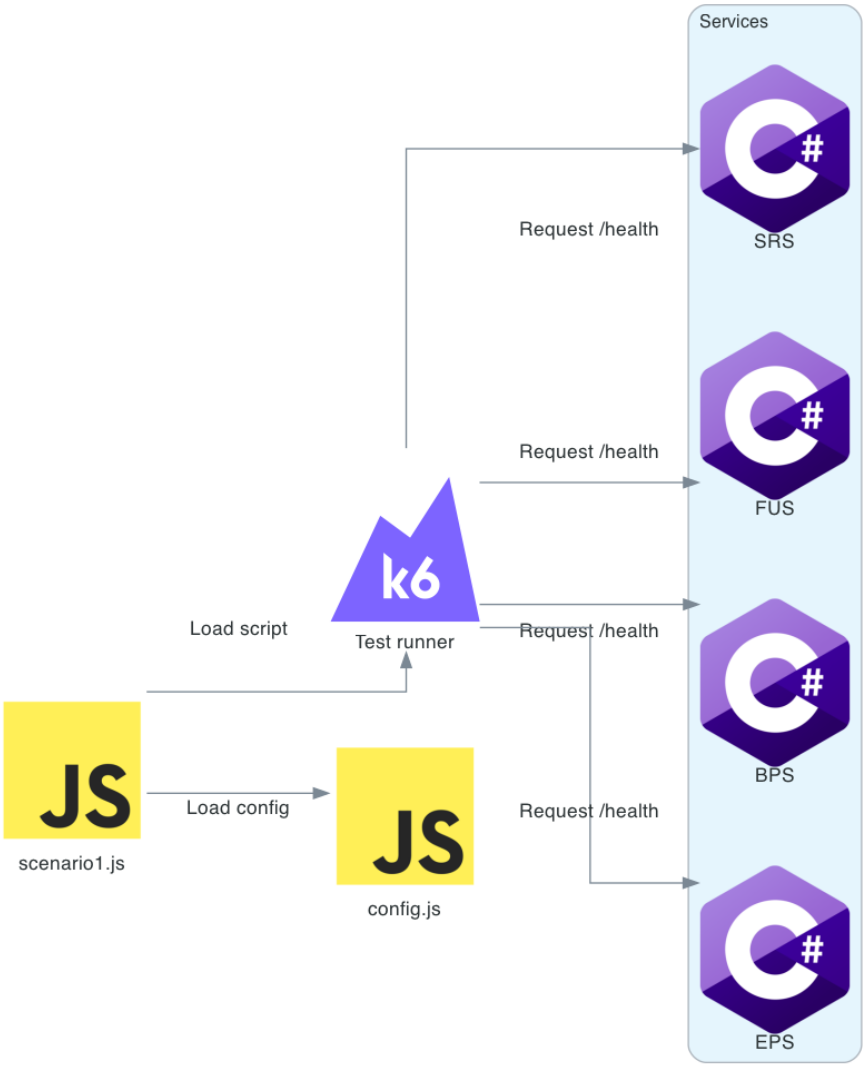
Scénář 1 je zaměřen na schopnost mikroslužeb odpovídat na požadavky. K tomuto účelu je využit základní endpoint */health*, který informuje o stavu služby. Scénář je vytvořen tak, aby simuloval vysoký počet požadavků na mikroslužby a zjišťoval, jak výkonné jsou jednotlivé služby při odpovídání. Jelikož healthcheck endpoint je triviální ve své implementaci, nehraje roli další režie spojená se zpracováním logiky požadavku. Tímto je zajištěno, že se otestuje maximální vliv jednotlivých nasazení na výkon mikroslužeb. Scénář se dělí na více kroků. Jednotlivé služby jsou rovněž otestovány individuálně, aby byl zajištěn dostatek zdrojů pro testovanou službu.

Relevantní služby pro tento scénář jsou všechny služby, které mají definovaný healthcheck endpoint. Následující postup popisuje průběh scénáře:

- **Krok 1** - Spuštění služeb v rámci stacku. Každá služba je spouštěna individuálně dle konkrétní služby a specifické kompilace dle konfigurace testu.
- **Krok 2** - Na služby jsou zasílány požadavky na healthcheck endpoint. Požadavky zasílá 10 VUs po dobu 5s, načež se počet VUs zvyšuje o dalších 10 v průběhu 10s. Po dosažení maximálního počtu VUs se počet snižuje na 0 během 5s.
- **Krok 3** - Po skončení všech služeb dochází k ukončení testovacího scénáře a zaslání dat o provedeném testu do InfluxDB.

5.4.2 Scénář 2 - Přístup k perzistenci

Cílem tohoto scénáře je otestovat výkonnost při zpracování množství požadavků na přístup k datům v perzistenci. Scénář se pokouší identifikovat dodatečné režie spojené

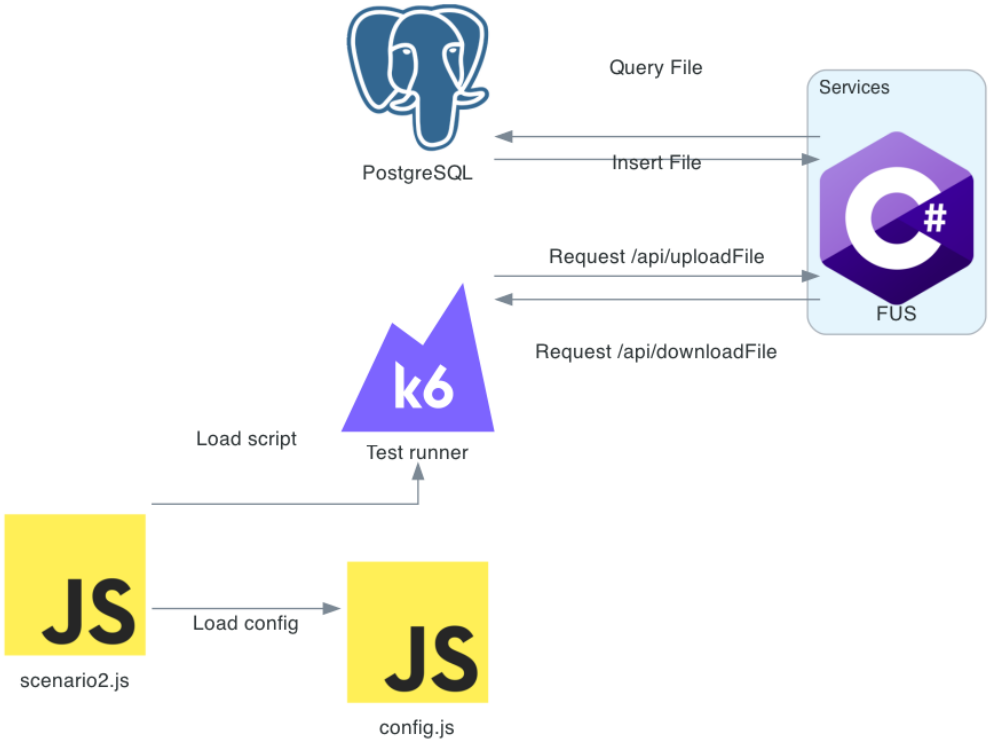


Obrázek 5.1 Diagram scénáře 1

s přístupem k databázi a zjišťuje, jak se služby při něm chovají. Zejména je cílem pozorovat potencionál rozdíl ve využití paměti jednotlivých služeb kompliovaných AOT a JIT.

Pro scénář je relevantní pouze služba FUS, která poskytuje rozhraní pro zápis a čtení dat z perzistentního úložiště. Průběh scénáře je následující:

- **Krok 1** - Služba je spuštěna v rámci stacku ve specifické kompilaci dle konfigurace testu.
- **Krok 2** - Na službu jsou zasílány požadavky na zápis i čtení dat z perzistentního úložiště. Požadavky zasílá 1 VU po dobu 1 minuty. Soubor použitý pro zápis je pevně stanoven a součástí repositáře testů. Jeho velikost činí 1MB.
- **Krok 3** - Služba ukončuje svoji činnost a K6 zasílá data o prevedeném testu.



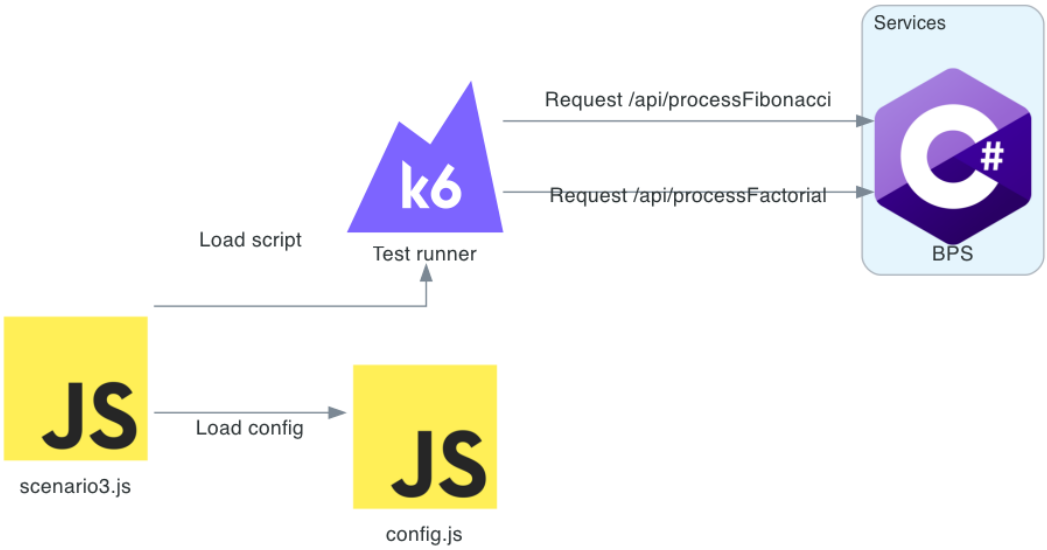
Obrázek 5.2 Diagram scénáře 2

5.4.3 Scénář 3 - Výpočetní zátěž

Cílem tohoto scénáře je otestovat schopnost mikroslužeb v jednotlivých kompilacích zpracovat náročnější operace. Scénář se zaměřuje na samotnou podstatu přístupu k vnitřnímu systémového API, efektivitě jeho využití a další režii, která by mohla být odlišná mezi JIT a AOT kompilací. Předmětem scénáře je výpočet 40-tého čísla Fibonacciho posloupnosti rekurzivní metodou. Algoritmus je implementován v rámci služby a volán zvenčí pomocí Rest API. Scénář je vytvořen tak, aby simuloval zátěž na službu a prozkoumal tak potencionální výkonnostní rozdíly v rámci přístupu k systémovému API a vyžížení zdrojů. Vyšší počet požadavků rovněž testuje schopnosti paralelního zpracování.

Ve scénáři má význam pouze BPS služba, která poskytuje rozhraní a logiku pro výpočet a čísla Fibonacciho posloupnosti. Průběh scénáře je následující:

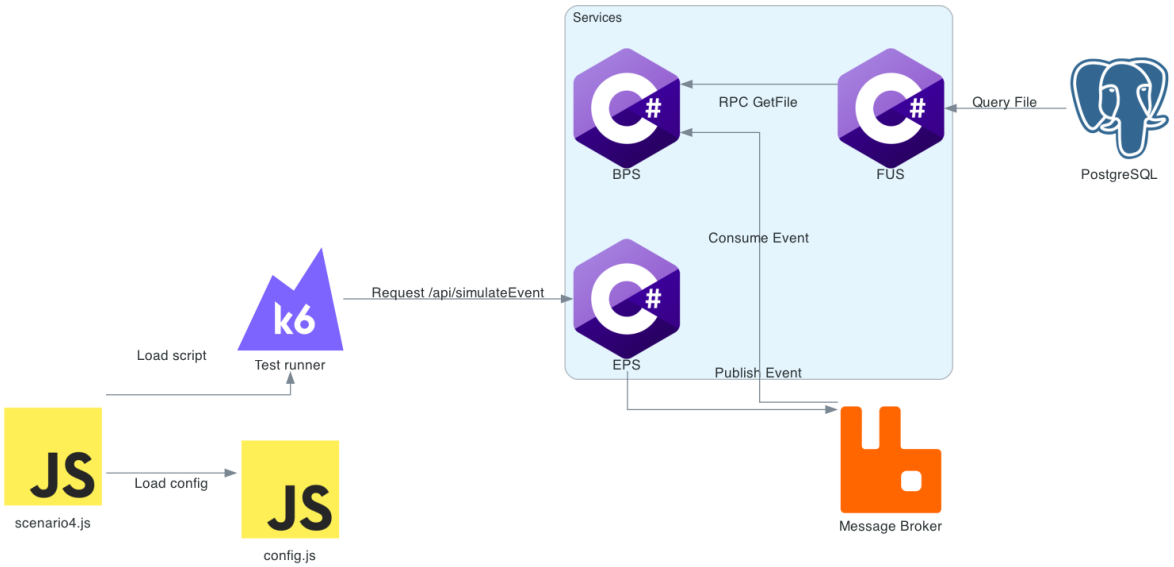
- **Krok 1** - Služba je spuštěna v rámci stacku ve specifické kompilaci dle konfigurace testu.
- **Krok 2** - Na službu jsou zasílány požadavky na výpočet 40-tého čísla Fibonacciho posloupnosti. Testování začíná na 3 VUs, jež jsou zvýšeny o 3 VUs po 5s. Po 10s na maximálním počtu dochází během 5s k vyřazení VUs.
- **Krok 3** - Činnost služby je ukončena a K6 zasílá data o prevedeném testu.



Obrázek 5.3 Diagram scénáře 3

5.4.4 Scénář 4 - Vzájemná komunikace služeb

Tento scénář je zaměřen na rychlost a zátěž celkového systému při splnění požadavků vyžadující komunikaci mezi službami. Scénář je vytvořen tak, aby vyvolal událost a vynutil přenos dat a zpracování v jiných službách. Snaží se identifikovat rozdílnou režii mezi kompilacemi JIT a AOT při složitější operaci u množiny služeb jako celku.



Obrázek 5.4 Diagram scénáře 4

Pro tento scénář jsou relevantní tři služby, které spolu komunikují. FUS hraje roli serveru, na něž se dotáže klient gRPC voláním. Následně přistupuje k perzistenci pro získání dat k splnění volání. BPS poslouchá nad předem definovanou frontou a vyčkává na zprávu pro zpracování. V momentu přijetí zprávy, zpracovává vyvolanou událost a získává data pomocí vzdáleného volání procedury z FUS. EPS na základě přijatého

volání přes REST API, zasílá služba EPS zprávu do předem definované fronty, na niž naslouchá BPS. Průběh scénáře je následující:

Průběh scénáře

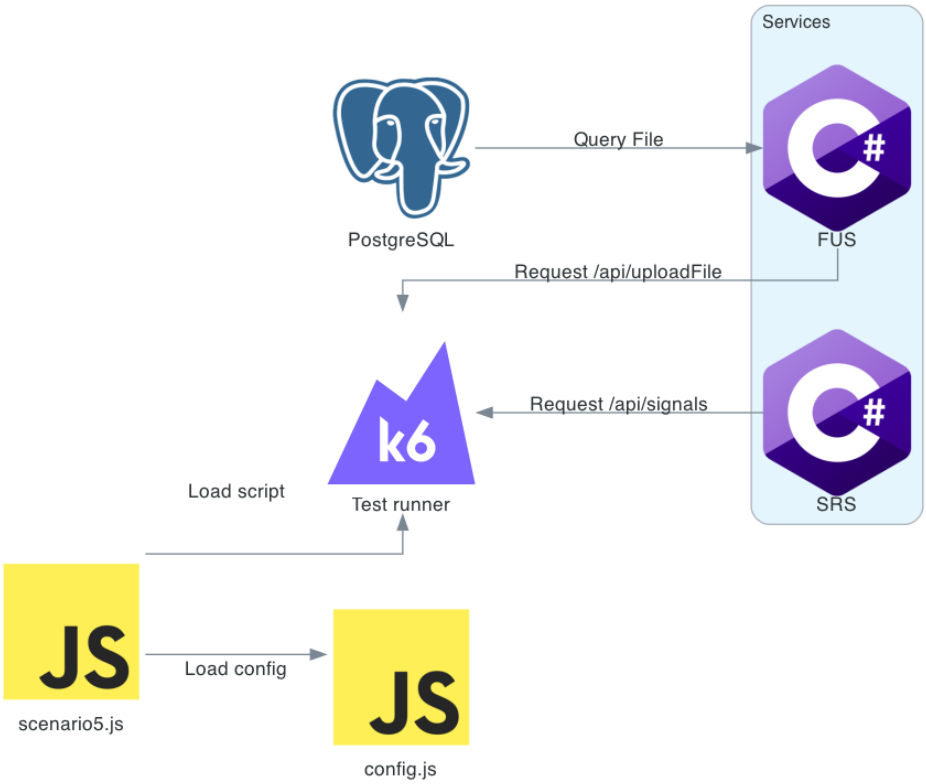
- **Krok 1** - Služby jsou spuštěny v rámci stacku ve specifické kompilaci dle konfigurace testu.
- **Krok 2** - Do služby FUS je nahrán textový soubor o velikosti 1MB pomocí REST API. Z odpovědi je získán identifikátor souboru.
- **Krok 3** - Do služby EPS jsou zasílány požadavy na zpracování dat pomocí REST API. Vykonává je jediny VU, který zasílá požadavky po dobu 1 minuty v intervalu co 5s.
- **Krok 4** - Služba EPS zprávu zasílá do fronty v RabbitMQ, na které naslouchá služba BPS.
- **Krok 5** - Služba BPS zprávu zpracovává z RabbitMQ fronty a získává data ze vzdáleného volání na službu FUS pomocí rozhraní gRPC.
- **Krok 6** - Služba FUS získává data z perzistence a zasílá je zpět službě BPS.
- **Krok 7** - Služba BPS provádí výpočet 40tého čísla Fibonacciho posloupnosti.
- **Krok 8** - Služby jsou ukončeny a v rámci K6 dojde k zaslání dat o provedeném testu do InfluxDB.

5.4.5 Scénář 5 - Rychlost odpovědi po startu služby

Cílem tohoto scénáře je otestovat rychlost spuštění služby. Scénář testuje, jak rychle je služba schopna odpovědět na požadavek po spuštění. V rámci testu je testován pouze healthcheck endpoint pro oddělení potencionální doménové režie od výsledného času. Základem scénáře je pomocí CLI příkazů vyvolat spuštění služby a ihned po jejím spuštění zaslat požadavek na získání dat. Tímto je zjištěno, že měření rychlosti odpovědi je závislé na rychlosti spuštění služby a její připravenosti k zpracování požadavku.

Pro scénář je využita pouze SRS. Průběh scénáře je následující:

- **Krok 1** - Dochází k zahájení časového měření a spuštění služby v rámci stacku dle konfigurace.
- **Krok 2** - V 10ms intervalech je zasílán požadavek na stav služby. Požadavky jsou zasílány po dobu 30s nebo dokud služba nevrátí stav *Helthy*.



Obrázek 5.5 Diagram scénáře 5

- **Krok 3** - Je zaslán požadavek na získání 3 generovaných signálů ze služby SRS.
- **Krok 4** - Služba SRS zpracovává požadavek a zprostředkovává data.
- **Krok 5** - Data jsou zaslána zpět klientovi.
- **Krok 6** - Po získání odpovědi dochází k ukončení časového měření a zaslání výsledku do InfluxDB do tabulky K6.
- **Krok 7** - Služba SRS je ukončena a dochází k případnému opakování pokusu experimentu, dokud nebyl proveden definovaný počet opakování (10 pokusů v rámci konfigurace).

5.5 Spouštění scénářů

Jednotlivé scénáře jsou spouštěny formou pomocných runner skriptů v jazyce Bash. Tyto scripty představují zjednodušenný způsob spuštění testů a zajišťují správu orchestrace testů, včetně spouštění a ukončování služeb, označení a exportování výsledků testů. Runner skripty mají jeden obecný parametr, kterým je identifikátor testu. Tento identifikátor je následně použit pro identifikaci výsledků testu a pro zobrazení výsledků v Grafaně. Jednotlivé skripty jsou pojmenovány dle čísla scénáře, který spouští. Pro

spuštění testů je nutné mít buď nastavené prostředí s nainstalovanými nástroji K6 a Docker, nebo upravit konfiguraci skriptu pro spuštění testů v rámci kontejneru.

5.6 Zpracování a vizualizace dat

Po provedení testování scénářů je nutné zpracovat a vizualizovat data, která byla získána. Data z průběhu testů jednotlivých scénářů jsou zpracována pomocí InfluxDB a zobrazena prostřednictvím Grafany. Týkající se výstupu nástroje K6 a v případě scénáře 5 pouze konkrétní metriky napřímo vložené po vzoru exportu K6 do jeho databáze v InfluxDB. Data o výkonu kontejnerů jsou získávána pomocí OpenTelemetry metrů z testovaných služeb a pomocí NodeExporteru a Cadvisor z hostitelského systému. Tato data jsou zaslána a spravována ve službě Prometheus a zprostředkována Grafaně formou data source. Jednotlivé .NET služby zprostředkovávají monitorovací data exportem metrik do OpenTelemetry kolektoru, který je následně dle druhu propaguje do jednotlivých služeb Loki, Tempo a Prometheus.

Vizualizace dat je zajištěna pomocí Grafany, respektive dahboardů pro jednotlivé scénáře. Ty mají nastaveny panely s relevantními daty pro daný scénář. Všechny scénáře jsou doplněny o společné panely, jenž slouží k zobrazení režie kontejnerů. Mimo to jsou vytvořeny dashboardy s obecnými daty o výkonu a zátěži služeb.

III. ANALYTICKÁ ČÁST

6 ANALÝZA APLIKACE

Tato kapitola se zabývá analýzou aplikace z hlediska vývoje, výstupu a výkonu. Využívá k tomu definovanou metodiku a scénáře testování. Výsledky jsou důkladně analyzovány a závěry shrnuty v jednotlivých sekcích.

6.1 Analýza vývojového procesu

Následující sekce popisuje vývojový proces, tak jak se týkal testovaných služeb. Vývojový proces byl založen na experimentaci a snaze primárně vyzkoušet dostupných knihoven a nástrojů. Bylo tak provedenu za cenu řešení problémů z nekompatibility a vynucených změn implementace.

```
8   #if AOT
9   using DTA.Models.Extensions;
10  using DTA.Models.JsonSerializers;
11  #elif DEBUG_JIT
12  using DTA.Extensions.Swagger;
13  #endif
14
15  // Create builder
16  #if AOT
17  var builder = WebApplication.CreateSlimBuilder(args);
18  #else
19  var builder = WebApplication.CreateBuilder(args);
20  #endif
```

Obrázek 6.1 Ukázka kódu s vyzualizací direktiv dle konfigurace

6.1.1 Vývojové prostředí

K vývoji byl použit IDE Rider od společnosti JetBrains. Vyzkoušena byla rovněž i práce ve Visual Studio 2022 Community Edition a Visual Studio Code s doporučenými rozšířeními od Microsoft. Všechna vývojová prostředí jsou kompatibilní, co se týče procesu kompilace respektive sestavení, jelikož to se odehrává pomocí CLI .NET. Samotný vývoj s ohledem na práci s direktivami pro různé kompilace byl značně zjednodušen vizualicemi, jenž poskytovala vývojová prostředí Rider a Visual Studio. Obdobně byla v těchto IDE zjednodušena i analýza a hledání chyb díky integraci referencí na kód generovaný na pozadí pro kompatibilitu s AOT. V tomto ohledu Visual Studio Code zaostávalo. S ohledem na aktivní vývoj a podporu, jenž je ze strany Microsoft poskytována podpoře vývoje .NET ve Visual Studio Code (po vyřazení produktu Visual Studio pro Mac), lze očekávat, že se tato situace v budoucnu změní.

```
app.MapPost (pattern: "/api/file/upload", UploadFile).DisableAntiforgery();
```

Obrázek 6.2 Vizualizace zachycení metody interceptorem v Rider IDE

```
[InterceptsLocation(filePath: @"/Users/noe/School/Thesis/Source/DTA/Services/DTA.FUS/Api/Rest/FileModule.cs", line: 12, column: 13)]
very complex (260%)
internal static RouteHandlerBuilder MapPost1(
    this IEndpointRouteBuilder endpoints,
    [StringSyntax("Route")] string pattern,
    Delegate handler)
{
    MetadataPopulator populateMetadata = (methodInfo, options) =>
    {
        Debug.Assert(condition: options != null, message: "RequestDelegateFactoryOptions not found.");
        Debug.Assert(condition: options.EndpointBuilder != null, message: "EndpointBuilder not found.");
        options.EndpointBuilder.Metadata.Add(item: new System.CodeDom.Compiler.GeneratedCodeAttribute(tool: "Microsoft.AspNetCore.Ht
        options.EndpointBuilder.Metadata.Add(item: AntiforgeryMetadata.ValidationRequired);
        options.EndpointBuilder.Metadata.Add(item: new AcceptsMetadata(contentTypes: GeneratedMetadataConstants.FormFileContentType
        return new RequestDelegateMetadataResult { EndpointMetadata = options.EndpointBuilder.Metadata.AsReadOnly() };
    };
    RequestDelegateFactoryFunc createRequestDelegate = (del, options, inferredMetadataResult) =>
    {
        Debug.Assert(condition: options != null, message: "RequestDelegateFactoryOptions not found.");
        Debug.Assert(condition: options.EndpointBuilder != null, message: "EndpointBuilder not found.");
        Debug.Assert(condition: options.EndpointBuilder.ApplicationServices != null, message: "ApplicationServices not found.");
        Debug.Assert(condition: options.EndpointBuilder.FilterFactories != null, message: "FilterFactories not found.");
        var handler:Func<IFormFile,IFileService,…> = Cast(del, _:global::System.Threading.Tasks.Task<global::Microsoft.AspNetCore.H
        EndpointFilterDelegate? filteredInvocation = null;
        var serviceProvider = options.ServiceProvider ?? options.EndpointBuilder.ApplicationServices;
```

Obrázek 6.3 Ukázka kódu nové implementace metody vygenerované na pozadí v Rider IDE

Příprava nástrojů a sestavení projektů bylo vyzkoušeno jak na platformě macOS, tak na Windows 10. Návod uvedený v oficiální dokumentaci pro platformu macOS úspěšně umožnil připravit vývojové prostředí a sestavit aplikace. [29] V případě Windows a návodu z dokumentace nastala chyba, která vyžadovala zásah. Pro sestavení projektů je nutné mít nainstalován nástroj pro vývoj C++ z dostupných komponent Visual Studio. Tento nástroj je nutný pro linkování knihoven a vytváření nativních kódů. Bohužel pro jeho správné fungování se musí nacházet na adrese v rámci souborového systému, jenž neobsahuje mezery nebo speciální znaky. Bohužel výchozí instalační cesta k nástroji Visual Studio obsahuje mezeru, což způsobuje chybu při sestavení aplikací v režimu nativní AOT. Tomuto problému lze předejít instalací nástroje na jiné místo, než je výchozí cesta a dohlédnout na validní adresu bez mezer a speciálních znaků. V případě již nainstalováného nástroje je potom potřeba jak reinstalace Visual Studio, tak i zmíněné komponenty pro vývoj C++ a vybrat validní cestu. Celý problém je zvýrazněn výchozím nastavením OS Windows instalovat programy do složky *Program Files*, která sama o sobě obsahuje mezeru a způsobuje problém.

6.1.2 JIT

Vývojový proces pro kompilaci služeb JIT se zacílením na .NET runtime probíhal standartním způsobem. Veškeré dostupné knihovny a nástroje byly plně kompatibilní s JIT

kompilací. Nedošlo k žádným nepředpokládaným problémům. Znatelný rozdíl oproti běžnému vývoji byl výběr technologií, který přihlížel k potencionální kompatibilitě s AOT a tedy řešení, které inherntně vyžadovala funkce rezervované pro využití .NET runtime, byly ihned zavrženy.

6.1.3 AOT

Kompilace do nativního kódu probíhala s průběžnými problémy. Podpora ze strany knihoven 3. stran ve spoustě případů neodopvídala deklarovaným možnostem. Vývojový proces byl značně zpomalován nutností řešení problémů, které byly způsobeny nedostatečnou podporou. Experimentace s řešeními často vyústila v nutnost změny implementace, případně v implementaci zcela vlastní.

6.1.4 Knihovny třetích stran

Pro zjednodušení procesu vývoje a využití existující funkcionality byly využity knihovny třetích stran. Následující seznam obsahuje knihovny, které byly využity použity v rámci vývoje a zda byly kompatibilní s AOT kompilací.

- **Entity Framework** - Entity framework se pyšní vysokou kompatibilitou s AOT kompilací. V rámci vývoje nebyly zaznamenány problémy, avšak následné testování se ukázalo problematické. EF jakožto plnohodnotný ORM framework stopuje stav objektu a jeho změny. Toto chování bohužel vyžaduje dynamické generování kódu, což je v rozporu s možnostmi AOT kompilovaného kódu. Vypnutí této funkcionality je pouze částečné, neb EF stále vyžaduje reflexi při vkládání nových entit do databáze.
- **Fluent Migrator** - Fluent Migrator je knihovna, která umožňuje verzování databáze pomocí kódu. V rámci testování bylo zjištěno, že knihovna využívá reflexi pro načítání migrací. Toto chování je v rozporu s AOT kompilací a výsledkem je chyba při spuštění migrace. Problém byl vyřešen vytvořením vlastního minimalistického migrátoru, který nepoužívá reflexi.
- **Grpc** - Vytváření rozhrání a modelů pro gRPC komunikaci vyžadovalo využití přístupu model first. Tento přístup využívá generátorů pro tvorbu kódu, definijucího kódového rozhraní pro .NET. Tímto je dosaženo vygenerování veškerého potřebného kódu v době kompilace a je zajištěna kompatibila s AOT. Pro definici modelu code first ovšem kombatibila s AOT není zajištěna.
- **Párování konfigurace** - V rámci systémové .NET knihovny je umožněno volání API, jenž načte data ze sjednocení stavu proměnných prostředi a konfiguračního

souboru. Součástí API je volání metody mapující tuto konfiguraci na předem definovaný objekt. Toto chování dle dostupných informací není v rozporu s AOT kompilací a volání relevantního kódu neprodukuje AOT warning. Z testování však vyplynulo, že mapování konfigurace ne objekt bylo problematické a neprobíhalo správně. Z toho důvodu je v případě AOT kompilace za pomocí deriktivy použité přímé načtení jednotlivých hodnot z konfigurace, dle stromomvého klíče.

6.2 Výstup služeb

Samotný proces nativní AOT a JIT kompilace je různě výkonnostně náročný. Při tvorbě obrazu službeb, ale i kompilace je hlavní náročná operace *restore*, která stahuje potřebné závislosti a balíčky pro projekt. Proces kompilace je vysoce závislý na specifickém HW, SW a přítomnosti závislostí. Pro účely testování byly potřebné NuGet balíčky nacachovány v systému. Následující tabulka zobrazuje přehled časové náročnosti kompilace služeb pro oba kompilační cíle. Data byla získána na základě průměru z 6 pokusů. Nejdelší a nejkratší hodnota kompilace byla zahozena pro omezení vlivu cachování. Jelikož je hodnota ovlivněna externím zatížením systému, má výsledek pouze orientační charakter.

K získání času výstupu bylo využity diagnostického režimu příkazu *dotnet*. Pro AOT byl použit příkaz *dotnet publish -v d -c Release-AOT -r osx-x64*, pro získání výstupu JIT byl použit příkaz *dotnet publish -v d -c Release-JIT -r osx-x64 –self-contained false*.

Tabulka 6.1 Čas kompilace služeb

Služba	JIT (s)	AOT (s)	AOT % nárůst
SRS	01.99	19.49	979.3
FUS	03.85	30.36	788.5
BPS	02.02	20.74	1026.7
EPS	01.85	20.05	1083.7

Výstup poukazuje na dodatečnou režii intercepterů, generátorů, nástroje pro generování nativního kódu a linkeru. Výsledek se shoduje s předpoklady větší náročnosti AOT kompilace. Pohled na velikost výstupního souboru služeb v kompilačních režimech AOT a JIT je zobrazen v následující tabulce. Služby cílí na architekturu ARM a OS macOS. Pro vytvoření výstupů na základě JIT byl použit příkaz *dotnet publish -c Release-JIT -r osx-x64 /p:PublishSingleFile=true –self-contained false*, pro vytvoření výstupů AOT byl použit příkaz *dotnet publish -c Release-AOT -r osx-x64*.

Velikost samotného výstupního programu vyšla dle očekávání definovaného v hypotéze 2. Program má výrazně menší velikost v případě JIT kompilace. To je dáno tím, že výstupní program této kompilace je závislý na .NET runtime, který poskytuje dodatečnou obecnou funkcionalitu a vytváří nativní kód včetně generování typů až za

Tabulka 6.2 Velikost programu služeb pro architekturu osx-arch64

Služba	JIT (MB)	AOT (MB)	AOT % nárůst
SRS	5.7	21.4	375.4
FUS	12.4	28.4	229.0
BPS	6.0	21.8	363.3
EPS	6.0	21.7	361.6

běhu aplikace. Rovněž proces sestavení obsahuje méně kroků.

Z pohledu velikosti výstupního programu nativní AOT kompilace má smysl porovnat výstup napříč architekturami a OS. Následující tabulka ukazuje velikost výstupu pro Windows v architektuře x64, macOS v architektuře ARM (označení arch64) a Linux v architekturách x64 a ARM. Pro vytvoření výstupů na základě JIT byl použit příkaz *dotnet publish -c Release-AOT -r <target_ OS>-<target_ architecture>*. V příkazu *<target_ OS>* a *<target_ architecture>* představuje vybraný operační systém a architekturu.

Tabulka 6.3 Velikost výstupu programu AOT služeb pro různé OS

Služba	win-x64 (MB)	osx-arch64 (MB)	linux-x64 (MB)	linux-arch64 (MB)
SRS	17.2	21.4	19.4	20.2
FUS	22.9	28.4	25.8	26.8
BPS	17.5	21.8	19.8	20.6
EPS	17.4	21.7	19.7	20.5

Velikost výstupního programu naznačuje větší optimalizovanost výstupu pro Windows. Tato skutečnost může být způsobena dostupností specifických knihoven ve Windows, respektive API. Následkem toho je menší množství závislostí v rámci výstupního programu. Obecně výsledek vychází lépe pro architekturu x64 než ARM. Cílové využití nativních AOT služeb je vhodné pro cloudové nasazení, které ve většině případů běží na Linux server, kdy architektura ARM vykazuje efektivnější provoz k poměru nákladu. Lze tedy předpokládat, že tato architektura byla optimalizována. Z tohoto předpokladu plyne že výsledek Linux x64 a Linux ARM ukazuje na roli architektury na velikost výstupu programu, kdy instrukční sada x64 architektury umožňuje menší výstupní program.

Sestavení obrazu je závislé na přípravu prostředí, vyhodnocení a stažení závislostí, kompilaci a publikování aplikace. Výstupné obrazy jsou založené na linuxovém systému, Alpine s .NET runtime v případě JIT výstupu služby, zredukované Ubuntu v případě nativního AOT výstupu. Z pohledu použitelnosti výsledného obrazu služeb má smysl měřit velikost výstupního obrazu. Následující tabulka zobrazuje velikost obrazu služeb pro oba kompilační cíle. Použitý příkaz je *docker build -t <service>:<tag> -f Dockerfile-*

<target> .. V příkazu *<target>* představuje vybranou kompilační metodu AOT nebo JIT a *<service>* jméno služby.

Tabulka 6.4 Velikost obrazu služeb

Služba	JIT (MB)	AOT (MB)	AOT % zmenšení
SRS	121.97	31.41	74.3
FUS	134.36	38.32	71.5
BPS	122.39	31.40	74.3
EPS	122.26	31.74	74.0

Výsledek potvrzuje hypotézu 2, tedy že větší výstupní velikost programu nativní AOT kompilace bude kompenzována menší velikostí výstupního obrazu.

6.3 Analýza testování

Následující sekce se zabývá analýzou testovacích scénářů a výsledků testování. Testování bylo provedeno na základě předem definované metodiky. Podkladem testů byly definované scénáře, které byly vytvořeny s ohledem na funkční a nefunkční požadavky. Při testování byl nezávisle na spuštěný test zaznaménáván stav hostitelského systému s ohledem na spuštěné kontejnery a využití systémových prostředků. Samotné služby využívaly předem definované metry ve frameworku ASP.NET pro dodatečnou diagnostiku a monitorování. Výsledky testování byly zaznamenány a analyzovány.

6.3.1 Scénář 1 - Výkonnost komunikace

První scénář se zabíral jednoduchou funkcionalitou dotazu na healthcheck endpoint a meřením výkonu kestrel serveru u odpovědí na požadavky skrze REST API. Následující tabulka zobrazuje průměrné využití zdrojů a dobu odpovědi služeb v testovacím scénaři 1 pro oba kompilační režimy.

Tabulka 6.5 Průměrné využití zdrojů a doba odpovědi healthcheck služeb

Služba - Režim	CPU (ms)	IO (ns)	Paměť (MB)	Doba požadavku (ms)
SRS-AOT	3.41	0.550	41.1	1.61
SRS-JIT	9.69	0.453	41.3	3.84
FUS-AOT	1.99	0.825	52.5	1.27
FUS-JIT	7.62	0.458	39.3	2.22
BPS-AOT	1.21	0.425	37.9	2.57
BPS-JIT	9.24	0.550	36.3	1.96
EPS-AOT	2.47	0.451	36.5	2.07
EPS-JIT	6.63	0.686	35.3	3.09

Testování přineslo rozdílné výkonostní výsledky mezi JIT a AOT kompilací. Dle předpokladu AOT služby využívaly méně času CPU. Paměťová stopa však u nich byla

větší. Konečně, AOT služby byly schopné v průměru rychleji odpovídat. Výsledek poukazuje na dodatečnou režii od CLR na CPU v případě JIT služeb. Větší využití paměti u AOT služeb poukazuje na režii vygenerovaných typů a přítomnost dalších závislotí v programu načteného v paměti. Rozdíl využití IO je s ohledem na dobu využití a počet požadavků zanedbatelný. Čistá rychlost zpracování požadavku ukázala na rychlejší zpracování AOT službami. I když tato hodnota není v mnoha případech kritickým faktorem. Avšak v případě velkého množství využitých služeb pro splnění požadavku, může být rozdíl v řádech milisekund znatelný.

6.3.2 Scénář 2 - Přístup k perzistenci

Scénář se zabýval výkonností přístupu k perzistenci, respektive zachystením reálného scénáře, kdy jsou data získávána a ukládána do databáze. Faktorem byla jak samotná rychlost služby v ohledu komunikace a serializace dat, tak rychlost zpracování požadavku databází. Následující tabulka zobrazuje průměrné využití zdrojů a dobu odpovědi služby FUS v testovacím scénáři 2 pro oba kompilační režimy.

Tabulka 6.6 Průměrné využití zdrojů službou FUS a doba odpovědi stažení a nahrání souboru

Služba - Režim	CPU (ms)	IO (ns)	Paměť (MB)	Doba požadavku (ms)
FUS-AOT	1.9	2.208	29.3	4.18
FUS-JIT	16.7	2.000	60.9	8.05

Ve výsledku je vidět výrazný rozdíl ve využití zdrojů mezi AOT a JIT verzi služby. Oproti předchozímu měření je AOT paměťově efektivnější. Poukazuje na méně efektivní datovou manipulaci při serializaci v přístupu k perzistenci, ale také při serializaci a kompresi souboru na rozhrání REST API. V případě doby odpovědi služby je velmi znatelný rozdíl služby kompilované JIT kdy její hodnota činila 93.6 ms. Následkem JIT kompilace potřebného kódu při prvním volání byla tato doba výrazně vyšší než v dalších voláních. Oproti tomu AOT varianta služby měla i při prvním volání odpoveď srovnatelnou s průměrným voláním a to 11.8 ms.

Tabulka 6.7 Průměrné využití GC službou FUS

Služba - Režim	Alokovaná paměť (MB)	Doba běhu (ms)	Velikost objektů (MB)
FUS-AOT	25.8	25.9	15.2
FUS-JIT	14.0	5.3	12.9

Přítomnost vygenerovaných typů a funkcionality v nativní AOT verzi služby má za výsledek větší alokace paměti, jenž jsou následně uvolněny, a větší doba běhu GC.

6.3.3 Scénář 3 - Výpočetní zátěž

Za účelem zjištění výkonnosti služeb, jejich potencionálně odlišné využití systémového API byl otestován scénář výpočetní zátěže. Na jednotlivé služby byly vysílány požadavky na výpočet 40-tého Fibonacciho čísla rekurzivní metodou. Následující tabulka zobrazuje průměrné využití zdrojů a dobu odpovědi služeb v testovacím scénáři 3 pro oba kompilační režimy.

Tabulka 6.8 Průměrné využití zdrojů službou BPS a doba odpovědi výpočtu Fibonacciho čísla

Služba - Režim	CPU (ms)	IO (ns)	Paměť (MB)	Doba požadavku (s)
BPS-AOT	58.4	2.940	46.1	5.80
BPS-JIT	48.8	1.164	44.8	6.07

Výsledky testování ukázaly že při náročné výpočetní zátěži žádná z kompilací nebyla výrazně výkonnější. Oproti dřívějším situacím využila AOT kompilovaná služba více času CPU než JIT kompilovaná. Tato skutečnost naznačuje, že výkonnostní výhody AOT služeb jsou při větší zátěži srovnány. V ostatních měřených hodnotách byly rozdíly zanedbatelné.

Tabulka 6.9 Průměrné využití GC službou BPS

Služba - Režim	Alokovaná paměť (MB)	Doba běhu (ms)	Velikost objektů (MB)
BPS-AOT	19.0	18.3	9.6
BPS-JIT	9.1	3.1	8.5

Dle očekávání z dřívejších výsledků i zde AOT varianta služby si vyžádala více běhu GC a alokované paměti. Na výslednou rychlost odpovědi služby to však nemělo vliv.

6.3.4 Scénář 4 - Vzájemná komunikace služeb

Komplexnější situace pro aplikaci byla simulována ve čtvrtém scénáři. Situace simulovala kombinaci synchronní a asynchronní komunikace mezi službami, přístup k perzistenci a výpočetní zátěž.

Tabulka 6.10 Průměrné využití zdrojů službami dle nasazení v kompilačních režimu

Režim	CPU (ms)	IO (ns)	Paměť (MB)
AOT	8.1	0.683	30.4
JIT	24.3	1.634	36.7

Výsledky ukázaly očekávané chování dle předchozích výsledků a hypotéz. Zpracování požadavku skrze více služeb je obdobně náročné pro oba kompilační režimy. Samotné využití zdrojů neposkytuje opověď pro vhodnější kompilační režim pro množinu minimalistkých služeb zpracovávajících požadavek v rámci aplikace.

6.3.5 Scénář 5 - Rychlost odpovědi služby po startu

V tomto scénáři byla vyvolána simulace serveless nasazení. Jednotlivé varianty služby SRS byly v rámci testu spuštěny, kontrolovány než se dostaly do stavu *healthy* a následně nad nimi zavolán dotaz pro získání generovaných dat. Tabulka níže zobrazuje průměrné využití paměti a dobu odpovědi služby SRS v testovacím scénáři 5 pro oba kompilační režimy.

Tabulka 6.11 Průměrné využití zdrojů službou SRS a doba odpovědi včetně startu služby

Služba - Režim	Doba startu služby a požadavku (s)	Paměť (MB)
SRS-AOT	0.91	9.59
SRS-JIT	1.40	8.46

Výsledky ukázaly, že služba kompilovaná nativním AOT způsobem je rychleji dostupná a odpovídá na požadavky dříve, než služba kompilovaná pro .NET runtime. Oproti propagovaným zrychlení v dokumentaci .NET však nebylo dosaženo tak výrazného rozdílu. To je dáno režií kontejneru, tedy jeho OS. Rychlost odpovědi je tedy velkým poměřem závislá na konkrétním prostředí, ve kterém je služba spuštěna a scénáři nasazení.

6.4 Závěr analýzy

Na základě výsledků vývoje, výstupu a testování služeb lze zhodnoti definované hypotézy následujícím způsobem:

- **Hypotéza 1** - Hypotéza, že vývoj služeb s jak AOT, tak JIT kompilací je v rámci podporované funkcionality systémových knihoven a ASP.NET možný s podobným API se ukázal jako ne zcela pravdivý. Při vývoji nastaly komplikace se serializací konfigurace, na které bylo nutné reagovat využitím odlišného API. Zároveň tento způsob serializace nebyl kompilátorem označen jako potencionálně problematický. Další problémy nastaly s využitím Entity Framework. Tento ORM využívá pro provádění operací nad databází tzv. tracking, který zaznamená změny nad aplikačními objekty a podle nich tvoří výsledné databázové operace. Vypnutím trackingu bylo umožněno se na datové entity dotázat a aktualizovat je. Operace vložení nové entity však bez trackingu nebyla možná. Pro knihovny 3. stran lze obecně říci, že podpora AOT kompilace není vždy úplně zřejmá a i v situacích kdy AOT varování jsou implementovány, lze očekávat chybné chování.
- **Hypotéza 2** - Výsledky ukazují, že služby napsané v nativním kódu se výrazněji rychleji spouští jak na hostitelských systémech, tak ve virtualizovaném prostředí.

Zároveň binární velikosti samotných aplikací jsou mnohonásobně větší, než je tomu u služeb vyžadující .NET runtime. To je ovšem kompenzováno při virtualizovaném spuštění, kdy obraz služby pro vytvoření plnohodnotného kontejneru vyžaduje mnohem méně závislotí z hlediska paměti. Výsledné obrazy jsou tedy menší a rychleji spustitelné. Hypotéza byla potvrzena.

- **Hypotéza 3** - Na základě dostupných metrik bylo potvrzeno, že obecně služby kompilované AOT mají menší využití CPU. Využití paměti a IO se ukázalo být srovnatelné jak u služeb kompilovaných JIT v runtime. Toto chování je způsobeno rozdílem v době, kdy se generují potřebné typy a části funkcionality aplikace. JIT kompilace umožňuje za provozu využít méně paměti, nicméně ani tento fakt nebyl pravidlem ve všech scénářích. Zároveň bylo ale pozorováno zvýšené využití GC v případě služeb kompilovaných AOT. I přes tuto dodatečnou režii byly obecně efektivnější a hypotéza byla potvrzena. Rozdíly však nebyly jednoznačné a signifikantní na to, aby bylo možné určit, který z kompilačních režimů je výkonnější z pohledu běhu služeb a zpracování požadavků.

ZÁVĚR

V rámci diplomové práce byly analyzovány kompilační režimy JIT a nativní AOT na platformě .NET. První část představuje rešerše, ve které byly popsány základní principy fungování platformy .NET, jejich kompilačních režimů a cílů kompilace. Následně byla popsána architektura microservice, která slouží jako primární zacílení nativních AOT aplikací a která poskytuje vzor pro testovací nasazení. V neposlední řadě byla popsána problematika testování, telemetrie a monitorovacích řešení. V praktické části byly přiblíženy nástroje a techniky použité k vývoji, sestavení a nasazení testovacích služeb spolu s monitorovacími nástroj. Poté byly provedeny testy podle předem definované metodiky a scénářů. V analytické části byly výsledné data popsána a vyhodnocena. Výsledkem práce je komplexní analýza použití kompilačních režimů JIT a nativní AOT pro vývoj služeb v .NET. Dále byla vytvořena sada testovacích služeb, které slouží jako ukázka možností platformy. V neposlední řadě pro účely analýzy byl vytvořen testovací stack, jenž umožňuje vytváření a nasazování testovacích služeb. Výstup práce zprostředkovává přehled o možnostech a omezeních nativní AOT kompilace v .NET a poskytuje testovací aplikaci pro jednoduchou implementaci požadovaného scénáře. Dodává teoretický základ, zkušenosti a výsledky testování do specifické oblasti vývoje nativně AOT kompilovaných služeb.

Vývojový proces při kompilaci do nativního AOT kódu se ukázal nepřívětivý. Primárně podpora knihoven 3. stran a princip interceptorů a generátorů má za vinu subjektivně neintuitivní proces debugování kódu. Samotný programový výstup vyšel dle očekávání a nativní AOT služby produkují větší výstup než JIT služby. Situace se však obrací při kontejnerizovaném nasazení, kdy závislost JIT služeb na runtime prostředí produkuje daleko větší obrazy služeb. Ve výkonnostním porovnání byly obrazy nativních AOT služeb výrazně efektivnější ve využití systémových zdrojů. Výsledky testování ukázaly, že na platformě .NET nativní AOT aplikace mají obecně srovnatelný výkon jako aplikace v režimu JIT. Rozdíl je znatelný v situacích, kdy je nutno využít velké množství instancí stejné služby (plyne z velikosti obrazu) a v situacích, kdy je pro systém rozhodující rychlost zpracování požadavku službou včetně spuštění (Serverless nasazení). Konkrétně rozdíl rychlosti spuštění však nebyla natolik markantní z důvodu režie startování celého kontejneru služby k poměru s režií startu samotné služby. Výsledky výkonnostního testování byly zaznamenány a zpracovány do tabulek a grafů a jsou součástí práce. Zároveň byly zpracovány interaktivní dashboardy v rámci aplikace Grafana, jenž umožňují podrobný náhled na fungování systému v reálném čase.

Služby kompilované do nativního AOT kódu přináší specifické výkonnostní výhody za cenu kompatibility. Vývoj kódu je s ohledem na zažité postupy a praktiky v .NET nestandartní. Využitím interceptorů a generátorů je odebrána část inciativy z rukou

vývojáře a vytváří se na pozadí kompilace v .NET další úroveň abstrakce. Podpora knihoven a frameworků třetích stran je omezena a nelze se spolehnout na jejich plnou funkčnost. Tím připadá na vývojáře zodpovědnost za implementaci vlastních řešení, která by jinak byla dostupná. Při většině vzorů nasazení nejsou čistě výkonostní výhody dostatečným důvodem pro přechod na nativní AOT kompilaci a obětování funkcí runtime prostředí spolu s širokou podporou knihoven.

Mnoho výhod, jenž z platformy .NET plynou souvisí s možnostmi runtime prostředí. Nativní AOT kompilace má smysl ve specifických případech, jenž plynou z nutnosti rychlosti spuštění a velikosti výstupu aplikace (s přihlednutím k velikosti .NET runtime). Případy konkurenční výhody pro AOT kompilaci staví na předpokladu že existuje zájem či potřeba mít zdrojové kódy v .NET, respektive jazyce C#. Při rozmanitém technologickém přístupu, kdy je vývojář, respektive zapojený tým schopen přijmout jiný jazyk a framework, jsou výhody AOT kompilace ztraceny, zatímco nedostatky jsou zvýrazněny. Tento předpoklad je relativně v rozporu s požadavky na poskytování cloudových služeb, kdy je očekáváno silné technické a vědomostní zázemí a flexibilní přístup k technologiím. Dále tento předpoklad jde proti jedné z výhod microservice architektury a to možnosti kombinovat různé technologie a nástroje. Z širšího pohledu na platformu .NET a strategie Microfostu, nativní AOT zaplňuje specifickou díru v portfoliu technologií. Je totiž klíčová k poskytnutí kompletní sady nástrojů pro tvorbu kompletního řešení cloudové platformy pouze s použitím .NET platformy. Vývojáři, kteří se rozhodnou pro AOT kompilaci, by měli být obeznámeni s těmito specifiky a měli by být schopni je zohlednit v návrhu a implementaci řešení.

V návaznosti na testovací stack, který v práci vznikl za účelem výkonnostního testování služeb, se nabízí doplnit implementaci dalších služeb, případně rozšířit stávající. Podle vzoru současného řešení lze dodat další funkcionalitu, nastavit další zdroje telemetrie, případně rozšířit možnosti vizualizace dat. Z pohledu uživatelské přívětivosti se nabízí tvorba aplikace s grafickým rozhraním využívající princip Docker outside of Docker. Tímto by bylo možné zjednodušit spouštění konkrétních testovacích scénářů. V rámci webové aplikace by bylo možné nastavit parametry testování, spustit konkrétní test a prokliknout se odkazem na relevantní dashboard v Grafaně. S ohledem na citlivé data a přístupy, které aplikace zprostředkovává, se nabízí rozšíření o autentizaci a autorizaci v případě vystavení stacku v síti. Jelikož grafické rozhraní aplikace je založeno na aplikaci Grafana, jež je schopna připojit se k externím zprostředkovatelům autentizace, bylo by vhodné zapojit službu jako Keycloak pro sjednocení autentifikace napříč stackem.

SEZNAM POUŽITÉ LITERATURY

[1] TROELSEN, Andrew. *C# and the .NET Platform*. Second edition. Apress, 2003. ISBN 9781590590553.

[2] RICHTER, J. *CLR via C#: The Common Language Runtime for .NET Programmers*. 4th ed. Microsoft Press, Redmond, Wash., 2012. ISBN 978-0735667457.

[3] MICROSOFT. *.NET CLI overview*. Online. Microsoft Learn. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/tools/. [cit. 2024-05-01].

[4] MICROSOFT. *MSBuild*. Online. Microsoft Learn. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/tools/. [cit. 2024-05-02].

[5] PRICE, Mark J. *C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals*. Eight Edition. Packt Publishing, 2023. ISBN 978-1-83763-587-0.

[6] HARRISON, Nick. *Code Generation with Roslyn*. Apress, 2017. ISBN 978-1-4842-2211-9.

[7] WILLIAMS, Trevoir. *Microservices Design Patterns in .NET*. 1st Edition. Packt Publishing, 2023. ISBN 978-1-80461-030-5.

[8] DANYLKO, Jonathan R. *ASP.NET 8 Best Practices*. Packt Publishing, 2023. ISBN 978-1-83763-713-3.

[9] MICROSOFT CORPORATION. *ASP.NET Documentation* [online]. [cit. 2024-03-18]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-8.0

[10] LIBERY, Jesse a Rodrigo JUAREZ. *.NET MAUI for C# Developers*. Packt Publishing, 2023. ISBN 978-1-83763-169-8.

[11] ALLS, Jason. *Clean Code with C#*. Second edition. Packt Publishing, 2023. ISBN 978-1-83763-519-1.

[12] BOCK, Jason. *.NET Development Using the Compiler API. Second edition*. Apress, 2016. ISBN 978-1-4842-2111-2.

[13] MICROSOFT. *ReadyToRun Compilation*. Online. Microsoft Learn. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/deploying/ready-to-run. [cit. 2024-05-02].

[14] PFLUG, Kenny. *Native AOT with ASP.NET Core - Overview* [online]. 2023 [cit. 2024-02-23]. Available from: https://www.thinktecture.com/en/net/native-aot-with-asp-net-core-overview/

[15] PRICE, Mark J. *Apps and Services with .NET 8. Second Edition.* Packt Publishing, 2023. ISBN 978-1837637133.

[16] MICROSOFT. *Source Generators.* Online. Microsoft Learn. Dostupné z: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview. [cit. 2024-05-02].

[17] RICHARDSON, C. *Microservices Patterns: With Examples in Java.* O'Reilly Media, Sebastopol, Calif., 2018. ISBN 978-1617294549.

[18] DOCKER INC. *Docker Docs* [online]. 2013 [cit. 2024-03-13]. Dostupné z: https://docs.docker.com

[19] GAMMELGAARD, C. H. *Microservices for .NET Developers: A Hands-On Guide to Building and Deploying Microservices-Based Applications Using .NET Core.* 2nd ed. Apress, 2021, ISBN 978-1617297922.

[20] NEWMAN, Sam. *Building microservices.* Sebastopol, CA: O´Reilly Media, [2015]. ISBN 1491950358.

[21] SAZANAVETS, Fiodar. *Microservice Communication in .NET Using gRPC.* Packt Publishing, 2022. ISBN 978-1-80323-643-8.

[22] RICHARDSON, Chris. *Pattern: Saga.* Online. Dostupné z: https://microservices.io/patterns/data/saga.html. [cit. 2024-05-01].

[23] GARRISON, J.; NOVA, K. *Cloud Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment.* 1st ed. O'Reilly Media, Sebastopol, Calif., 2017. ISBN 978-1491984307.

[24] RIEDESEL, Jamie. *Software Telemetry: Reliable logging and monitoring.* Manning, 2021. ISBN 978-1617298141.

[25] MAJORS, Charity; FONG-JONES, Liz a MIRANDA, George. Observability Engineering: Achieving Production Excellence. O'Reilly Media, 2022. ISBN 978-1492076445.

[26] MOLKOVA, Liudmila a Sergey KANZHELEV. *Modern Distributed Tracing in .NET.* Packt Publishing, 2023. ISBN 978-1-83763-613-6.

[27] BLANCO, Daniel Gomez. *Practical OpenTelemetry: Adopting Open Observability Standards Across Your Organization*. Apress, 2023. ISBN 978-1484290750.

[28] CHAPMAN, Rob a Peter HOLMES. *Observability with Grafana*. Packt Publishing, 2023. ISBN 978-1-80324-800-4.

[29] MICROSOFT. *Native AOT deployment*. Online. Microsoft Learn. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/?tabs=net7

[30] SALITURO, Eric. *Learn Grafana 10.x*. Second Edition. Packt Publishing, 2023. ISBN 978-1-80323-108-2.

[31] MARCOTTE, Carl-Hugo. *Architecting ASP.NET Core Applications*. Third Edition. Packt Publishing, 2024. ISBN 9781805123385.

[32] AKINSHIN, Andrey. *Pro .NET Benchmarking*. Apress Berkeley, CA, 2019. ISBN 978-1-4842-4941-3.

[33] KOKOSA, K. *Pro .NET Memory Management: For Better Code, Performance, and Scalability*. For Professionals By Professionals. Apress, New York, 2018. ISBN 978-1484240267.

[34] .NET 7 Preview 3 Is All About Native AOT. RAMEL, David. Visual Studio Magazine [online]. 2022 [cit. 2024-03-19]. Dostupné z: https://visualstudiomagazine.com/articles/2022/04/15/net-7-preview-3.aspx

[35] MARTIN, Robert C. *Clean architecture: a craftsman's guide to software structure and design*. Robert C. Martin series. London, England: Prentice Hall, 2018. ISBN 978-0134494166.

[36] LOCK, A. *ASP.NET Core in Action*. 2nd ed. Manning Publications, Greenwich, CT, 2021. ISBN 978-1617298301.

[37] PFLB, INC. *User Manual for k6, an Open Source Tool for Load Testing*. PFLB [online]. 2021 [cit. 2024-04-01]. Dostupné z: https://pflb.us/blog/k6-user-manual/

[38] GARVERICK, Joshua a Omar Dean MCIVER. Implementing Event-Driven Microservices Architecture in .NET 7. Packt Publishing, 2023. ISBN 978-1-80323-278-2.

[39] MICROSOFT CORPORATION. *.NET Documentation* [online]. [cit. 2024-03-18]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/

[40] RAINTANK, INC. *Grafana Labs - Technical Documentation* [online]. [cit. 2024-03-22]. Dostupné z: https://grafana.com/docs/

[41] F5, INC. *NGINX Product Documentation* [online]. [cit. 2024-04-12]. Dostupné z: https://docs.nginx.com

[42] THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL Documentation* [online]. [cit. 2024-04-14]. Dostupné z: https://www.postgresql.org/docs/

[43] INFLUXDATA, INC. *InfluxDB v1 Documentation* [online]. [cit. 2024-04-14]. Dostupné z: https://docs.influxdata.com/influxdb/v1/

[44] ESPOSIO, Dino. *Microservices Design Patterns in .NET*. Pearson Education, 2024. ISBN 978-0-13-820336-8.

[45] NICKOLOFF, J.; KUENZIL, S. *Docker in Action*. 2nd ed. Manning Publications, Greenwich, CT, 2019. ISBN 978-1617294761.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AOT	Ahead of Time
API	Application Programming Interface
AST	Abstract Syntax Tree
CAS	Code Access Security
CD	Continuous Delivery
CI	Continuous Integration
CIL	Common Intermediate Language
CLI	Command Line Interface
CLR	Common Language Runtime
CORS	Cross-Origin Resource Sharing
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DevOps	Development and Operations
DLL	Dynamic Link Library
DRY	Don't Repeat Yourself
GC	Garbage Collector
GUI	Graphical User Interface
HW	Hardware
HTML	Hypertext Markup Language
HTTPS	Hypertext Transfer Protocol Secure
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management
IDE	Integrated Development Environment
IL	Intermediate Language
IO	Input/Output
IoT	Internet of Things
IP	Internet Protocol
JIT	Just in Time
JSON	JavaScript Object Notation
JWT	JSON Web Token
LINQ	Language Integrated Query
MAUI	Multi-platform App UI
MSIL	Microsoft Intermediate Language
MVC	Model-View-Controller
MVVM	Model-View-ViewModel
NoSQL	Not Only SQL

OAuth	Open Authorization
ORM	Object-Relational Mapping
OS	Operační Systém
PC	Personal Computer
R2R	Ready to Run
RAD	Rapid Application Development
RAM	Random Access Memory
REST	Representational State Transfer
RPC	Remote Procedure Call
SCM	Source Code Management
SDK	Software Development Kit
SQL	Structured Query Language
SSL	Secure Sockets Layer
SW	Software
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VB	Visual Basic
VM	Virtual Machine
VUs	Virtual Users
WASM	WebAssembly
XML	Extensible Markup Language

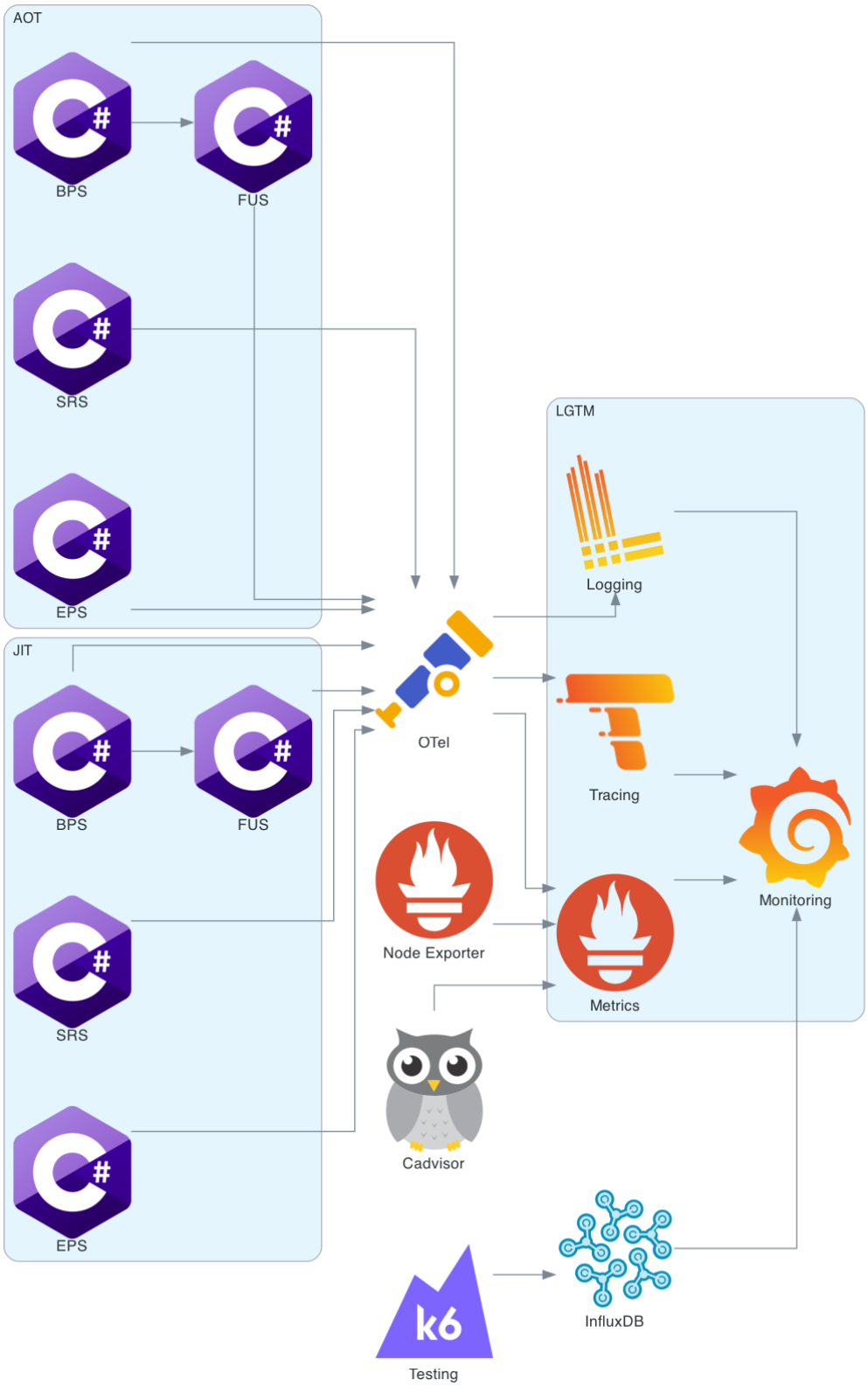
SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

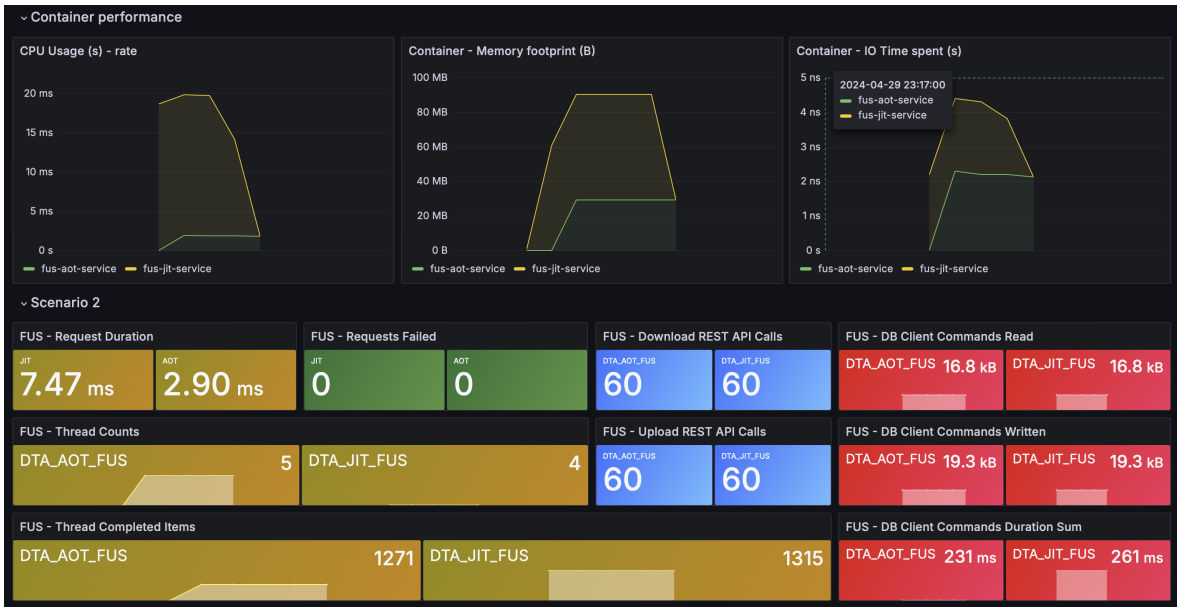
PŘÍLOHA P I. OBRÁZKOVÁ PŘÍLOHA

Příloha obsahuje obrázky vytvořeny v průběhu vývoje, testování a analýzy. Obrázky jsou v případě diagramů vytvořeny pomocí nástroje Graphviz a Python knihovny Diagrams mingrammer, v případě dashboardů se jedná o foto obrazovky.

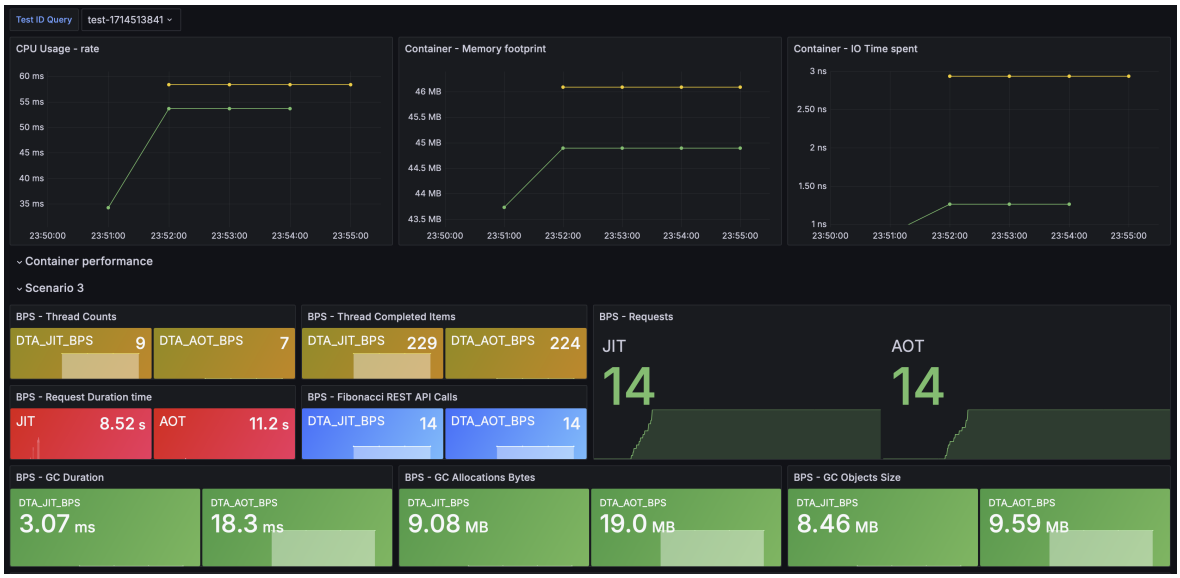


Obrázek 1.1 Telemetrie ve stacku

Obrázek 1.2 Scénář 1 - Grafana dashboard

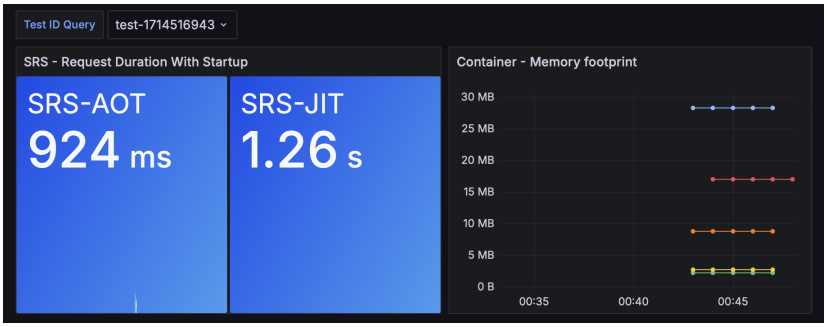


Obrázek 1.3 Scénář 2 - Grafana dashboard



Obrázek 1.4 Scénář 3 - Grafana dashboard

Obrázek 1.5 Scénář 4 - Grafana dashboard



Obrázek 1.6 Scénář 5 - Grafana dashboard