UNICORN VYSOKÁ ŠKOLA S.R.O.

Softwarový vývoj

BAKALÁŘSKÁ PRÁCE

Retrieval – augmented generation a kontextuální dialog nad vysoce dostupnými velkými daty

Autor BP: Petr Hejl

Vedoucí BP: Ing. Jan Kovář

Čestné prohlášení

Prohlašuji, že jsem svou závěrečnou práci na téma Retrieval – augmented generation a kontextuální dialog nad vysoce dostupnými velkými daty vypracoval samostatně pod vedením vedoucího závěrečné práce a s použitím výhradně odborné literatury a dalších informačních zdrojů, které jsou v práci všechny citovány a jsou také uvedeny v seznamu použitých zdrojů. Prohlašuji, že nástroje umělé inteligence byly využity pouze pro podpůrné činnosti a v souladu s principem akademické etiky.

Jako autor této závěrečné práce dále prohlašuji, že v souvislosti s jejím vytvořením jsem neporušil autorská práva třetích osob a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb.

Dále prohlašuji, že odevzdaná tištěná verze závěrečné práce je shodná s verzí, která byla odevzdána elektronicky.

V Praze dne 11.07.2025

Petr Hejl

Poděkování

Děkuji vedoucímu závěrečné práce Ing. Janu Kovářovi za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé závěrečné práce.

Retrieval – augmented generation a kontextuální dialog na vysoce dostupnými velkými daty

Retrieval – augmented generation and contextual dialogue over highly available big data.

UNICORN
UNIVERSITY

Abstrakt

Cílem práce je analyzovat stávající stav v oblasti softwarového vývoje komponent implementujících prvky umělé inteligence. Analýza je zaměřena v první řadě na distribuované databázové systémy a možnosti uložení a vyhledávání dat za použití natural language procesingu. Dále je analyzována situace v oblasti vývoje aplikační vrstvy, programovací jazyky a frameworky zaměřující se na problematiku umělé inteligence, knihovny pro embedding textu a pro komunikaci s vlastními jazykovými modely či službami třetích stran. Analýza je zakončena průzkumem nástrojů a aplikací pro koncové uživatele a popisem současných trendů v oblastech s narůstajícím využitím umělé inteligence. Součástí práce je i vlastní implementace aplikace typu Retrieval-Augmented Generation vytvořené na základě předchozí analýzy a porovnání výkonu AI modelů s ohledem na výsledky a výpočetní náročnost.

Klíčová slova: retrieval – augmented generation, large language models, text embedding, natural language procesing, Python, Elastic

Abstract

The aim of this thesis is to analyze the current state of software development for components implementing artificial intelligence elements. The analysis primarily focuses on distributed database systems and the possibilities of storing and retrieving data using natural language processing. Additionally, it examines the development of the application layer, programming languages, and frameworks that address artificial intelligence challenges, as well as libraries for text embedding and communication with proprietary language models or third-party services. The analysis concludes with an exploration of tools and applications for end users and a description of current trends in areas with increasing AI adoption. The thesis also includes the implementation of a Retrieval-Augmented Generation (RAG) application based on the preceding analysis, along with a performance comparison of AI models concerning both results and computational demands.

Keywords: retrieval – augmented generation, large language models, text embedding, natural language processing, Python, Elastic

Obsah

Úvod

Tato práce se zaměřuje na problematiku vývoje aplikací typu Retrieval-Augmented generation. Jejím cílem je analýza stávajícího stavu v oblasti softwarového vývoje komponent implementujících prvky umělé inteligence a implementace prototypu takové aplikace.

Práce je rozdělena do teoretické a praktické části, kdy teoretická část se věnuje analýze stávajícího stavu vývoje AI komponent s ohledem na jejich využitelnost při vývoji aplikací. Je popsána problematika, jsou vysvětleny základní pojmy a principy spolu s konkrétními příklady současných modelů. Následně se tato část věnuje analýze stávajících databází a podpoře AI problematiky v oblasti uložení dat způsobem vhodným pro následné využití komponentami aplikací typu RAG. Dále je rozpracována analýza současného stavu podpory oblasti AI napříč programovacími jazyky s velkou mírou zastoupení v komerční a vědecké sféře, jsou popsány aktuální frameworky a knihovny a způsob práce se strukturovanými daty na úrovni aplikační vrstvy s napojením na zájmové modely umělé inteligence. Teoretickou část završuje analýza soudobých trendů v oblasti vývoje frontendu, chatbotů a AI asistentů s ohledem na uživatelskou zkušenost.

Praktická část práce má za cíl tvorbu prototypu aplikace RAG (Retrieval-Augmented Generation). Jedná se o třívrstvou architekturu, kdy v první fázi je implementována vhodná databáze, navržena datová struktura a procesy ukládání a vyhledávání dat využívající zpracování přirozeného jazyka (dále NLP z Natural Language Processing). Druhou fází je tvorba aplikační vrstvy s vhodným komunikačním rozhraním (API) pro aplikace koncových uživatelů. Tato vrstva je modulární a důraz je kladen na konfigurovatelnost a možnost snadné tvorby a zapojení vlastních modulů pro integraci dalších jazykových modelů softwarovými vývojáři a datovými či AI inženýry. Aplikace ve výchozí konfiguraci podporuje využití veřejně dostupných jazykových modelů. Třetí fáze představuje prototyp forntendové aplikace pro koncové uživatele a její využití je oproti aplikační vrstvě omezeno na vyhledávání zájmových dat a kontextuální dialog nad nimi bez nutnosti jejich předchozí znalosti nebo orientace v datech uživatelem.

Aplikace umožní snadné porovnávání výkonu a výsledků libovolných jazykových modelů při zachování snadné a intuitivní konfigurovatelnosti a ladění chování. Součástí aplikace je rovněž předpis konfigurace pro nasazení ve virtualizovaném prostředí s možností okamžitého horizontálního škálování, jak je standardem v enterprise prostředích, a příklad takového nasazení.

Principy Retrieval-Augmented Generation, AI a způsoby jejich užití ve vývoji aplikací

1 Základní pojmy a principy

V oblasti využití umělé inteligence v souvislosti s natural language procesingem a jeho použití nad velkými daty se setkáváme se sadou odborných pojmů a principů, jejichž správné pochopení je nezbytné pro porozumění principům a logiky celého procesu.

1.1 Big Data

Jedná se o obecnější pojem, překládaný do českého jazyka doslovně jako „velká data“. Na otázku „Která data jsou již velká a která ještě nikoliv“ není jednoznačná odpověď, tato se navíc mění v čase s ohledem na narůstající kapacitu komerčně dostupných úložišť. Zatímco na hraně třetího tisíciletí mohl být datový objem 1 GB považován za velká data, v současnosti jsou dostupná distribuovaná úložiště o celkových kapacitách v petabytech či exabytech. Charakter samotných dat je navíc v čase rovněž proměnný, kdy v současnosti takto velké datové sady mohou zahrnovat korporátní klientská data, businessová data, interní směrnice a rozhodnutí výkonných orgánů, provozní, bezpečnostní či auditní data, mapy, systémové a aplikační logy, výkonnostní logy, senzorická data či logy zařízení z Internetů věcí. Z uvedeného vyplývá, že tato data jsou rozličná i svou strukturou, mnohdy i v rámci stejné datové sady, proto se v otázce jejich uložení často používá vlastností objektových databází, kdy přední z nich v současnosti disponují pokročilými funkcemi vyhledávání nad různými typy vektorů, jedním z principů natural language procesingu.

1.2 Objektové databáze

Objektové databáze nalézají použití v oblasti velkých dat a strojového učení zejména kvůli míře volnosti struktury dat, která jsou schopny uložit. Tradiční relační databáze lze sice použít a podpora vyhledávání nad vektory je v některých rovněž přítomna, avšak v praxi se často setkáváme s tím, že struktura zájmových dat a někdy ani jejich přibližný obsah nejsou v čase budovaní a integrace datových pipelines známy, a pro vývoj aplikace poskytující funkcionality RAG tak nelze snadno navrhnout datovou vrstvu. V objektových databázích tento problém do značné míry odpadá, byť ne vždy zcela (dochází-li například k mapování klíčů uložených objektů, je třeba zohlednit možné kolize datových typů v jejich hodnotách napříč dokumenty). Z objektových databází pak nachází nejširší uplatnění clusterové dokumentové

databáze [1], jmenovitě například Elasticsearch, MongoDB, Couchbase či proprietární databáze poskytované v rámci modelů PaaS.

1.3 AI – Umělá inteligence

Artificial intelligence, umělá inteligence, nalézá v posledních letech široké uplatnění, kdy významným milníkem vedoucím k umožnění jejího využití ve vývoji aplikací širokému spektru vývojářů a datových inženýrů jakož i k popularizaci AI jako služby bylo uvedení produktu ChatGPT z dílny OpenAI na trh dne 30. listopadu 2022 [2]. Od té doby lze zaznamenat exponenciální nárůst modelů, knihoven, publikací, návodů i vědeckých prací v této problematice. Termín „umělá inteligence“, poprvé použit v roce 1950 Johnem McCarthym, je velmi obecný a užívá se jak k označení konkrétních modelů, tak k procesům jejich učení, vývoje, užívání i poskytování, kdy společným jmenovatelem je simulace lidského chování stroji a algoritmy. AI modely jsou trénovány k různým účelům, avšak mezi nejpoužívanější patří v současnosti modely zpracovávající a generující grafický obsah a zejména text (tzv. LLM) [3]. V rámci zpracování textu pak můžeme při pohledu shora rozlišovat zejména modely pro text embedding (sentence-similarity modely) a chatovací modely pro kontextuální konverzaci v rámci znalostní báze a definovaného časového okna, kam patří obsah dané konverzace samotné.

1.4 AI modely

AI modely se svým zaměřením liší napříč odvětvími lidské činnosti a jejich rozdělení závisí toliko na úhlu pohledu. Z hlediska jejich učení je můžeme dělit na modely užívající strojové učení, učení s učitelem, učení bez učitele a deep learning modely [4]. Další možnost dělení je dle jejich účelu na LLM, difuzní modely, GAN (Generative Adversarial Networks), VAE (Variational Autoencoders), transformery, multimodální modely či modely základní (tedy obecně algoritmy vykonávající úkoly, pro něž je třeba lidská inteligence).

1.4.1 Strojové učení (ML – machine learning)

K tomuto trénování modelů umělé inteligence jsou užívány datové sady s označeními dat, bez označení dat nebo jejich kombinace. Klasifikační či regresní algoritmus je následně použit k samotnému učení, kdy klasifikační algoritmy rozeznávají datové struktury a rozhodují o jejich označení, regresní algoritmy jsou pak užívány k predikci a hledání závislost či nezávislosti proměnných v datové sadě. Výsledkem jsou modely se schopností identifikovat charakter nových dat (například rozeznat fotografii vozidla od květiny).

1.4.2 Učení s učitelem

K nejčastějšímu způsobu trénování modelů patří učení s učitelem [4]. Algoritmus je trénován na datové sadě vytvořené a s daty označenými člověkem, kdy dle značek model rozeznává a kategorizuje data dle přání tvůrce datové sady. Na základě srovnávání velké množství vstupů s očekávanými výstupy se modely učí vzorce, vztahy a způsoby predikce. Výsledkem jsou modely, které z nových vstupů generují výstupy a predikce podle naučené logiky.

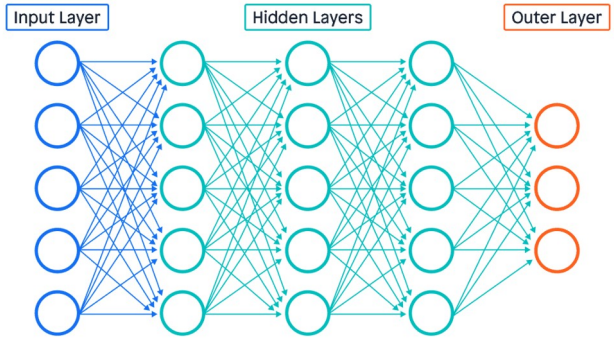
1.4.3 Učení bez učitele

Tento způsob trénování modelů spočívá ve vkládání datových sad s neoznačenými daty. Vazby, vzorce a podobnosti jsou následně detekovány algoritmicky bez další účasti člověka a bez explicitních předpisů, jak mají být data kategorizována. Výsledkem jsou modely se schopností dělení a kategorizace nových vstupních dat (například rozdělení koček na fotografiích dle barvy či plemene a jejich odlišení od stromů).

1.4.4 Deep learning

Jedná se o pokročilou formu trénování modelů k vyhledávání složitých vzorců v textu. Klasifikace probíhá na několika vrstvách s rozdílným účelem. Vstupní vrstva slouží k příjmu vstupních dat a jejich distribuci do vrstvy další. Následuje několik na sebe navazujících skrytých vrstev sloužících k postupnému zprocesování dat a genezi výstupu. Na výstupní vrstvu jsou následně doručena již hotová výstupní data. Množství skrytých vrstev se pohybuje od dvou do stovek vrstev, kdy limitující je zejména kapacita neuronové sítě, na níž je učení realizováno. Typickým zástupcem takovýchto modelů jsou právě LLM.

Obrázek 1: Schematické znázornění vrstev Deep learning procesu



Zdroj: https://www.mendix.com/wp-content/uploads/image_Different-AI-Models.webp

1.4.5 LLM

LLM – Large Language Model, je označení modelů umělé inteligence, které jsou trénované k simulaci přirozené konverzace s člověkem v rámci NLP. Jde o oblast počítačové vědy užívající mechanismy strojového učení k umožnění komunikace lidským způsobem. Mezi hlavní přednosti tohoto mechanismu patří zejména automatizace opakujících se úloh, lepší analýza a vhled do zpracovávané problematiky (v rámci konverzace), pokročilé prohledávání a porozumění textu a geneze obsahu [5]. Komerční LLM jsou v současnosti dostupné pro koncové zákazníky cestou RESTových API pro napojení vlastních aplikací nebo cestou zcela hotových aplikací (chatbot v rámci zákaznické podpory apod.). Za účelem vývoje vlastní RAGové aplikace je k dispozici řada jazykových modelů s možností provozu on-premise, kdy mezi nejčastěji užívané patří DeepSeek, Meta-Llama-3.8B či bloom [6].

1.4.6 Difuzní modely

Jedná se o generativní modely, které vytváří obsah tak, že v iteracích třídí náhodný šum do doby, než jeho charakter odpovídá požadovanému výstupu, tedy reverzní operaci k té, na níž jsou trénovány, kdy vstupní grafický obsah náhodným šumem rozptylují. V současné době jsou používány zejména ke genezi obrázků s vysokým rozlišením a AI umění (souhrnný termín pro umělecká díla zhotovená AI modelem). DALL-E 2, Midjourney či open source Stable Diffusion patří k zástupcům této kategorie [7].

1.4.7 Generative Adversarial Networks

Modely z kategorie GAN sestávají ze dvou nezávislých neuronových sítí označovaných jako „generator“ a „discriminator“. Funkce generátoru spočívá v genezi vzorkových dat z náhodného šumu, zatímco úlohou diskriminátoru je platnost či neplatnost těchto dat ověřit, kdy rozlišuje mezi daty z datové sady (autentická data) a daty generovanými. Tento typ modelů představil Ian Goodwellow 10. června 2014 [8] a uplatňuje se zejména v syntetizaci grafického obsahu – obrázků a videí, dále ve zvyšování rozlišení obrazu s nízkou kvalitou, záměnu tváří ve videích v reálném čase, genezi tréninkových datových sad, zlepšování kvality obrazu v oblasti medicíny (rentgenové snímky apod.) i pro genezi módních návrhů.

1.4.8 Variational Autoencoders

Modely z kategorie VAE jsou trénovány ke kódování vstupních dat do latentního prostoru a jejich zpětné dekódování do původního obsahu. Latentním prostorem zde rozumíme abstraktní reprezentaci vstupních dat, kde podobné hodnoty jsou mapovány blízko sebe. Tyto modely sestávají ze dvou hlavních komponent, obdobně jako modely GAN, jejich role se však liší. V případě VAE modelu existuje Encoder kódující vstupní data do latentního prostoru za použití distribuční funkce. Jeho výstupem jsou dva vektory – význam a odchylka, z nichž vychází výsledná distribuce hodnot. Decoder pak rekonstruuje původní objekty.

1.4.9 Transformační modely – Transformery (Transformer-Based)

Předností modelů z této kategorie je zejména jejich přínos v oblasti NLP a počítačového vidění (schopnosti počítače získávat informace z vizuálního obsahu používané v autonomních vozidlech či průmyslových robotech nebo detekce jevů na kamerových záznamech). Princip představil 12. června 2017 Ashish Vaswani [9] a tvoří základ modelů jako ChatGPT, FLAN-T5 či BERT. Tyto modely sestávají z několika základních komponent: self-attention – mechanismus umožňující zaměření na odlišné části vstupní sekvence v rámci zpracování slov, multi-head attention pro vícenásobné zachycování vzorců v datech, poziční kódování pro zachování pořadí slov i při paralelním zpracování a normalizaci vrstev pro čitelný výstup. Normalizace se používá zejména pro stabilizaci trénování, tak, aby se typicky gradient-descent udržel pod kontrolou. Genialita transformeru spočívá v tom, že vše probíhá paralelně, tedy vstup se nemusí zpracovávat slovo po slovu, ale všechny slova se analyzují v závislosti na všech ostatních slovech vstupu vedle sebe.

Tabulka 1: Transformační modely dle typu

Typ	Model	Účel
Encoder-only	BERT	Porozumění textu
Decoder-only	ChatGPT-4, Llama	Geneze textu, chatboti
Encoder-Decoder (také Seq2seq)	T5, BART	Překlady, shrnutí textů
Vizuální transformery	ViT, DINO	Detekce a klasifikaci objektů v obrazu
Multimodální transformery	CLIP, Gemini	Zpracování textu a obrazu

Zdroj: vlastní zpracování

1.5 Embedding

V oblasti AI je tento termín používán k pojmenování vektorového vyjádření smyslu částí textu nebo jednotlivých výrazů. Existuje více typů takových vektorů, kdy mezi nejpoužívanější se řadí typy dense vector a sparse vector. Dense vector je pole vektorů reprezentované sadami čísel, kdy každé představuje pozici na ose v relevantní dimenzi. Těmito dimenzemi lze rozumět referenční významovou hodnotu a míru jejího zastoupení pro konkrétní výraz v konkrétním lidském jazyce, kdy tato reprezentace je pro každý model odlišná a ani případný stejný počet dimenzí tak nevypovídá o míře zaměnitelnosti jednotlivých modelů. Množství těchto dimenzí se standardně pohybuje ve stovkách, ale výjimkou nejsou ani embeddovací modely generující vektory s více než tisíci dimenzemi (např. Cohere embed-english-v3.0 [10]). Sparse vector pak má podobu sady dokumentů se strukturou klíč – hodnota, kdy klíčem jsou referenční významové prvky zastoupené v textu či odhalená synonyma, hodnotami jsou pak čísla zastupující míru tohoto dílčího významu, obdobně jako je tomu v případě dense vectoru.

1.6 Frameworky, rozhraní a dostupnost modelů v oblasti vývoje aplikací

Komerčně poskytované jazykové modely v současné době zpravidla neumožňují jejich snadné použití k vývoji vlastní aplikace zejména s ohledem na fakt, že přístup k modelu cestou API není obvykle obsažen v základní licenci, byť například platforma Cohere tuto možnost nabízí [11]. Neméně důležitým aspektem při tvorbě vlastní aplikace RAG je pak otázka bezpečnostní. Bude-li aplikace pracovat s citlivými uživatelskými, businessovými nebo bezpečnostními daty, je ke zvážení otázka dostupnosti jazykových modelů s možností provozu on-premise či kapacita k vytvoření a vytrénování vlastního modelu. Přední soudobou svobodnou platformou umožňující sdílení AI modelů napříč společnostmi i jednotlivci nejen s možností jejich provozu na vlastní infrastruktuře, ale rovněž poskytující společný framework pro jejich intuitivní a relativně snadné použití, je platforma Huggingface. Sdílené knihovny tohoto frameworku jsou psány a poskytovány pro jazyk Python, který se rovněž vyprofiloval jako primární programovací jazyk v oblasti AI obecně. Nejen s touto platformou pak souvisí další framework pro jazyk Python – LangChain, jenž je vyvíjen primárně za účelem abstrakce nad operacemi souvisejícími s jazykovými AI modely a sjednocení způsobů jejich použití. Dostupná je i varianta pro Javascript. Python a Langchain jsou pak typickými komponentami zastoupenými ve většině aplikací typu RAG. Vývoj aplikací využívajících AI však není těmito nástroji omezen, z dalších rozšířených lze zmínit knihovny Weka, Deeplearning4j, MOA či Java-ML pro jazyk Java. Nezanedbatelnou výhodou všech výše uvedených komponent je

rovněž jejich platformní nezávislost a z toho plynoucí možnost jejich provozu v libovolném prostředí včetně virtualizovaných, často škálovaných cloudových řešeních.

1.7 RAG

Retrieval-Augmented Generation je označení pro aplikaci využívající některé nebo všechny výše zmíněné prvky k vytvoření souvislého proudu informací od dat uložených v databázi přes embedding a LLM až ke koncovému uživateli bez nutnosti dopředné znalosti těchto dat uživatelem nebo i základní orientace v nich. Smyslem takové aplikace je umožnit uživateli rychlé získání co nejrelevantnějšího obsahu k jeho dotazům a dále pak poskytnutí kontextu nad daty, orientace v nich a vhled v souvislostech za udržení konverzace lidského typu. Aplikace tohoto typu jsou v současnosti vyvíjeny zejména s ohledem na fakt, že společnosti a instituce disponují velkými datovými objemy za dlouhé časové úseky (často i v jednotkách či desítkách let) bez schopnosti se v těchto datech efektivně orientovat a získat z nich co nejvyšší informační hodnotu v požadovaném čase, případně objem takových dat roste tak vysokou rychlostí, že průběžná orientace v nich je mimo možnosti člověka.

1.8 Měření výkonu a benchmarking

S ohledem na narůstající množství dostupných modelů získává pro firmy a instituce na váze otázka poměru ceny a výkonu těchto modelů, kdy pro objektivní porovnání užíváme tzv. Benchmarky. Měření výkonu LLM je tedy realizováno prostřednictvím standardizovaných testů. Obvykle se jedná a datovou sadu, sadu otázek či úkolů a metodu přiřazování bodů obvykle na škále od 0 do 100 [12]. Mezi aktuální populární nástroje se pak řadí ARC (testující uvažování a znalostní bázi formou otázka – odpověď), HellaSwag (racionální uvažování a porozumění přirozenému jazyku cestou dotazování na scénáře z fyzického světa snadno zodpověditelné člověkem) či MMLU (porozumění jazyku a následné řešení úloh ze znalostní báze) [12].

1.8.1 LLM leaderboard

LLM leaderboard slouží ke vzájemnému porovnání napříč modely. Jedná se o seznam výsledků různých benchmarků pro každý model. K dispozici jsou nezávislé platformy poskytující přehledy napříč benchmarky, jakož i přehledy tvořené a udržované designéry jednotlivých nástrojů. Mezi nezávislými platformami najdeme například komparativní srovnání

na Huggingface [13] nabízející přehled modelů LLM, embeddovacích modelů, Chatbotů, žebříčky dle výkonu či srovnání modelů rozeznávajících lidskou řeč.

1.8.2 Problémy s LLM benchmarkingem

Ačkoliv jsou benchmarkovací nástroje cennými pomocníky při porovnání modelů a kvantifikaci atributů jejich chování, jejich vypovídací hodnota není absolutní z několika důvodů. Hlavní překážkou k tomu je fakt, že modely lze trénovat na datových sadách, které používají tyto samotné nástroje, čímž se míra přesnosti výsledků zcela vychýlí, aniž by přitom odpovídala skutečné schopnosti modelu řešit zadané úkoly. Toto je tím častější problém, čím rozšířenějším se daný benchmarkovací nástroj stane. Dalším problémem je omezená vypovídací hodnota benchmarkovacích pravidel ve vztahu k reálnému světu, který těmito není omezen a obsahuje oproti sterilnímu prostředí testovacích nástrojů řadu deviací. Řadu aspektů, jmenovitě spontánnost lidské konverzace, jejíž směr a charakter se mohou kdykoliv změnit, navíc nelze se současnou mírou poznání snadno do takových pravidel převést, jelikož každá trénovací sada je z logiky věci omezenou množinou, zatímco spontánnost lidské konverzace nelze snadno (a možná vůbec) kvantifikovat.

1.9 Etika, autorství a další právní aspekty

V reakci na rychlý rozvoj oblasti AI dochází ke vzniku řady nových právních otázek v oblasti autorství, rozhodovacích oprávnění i otázek etických. „Zákon striktně říká, že autorem jakéhokoliv autorského díla je fyzická osoba.“ [14] Jelikož žádný AI model fyzickou osobou není, nelze tak k němu vztahovat autorství jakéhokoliv díla. I v českém právním prostředí se již vyskytly spory ohledně autorství vizuálního obsahu vygenerovaného na základě textového vstupu uživatele, kdy Městský soud v Praze „...došel k závěru, že obsah vygenerovaný umělou inteligencí nelze považovat za autorské dílo, jelikož nesplňuje pojmové znaky dané autorským zákonem. Jedná se zejména o to, že není jedinečným výsledkem tvůrčí činnosti autora.“ [14]

Avšak kromě otázky autorství existují spory přesahující otázku práva a pokrývající rovněž oblasti etiky či filozofie, kdy se setkáváme s úvahami, zda model může při tvorbě obsahu diskriminovat osoby na základě naučených principů z tréninkových datových sad či zda lze umělé inteligenci svěřit rozhodování o usmrcení osob ve válečných konfliktech při použití v autonomních zbraních. Tato obsáhlá problematika je však mimo rámec této práce.

2 Vývoj RAGových aplikací

Vývoj aplikací typu RAG není vázán na jedinou platformu, knihovnu, programovací jazyk ani databázový systém, nicméně míra dostupných vývojářských prostředků se napříč jazyky a dalšími systémy liší.

RAGové aplikace jsou zpravidla budovány na klasické třívrstvé architektuře, kdy datová vrstva je realizována cestou databáze s podporou datových struktur užívaných k vyhledávání embeddovaných dat (vektory). Aplikační vrstvu pak představuje samotná aplikační logika, kde je realizován embedding dat (v některých případech lze toto realizovat již v datové vrstvě nebo oba přístupy vhodně kombinovat), jejich další zpracování a napojení na model umělé inteligence, který může být provozován lokálně nebo může jít o službu třetí strany. Rozhraní na této vrstvě má zpravidla podobu REST API, případně API prostřednictvím Websockets pro lepší podporu datových proudů a zobrazování odpovědí již v průběhu jejich geneze. Pro koncové uživatele pak existuje frontendová vrstva mající zpravidla podobu webové, desktopové nebo mobilní aplikace.

2.1 Datová vrstva

Optimálně navržená datová vrstva je nezbytností pro jakoukoliv aplikaci operující nad velkými daty. Pro její relizaci se v enterprise prostředí zpravidla používají clusterové databáze, a to jak relační, tak objektové podle na konkrétního určení. Clusterové databáze pracují na principu nodů sdružených do jednoho celku, kdy distribuce dat jakož i řízení dalších funkcí clusteru jako přidávání či odebírání nodů je určováno řídícím nodem. Pro vysokou dostupnost dat a scénáře typu disaster recovery disponují současné clusterové databáze mechanismy typu node allocation awareness, jenž zajišťují replikaci dat a jejich fyzické umístění na odlišných fyzických strojích. Tato funkce je mimo jiné užívána v oblasti virtualizace s dynamickým škálováním, kdy mnoho virtuálních nodů může být fyzicky spuštěno na jednom fyzickém stroji a je třeba zajistit jejich optimální rozmístění napříč infrastrukturou pro případ selhání hardware. Takto připravená datová vrstva tvoří ideální základ pro bezproblémový běh RAGové aplikace a dostupnost dat pro uživatele i v případě výpadku či odstávky některých zařízení.

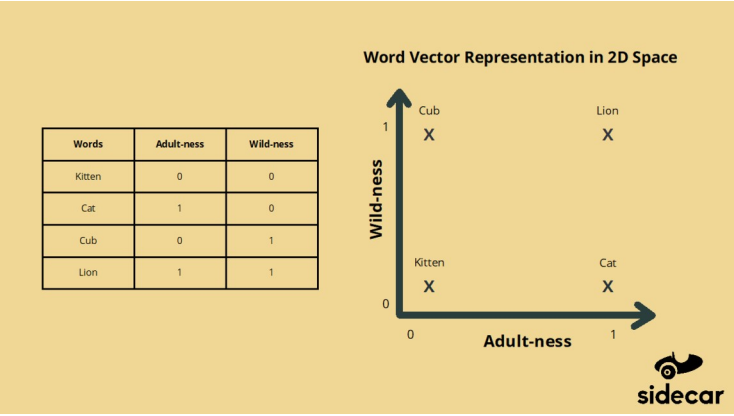
2.1.1 Embeddingy a vektory

V kapitole 1.5 jsem se embeddingu již věnoval, v této sekci uvažujme konkrétní případy a příklady, jak může být embedding realizován na datové vrstvě.

V oblasti ukládání a vyhledávání embeddovaných dat se nejčastěji setkáváme s typem dense vector. Z hlediska databáze se jedná o pole. Myšlenka dense vectoru spočívá v pojetí pevného počtu významových slov konkrétního lidského jazyka jako dimenzí a přiřazování hodnot dalším slovům v prostoru tvořeném těmito dimenzemi. Z uvedeného plyne, že různé embeddingové modely nejsou vzájemně zaměnitelné, a to ani v případě, že by měly stanovený stejný počet dimenzí, stejně jako fakt, že tyto modely se liší mírou přesnosti pro různé lidské jazyky. Výstupem takového modelu bude vektor typu `[1 34 321 …]`, kdy čísla představují koeficienty shody s referenčními významy. Při vyhledávání pak databázi zasíláme vstupní data – zpravidla vstup uživatele po provedení embeddingu – a míru volnosti či rozptylu, s jakým chceme vyhledávat, tedy určujeme požadovanou míry shody. Logicky pak dovodíme, že větší míra volnosti povede k většímu množství výsledků, ale menší míře relevance k uživatelskému vstupu.

Mezi další užívané typy řadíme sparse vector. Zde se jedná o seznam dvojic klíč – hodnota, kdy myšlenka je analogická k dense vectoru. Klíče zde reprezentují referenční významy, hodnotou je pak neceločíselný údaj o shodě. Namísto řady čísel pak dostáváme pole těchto dvojic typu `[{2:0.2}, {5768:0.5}, …]`. Některé modely pak jako klíče uvádí samotné jazykové výrazy, jako `[{„king“:0.9}, {„banana“:0.1}, …]`.

Obrázek 2: Příklad reprezentace významu slov 2 dimenzionálním dense vectorem



Zdroj: https://sidecar.ai/blog/demystifying-vectors-and-embeddings-in-ai-a-beginners-guide

Embedding textu se neomezuje pouze na výše uvedené typy vektorů. Zastoupení nalézají i další:

- Binary vector – zde má každá dimenze hodnotu 0 nebo 1 (některé modely uživají hodnoty -1 a 1), výhodou tohoto vektoru je rychlost jeho geneze za cenu menší přesnosti

při vyhledávání, užití nalézá v locality-sensitive hashing (LSH) – metoda hashování podobných vstupů do stejných „bucketů“, kterých je výrazně méně než možných vstupů.

- Quantized vector – jedná se o podkategorii dense vectoru se sníženou mírou přesnosti užívaného ke snížení paměťových nároků například v modelu FAISS.

- Multi-vector – textový vstup je zde reprezentován více vektory namísto jediného, používají jej například modely ColBERT či DPR.

- Poziční a strukturované vektory – složité modely kombinující význam vstupu s pozicí či strukturou v syntaktických stromech nebo grafech závislostí.

2.1.2 Analýza současného stavu vývoje a podpory problematiky AI v oblasti databází

Při výběru a porovnání databázových systémů podporujících uložení datových struktur snadno využitelných modely umělé inteligence uvažujeme zejména možnost uložení vektorů s vysokou hodnotou dimenze a porovnáváme výkon a efektivitu jejich prohledávání. Stávající databázové systémy již disponují solidní podporou těchto struktur, některé mají řadu souvisejících funkcí implementovanou v sobě a jsou schopny obstarat embedding dat přímo v rámci jejich ukládání v rámci takzvaných inference pipelines. Jiné databáze spoléhají na tvorbu embeddingu na aplikační vrstvě a data pak ukládají standardním způsobem. Tyto funkce jsou v současnosti dostupné ve většině předních databázových systémů, a to jak relačních, tak objektových.

2.1.2.1 PostgreSQL

Jedna z předních relačních databází používaná v enterprise prostředí, PostgreSQL disponuje pluginem pvgector [15] s funkcí vector similarity. Mezi datové typy přidává „vector“ s určením dimenze, kdy se jedná o dense vector. Ukládání pak probíhá standardním způsobem, vyhledávání nejbližších dokumentů užívá syntaxi `SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;` S následujícími možnými operátory [15]:

- <-> - L2 distance
- <#> - (negative) inner product
- <=> - cosine distance
- <+> - L1 distance
- <~> - Hamming distance (binary vectors)

- <%> - Jaccard distance (binary vectors)

Příklad uvádí vektor o třech dimenzích, čímž není datový typ vector limitován.

2.1.2.2 MariaDB

V relační databázi jejíž tvůrci prezentují inovace a moderní funkce jako jednu z jejích hlavních výhod, najdeme intuitivní použití vektorového vyhledávání prostřednictvím připravených funkcí nad datovým typem VECTOR [16]:

```
CREATE TABLE products (
    name varchar(128),
    description varchar(2000),
    embedding VECTOR(4) NOT NULL,
    VECTOR INDEX (embedding) M=6 DISTANCE=euclidean)
ENGINE=InnoDB;
```

Vytvoření tabulky s datovým typem VECTOR o 4 dimenzích.

```
INSERT INTO products (name, description, embedding)
VALUES ('Coffee Machine',
        'Built to make the best coffee you can imagine',
        VEC_FromText('[0.3, 0.5, 0.2, 0.1]'))
```

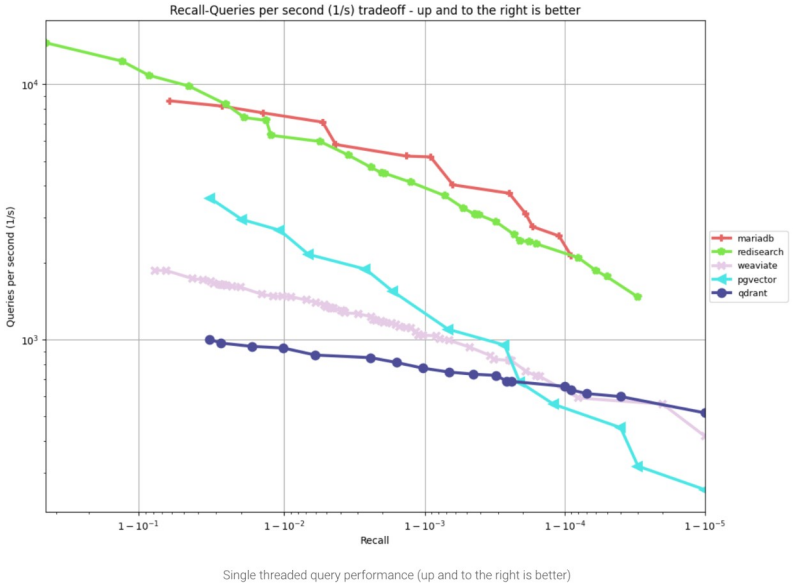
Vložení hodnot.

```
SELECT p.name, p.description
FROM products AS p
ORDER BY VEC_DISTANCE_EUCLIDEAN(p.embedding,
                    VEC_FromText('[0.3, 0,5, 0.1, 0.3]'))
LIMIT 10
```

Vyhledávání záznamů.

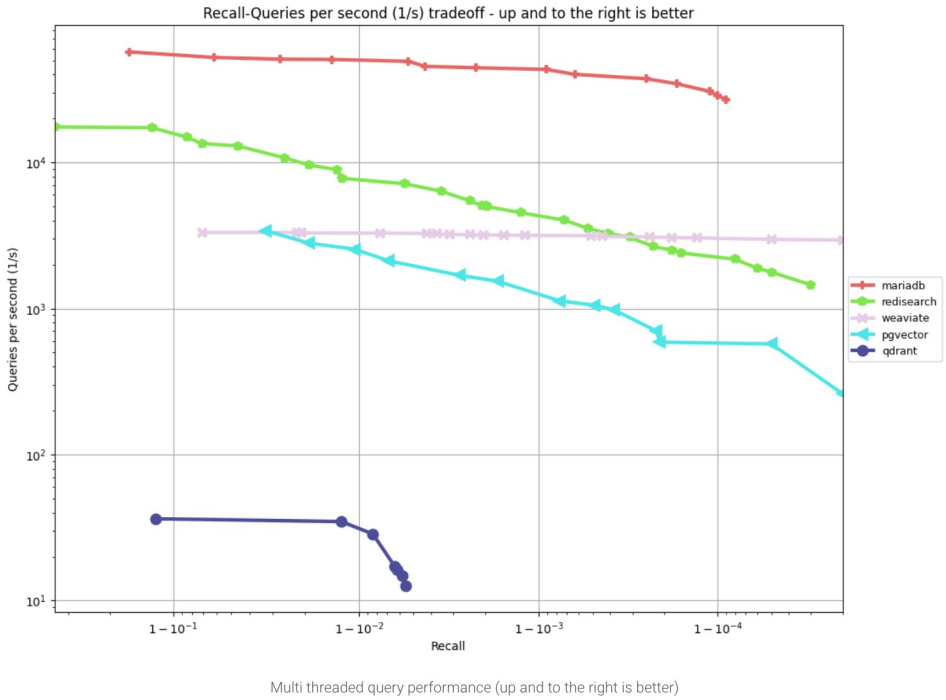
Obdobnou podporu vektorového vyhledávání nalezneme i v dalších relačních databázích jako Oracle či MySQL s různou mírou výkonu.

Graf 1 – Srovnání výkonu vybraných vektorových modulů v relačních databázích bez paralelizace



Zdroj: https://mariadb.org/projects/mariadb-vector/

Graf 2 – Srovnání výkonu vybraných vektorových modulů v relačních databázích s paralelizací



Zdroj: https://mariadb.org/projects/mariadb-vector/

2.1.2.3 MongoDB

Jako jedna z nejznámějších objektových databází dokumentového typu nalézá MongoDB zastoupení na datové vrstvě aplikací využívající flexibilní datová schémata. Mezi její další výhody patří vysoká propustnost dat, kdy tato vlastnost je využívána v prostředích, kde dochází k rychlému zapisování dat ve velkých objemech. Aktuální verze disponuje podporou vektorového vyhledávání, jehož použití je velmi intuitivní [17]:

```
{
  "$vectorSearch": {
    "exact": true | false,
    "filter": {<filter-specification>},
    "index": "<index-name>",
    "limit": <number-of-results>,
    "numCandidates": <number-of-candidates>,
    "path": "<field-to-search>",
    "queryVector": [<array-of-numbers>]
  }
}
```

Jako ostatní zde uvedené databázové systémy, MongoDB disponuje vlastní knihovnou pro jazyk Python *pymongo* s parametrizovanými metodami umožňujícími intuitivní dynamické sestavování vyhledávacích dotazů.

2.1.2.4 Elasticsearch

Jedna z předních a nejvýkonnějších clusterových objektových dokumentových databází v enterprise oblasti je součástí širší sady nástrojů Elastic stack tvořících vzájemně propojený ekosystém. Jde o sadu snadno škálovatelných, velmi výkonných komponent, jejichž středobodem je tato databáze. Pro svou flexibilitu, množství funkcí a granularitu, s níž je možné jednotlivé komponenty konfigurovat a jejich chování dramaticky měnit, je její nasazení v produkčních prostředích náročnější, základní nasazení je však intuitivní a nenáročné. Ačkoliv se jedná o enterprise-grade nástroj, velké množství pokročilých funkcí je dostupné pod základní nezpoplatněnou licencí a relevantní část kódu je dostupná jako open source, čehož využívají některé vývojářské týmy k jejímu vlastnímu rozšiřování. V posledních letech zde probíhá dramatický nárůst podpory funkcionalit umělé inteligence, z nichž některé jsou do Elasticsearch implementované jako vlastní součást, jiné je možné integrovat nebo napojit jako službu další strany. Podpora AI v Elasticsearch tak nekončí pouhým vyhledáváním nad dense nebo sparse vectorem. Jednotlivé nody clusteru lze přímo dedikovat pro strojové učení, další dedikované komponenty mohou stahovat celé weby dle flexibilních pravidel a provádět embedding s několika možnými formami výstupů. Nody dedikované pro roli *ingest* pak mohou z již stávajících dat vygenerovat podle předpisů uživatele nové indexy s daty obohacenými o embedding, aniž by za tímto účelem musela existovat aplikační vrstva.

Elastic disponuje sadou vlastních AI modelů pro embedding a reranking, které lze kombinovat v takzvaných inference pipelines a jejich výsledky sdružovat v konečné podobě dokumentů, což nevylučuje předchozí embedding na aplikační vrstvě, jenž lze také přidružit. Výsledný dokument tak může obsahovat celou řadu embeddingů. Elasticsearch následně umožňuje vyhledávat podle libovolného z těchto embeddingů či sloučit několik vyhledávacích výrazů do jednoho, specifikovat různé priority výsledků, sloučit výsledek s fulltextovým vyhledáváním a s použitím rerankingu tak dostat nejrelevantnější možný obsah. Vyjma zájmových klientských či businessových dat se rovněž rozšiřuje použití Elastic stacku za účelem sledování výkonu a bezpečnosti celých infrastruktur a systémů, kdy typickým příkladem je sběr bezpečnostních, auditních, aplikačních či systémových dat, jejich embedding na nodech dedikovaných pro machine learning již v procesu uložení na datové nody skrz inference pipeline, dále AI detekce anomálií a napojení na externích či lokálních LLM, kdy data před odesláním do těchto modelů lze anonymizovat dle specifických potřeb uživatele. Bezpečnostní a monitorovací týmy tak nejen mohou detekovat anomálie v reálném čase, ale

mohou jejich smysl, vývoj a mitigaci konzultovat s LLM vytrénovaným pro tento účel, aniž by sdíleli citlivé informace, či predikovat jejich vývoj v budoucnu.

Po vývojáře aplikací disponuje Elastic stack také komponentou APM – Application Performance Monitoring – s přidruženými knihovnami v několika programovacích jazycích, umožňující snadné napojení a sledování výkonnostních metrik aplikací, nad nimiž lze rovněž aplikovat AI detekci anomálií. Spolu s komponentami ze substacku Beats umožňující primární sběr a shipping logovacích zpráv lze v rámci vývoje nejen RAGových aplikací zjednodušit proces ladění výkonu, detekce chyb a v neposlední řadě detekci útoků a pokusů o zneužití aplikací či celých systémů. Pro samotnou práci se zájmovými daty poskytuje Elastic knihovny pro několik programovacích jazyků.

Příklad složeného vyhledávání dense vectoru vytvořeného na aplikační vrstvě a sparse vectoru s použitím nativního ELSER embeddovacího modelu z dílny skupiny Elastic*:

```
„query“: {
  „bool“: {
    „should“: [
      {
        "knn": {
          "field": "vector",
          "query_vector": [<dense_vector_here>],
          "k": <volatility_here>,
          "num_candidates": <number of candidates to return>,
          "boost": <boost_number>
        }
      },
      {
        "sparse_vector": {
          "field":"<document_inference_sparse_vecrot_field>",
          "inference_id": „<ELSER_inference_deployment_ID>“,
          "query": „What are our sales policies?“,
          "boost": <boost_number_here>
        }
      }
    ]
  }
}
```

*Zdroj: vlastní aplikace

2.1.3 Vektorové a hybridní vyhledávání

Vektorové vyhledávání a hybridní, kde k vektorovému přidružujeme vyhledávání dle klíčových slov, se liší svou přesností, latencí i komplexitou.

Klíčovou výhodou vektorového vyhledávání je hledání shodného významu se vstupním dotazem. Je vhodné pro dlouhé řetězce a v případech, kdy nelze hledat podle přesné shody textových částí (například když obsah databáze dopředu neznáme). Nevýhodou je, že na rozdíl od fulltextového vyhledávání je přesné znění klíčových slov ignorováno a obsahuje-li vstupní

dotaz řadu přesných klíčových slov, může tak výsledek tohoto typu vyhledávání mít menší relevanci. Kromě toho může potřebovat reranking, který se používá například pokud je potřeba sjednotit 2 hodnocení z 2 různých vyhledávání (vektorové/fulltextové), případně se použije semantic reranker, který ještě zanalyzuje menší počet chunků znovu a přeuspořádá je.

Hybridní vyhledávání obohacuje to vektorové o fulltextovou část a z logiky věci tak odpadají výše uvedené nevýhody. Dále je možné zapojení filtrace dat pomocí tagů/metadat. Přesnost vyhledávání je větší, klíčová slova jsou nalezena a nejsou tak ignorovány případné relevantnější dokumenty, a to i při krátkých vstupních dotazech často i o jednom či několika málo slovech. Tento typ vyhledávání je však komplexnější na implementaci, potřebuje jasně definovanou strategii přiřazování skóre dokumentům z jednotlivých podtypů vyhledávání a vykazuje vyšší latenci než samotné vektorové vyhledávání. Užití nalézá v aplikacích typu RAG a obdobných.

2.1.4 Datové struktury a optimalizace pro RAG

Při návrhu datového modelu pro vývoj RAGové aplikace je pak třeba uvažovat jak embedding nebo embeddingy, které budeme chtít zahrnout, tak způsoby vyhledávání, které má aplikace podporovat. Na základě této úvahy pak volíme samotný databázový systém a z něho vyplývající nároky na ukládaná data a jejich strukturu. Z výše uvedeného plyne, že zásadní funkcionalitou používanou v RAGu je vektorové vyhledávání, databáze a uložené dokumenty tedy musí podporovat patřičné vektorové datové typy a mít implementovány relevantní vyhledávací mechanismy. Ukládaný dokument pak musí přinejmenší obsahovat zájmová data v původní podobě (text se zájmovou informací) a alespoň jeden vektorový typ s embeddingem.

Jelikož předpokládáme ukládání velkých dat, je dále třeba uvážit strategii dělení dat, takzvaný chunking. Ukládání příliš dlouhého souvislého textu může mít na přesnost embeddovaného výstupu negativní vliv, proto volíme sekce textu buď podle počtu znaků v něm nebo podle logických celků v původním delším textu (například odstavce). K úvaze je rovněž překrývání jednotlivých částí původního textu označovaný jako overlap. Tímto přístupem lze zajistit kontinuitu významu napříč rozdělenými částmi originálního textu, kdy „okrajové“ hodnoty chunků se v jejich řadě vyskytují na začátku či konci dalších chunků. Délka chunků a velikost překryvu je věcí testování jednotlivých modelů, jelikož příliš krátké chunky mohou zapříčinit ztrátu významu a příliš velký překryv pak zbytečné využití kapacity úložiště bez další přidané hodnoty.

K úvaze je rovněž strategie zpětného získávání dat, kdy zohledňujeme zejména výpočetní náročnost (a z toho plynoucí latenci) a přesnost výsledků. Takto získané dokumenty pak budeme předkládat jako znalostní bázi pro LLM, který bude RAGová aplikace používat jako kontext pro konverzaci a chceme-li předejít halucinování zvoleného modelu a přitom zachovat pozitivní zkušenost koncového uživatele, je třeba najít rovnováhu mezi přesností a logickou posloupností dat a potřebným časem k jejich získání. Mezi nejčastěji používané vyhledávací strategie současnosti patří:

- Brute-Force – výpočet kosinové podobnosti nebo vzdálenosti v Eukleidovském prostoru mezi vektorem embeddovaného vstupního dotazu a všemi uloženými vektory – výhodou je stoprocentní přesnost, nevýhodou výpočetní náročnost a z toho plynoucí nízká rychlost pro velké datové sady.
- Approximate Narest Neighbour (ANN) – tato strategie používá indexování a grafové struktury k nalezení „n“ nejbližších vektorů ke vstupnímu dotazu.
- Hybridní vyhledávání dense + sparse vector, jak bylo prezentováno v této kapitole.
- Metadata a vektorové vyhledávání – před samotným vektorových vyhledáváním se dotaz omezí dle metadat dokumentů (časová známka, tag, …).
- Reranking – tato strategie má dva kroky, v prvním použijeme ANN k získání „n“ podobných dokumentů, následně na takto získanou omezenou sadu aplikujeme přiřazení skóre přesnějším modelem, který by byl příliš drahý (ve smyslu hardwarových prostředků) k provozu nad celou původní datovou sadou. Můžeme i analyzovat zcela jiným modelem, přímo LLM, které se na chunky podívá a ještě jednou zváží relevanci.

2.2 Aplikační vrstva

Aplikační vrstva obsahuje samotnou logiku vyvíjené aplikace. Na této vrstvě definujeme API, metody pro kontaktování datové vrstvy a implementujeme komunikaci s modely umělé inteligence. Jádrem aplikace je pak zpravidla zpracování uživatelských dotazů, jejich embedding, udržování kontextového okna s LLM, v případě ukládání dat také data chunking a celkové zpracování dat před jejich uložením do databáze na straně jedné či jejich prezentace koncovému klientovi na straně druhé. Napříč současnými předními programovacími jazyky nalézáme potřebné knihovny či celé frameworky v solidním počtu i kvalitě.

2.2.1 Analýza stávajícího stavu vývoje nástrojů, zdrojů a komponent AI v souvislosti s RAG ve vývoji software

Napříč různými srovnáními realizovanými periodiky se zaměřením na problematiku programování a AI se na přední příčce již několik let umísťuje Python [18]. Mezi hlavní důvody patří intuitivní a snadná syntaxe v porovnání s robustnějšími jazyky, možství dostupných knihoven a frameworků, platformní nezávislost a rovněž fakt, že řada AI platforem nabízí pro své použití knihovny primárně právě v tomto jazyce. Vzhledem k tomu je pak v problematice AI Python zastoupen zejména při prototypování aplikací, výzkumu a výuce.

Pro svou robustnost, škálovatelnost, paralelizaci a širokou sadu nástrojů a knihoven je v oblasti enterprise vývoje hojně používána Java, jejíž přínos a zastoupení v tomto prostředí se neomezuje pouze na problematiku umělé inteligence, ale tuto propojuje s desktopovými aplikacemi, serverovými službami, simulacemi, robotikou či IoT. Pro svou robustnost a složitější syntaxi se nehodí pro prototypování v takové míře, jako výše uvedený Python, oproti němu však poskytuje lepší výkon s ohledem na kompilaci a následný provoz na JVM a jedná se rovněž o multiplatformní jazyk.

Javascript se rovněž řadí mezi jazyky s pokročilou podporou AI pro integraci modelů umělé inteligence do dalších aplikací, zejména ve smyslu webových služeb [19]. Použití nalézá v procesech strojového učení, NLP či strojového vidění přímo v prohlížeči uživatele. S ohledem na své zaměření a fakt, že jde o interpretovaný jazyk však disponuje nižším výkonem ve srovnání s Javou a omezenými knihovnami a frameworky ve srovnání s Pythonem. Pro aplikace s vysokým výkonem či se zaměřením na bezpečnost je navíc krajně nevhodné používat jej v prohlížeči a jeho využití pro tyto účely je tak omezeno na serverový provoz.

2.2.2 Analýza aktuálních frameworků v oblasti AI v souvislosti s RAG

S ohledem na výše uvedené skutečnosti je tato podkapitola zaměřena na nástroje a frameworky pro jazyky Java zastupující enterprise svět, stabilitu a výkon a Python zastupující rapidní vývoj a prototypování, jako dva přední programovací jazyky v souvislosti s AI v současnosti.

2.2.2.1 Java

V enterprise prostředí náročném na optimalizaci výkonu, správu hardwarových prostředků a sítí, dynamické škálování a virtualizovaný provoz aplikací se Java ukazuje jako ideální volba [20]. V oblasti AI zaujímá přední místo ve vývoji ze stejných důvodů z jakých je v těchto prostředích preferována již dlouhou dobu. Jde o multiplatformní jazyk s možností

provozu výsledných aplikací nejen napříč operačními systémy, ale i různými hardwarovými platformami. Z toho důvodu nalézáme Javové aplikace na serverech, v osobních počítačích, mobilních zařízeních či v embeddovaných systémech. Java disponuje robustním ekosystémem s velkým množstvím knihoven pro rozličné účely včetně strojového učení, díky nimž je pro vývojáře snazší implementovat komplexní algoritmy. Správa paměti a garbage collection patří mezi další výhody Javy zejména s ohledem na velké datové sady používané při trénování AI modelů a jejich paměťovou náročnost, čímž se stává výrazně lepším nástrojem než jazyky preferující manuální sásahy do paměti, které při neodborné implementaci mohou představovat výkonnostní a bezpečnostní riziko. Škálovatelnost a paralelizace – nativní vlastnosti Javy – pak pomáhají při zpracovávání velkých dat v kratší době a k efektivnímu využití hardwarových prostředků. V neposlední řadě je předností Javy fakt, že jde o jazyk kompilovaný do bytekódu, díky čemuž nejen není třeba distribuovat zdrojový kód enterprise aplikací, ale provoz takového kódu je výrazně rychlejší oproti interpretovaným jazykům.

Pro Javu existuje několik knihoven a frameworků s orientací na strojové učení:

- Weka – knihovna s určením pro strojové učení a data mining s komponentami pro instance dat, filtry, klasifikátory a jejich evaluaci a selekci atributů s možností snadného vložení závislostí do vyvýjené aplikace prostřednictvím nástroje Maven [21].
- Deeplearning4j – také DL4J poskytuje nástroje k budování neuronových sítí a vývoj modelů za účelem deep learningu jakož i vývoj aplikací užívajících NLP. Tato sada nástrojů je zaměřena zejména na enterprise oblast a disponuje kompatibilitou s frameworky jako Hadoop nebo Apache Spark jakož i možností utilizace grafických procesorů [20] [22]. Moduly zahrnují TensorFlow, grafové algoritmy či konverzi dat do tensorů k jejich následnému užití v neuronových sítích.
- MOA – knihovna pro streamování a analýzu velkých dat používaná k vývoji AI aplikací jako detekce plagiátorství nebo analýza bezpečnostních událostí (neoprávněné vstupy do systémů, sítí, apod) [20].
- Java-ML – sada knihoven s dobře zdokumentovaným zdrojovým kódem, umožňující snadnou integraci AI alogoritmů do vyvíjených aplikací. K dispozici je řada tutoriálů a příkladů, knihovna však již není aktivně vyvíjena. [23] [24].

2.2.2.2 Python

Jazyk Python se vyprofiloval jako přední programovací jazyk v souvislosti s umělou inteligencí. Díky své intuitivní syntaxi a masivnímu množství knihoven se stal de facto standardem při prototypování aplikací nebo psaní celých jednodušších řešení a tam, kde výkon není prioritou (například ve výuce a demonstracích principů AI). Většina současných služeb třetích stran nabízející embedding či LLM cestou API či celé platformy sdílejících širokou škálu modelů poskytují hotové knihovny právě v tomto jazyce [25] [26].

V souvislosti s AI se pak na přední příčce drží framework LangChain a jeho sesterské platformy LangGraph a LangSmith. LangChain poskytuje potřebnou míru abstrakce nad opakujícími se úlohami a nabízí intuitivní vývojářské rozhraní. Disponuje rovněž hotovými třídami pro napojení vyvíjené aplikace na přední současné služby a modely jako OpenAI, Amazon Bedrock či Huggingface. Hotové třídy poskytují podporu pro:

- Napojení na konkrétní modely a jejich metody
- Prompty
- Vektorová úložiště
- Loadery (pro čtení dat z různých zdrojů)
- Parsery výstupů
- Rozdělení textu (chunking)
- Nástroje (definice vlastních metod pro použití ze strany LLM)

LangChain tak umožňuje fine-tuning modelů prostřednictvím řady jejich parametrů – nastavení jejich role, teploty, kontextového okna či definice vlastních metod, které modely mohou přijmout jako parametr a tyto následně použít pro specifické úlohy nad zaslaným kontextem z datové sady. Podpora těchto mechanismů i formáty návratových hodnot ve stávajících AI modelech se však značně liší a při vývoji aplikace je proto třeba určité obezřetnosti a získané výsledky z každého modelu je často třeba dále upravit před jejich prezentací koncovému uživateli.

2.2.2.3 Další knihovny, frameworky a nástroje

Knihovny a frameworky pro vývoj AI nejsou samozřejmě omezeny shora uvedeným seznamem. Mezi dalšími předními [26] nalezneme PyTorch – open source knihovnu pro strojové učení oblíbenou v oblasti výzkumu, s rostoucí komunitou a zastupující přední pozici

mezi nejšíře používanými AI frameworky vůbec. Scikit-Learn, knihovna rovněž pro strojové učení a data-mining disponující kvalitní dokumentací a uživatelskou přívětivostí, je vhodná zejména k prototypování a pro malé projekty. TensorFlow z dílny společnosti Google je open source framework se zaměřením na deep learning. Známý je především pro svou pružnost a škálovatelnost, disponuje širokou komunitou, kvalitní dokumentací a množstvím příkladů. OpenAI poskytuje nástroje pro řadu repetitivních úloh v oblasti umělé inteligence, jako převod obrázků do textu či text to mluveného slova, nejznámější je pro své GPT modely ze skupiny LLM. Je často používána pro tvorbu AI asistentů a RAGových aplikací. V oblasti výzkumu, výuky, vývoje a testování algoritmů strojového učení, konceptů AI a prototypování je rovněž hojně používána open source knihovna PyBrain, tato se však netěší tak velké komunitní podpoře a její dokumentace je oproti předchozím zmíněným omezená. K dispozici je rovněž řada enterprise nástrojů, jejichž cena a tím i dostupnost pro vývojáře se liší, jako Guru, Coveo, GoLinks či Elastic Enterprise AI Search.

2.2.3 Metody RAG

Ačkoliv se na základě dosud pokrytých skutečností a principů může tvorba RAGové aplikace jevit jako přímočará záležitost, v této podkapitole chci demonstrovat sled úvah a možných řešení, které je třeba při vývoji zvážit, a jejich dopady na výslednou podobu a výkon konečného produktu, jakož i pokročilé mechanismy umožňující zachovat konverzaci i v případě chybějících informací nebo takových, které nejsou součástí stávajícího kontextového okna.

V první řadě je ke zvážení otázka embeddingu. Lze jej realizovat na datové vrstvě, podporuje-li toto databázový systém dle výběru vývojáře či softwarového architekta. I pokud tomu tak je, lze embedding rovněž realizovat na aplikační vrstvě a obě hodnoty následně uložit do výsledné pobody ukládaných dat. Podpora embeddingu na datové vrstvě je však často předmětem pokročilých licencí, proto jej zpravidla realizujeme právě na aplikační vrstvě. Tomu pak odpovídá i podoba vyhledávacích dotazů, kde je třeba buď zohlednit patřičné inference pipelines a jejich parametry nebo jako součást dotazu zaslat již embeddovaná data či obojí. Sem patří i úvaha, jaké uživatelské dotazy embeddovat. Zašle-li uživatel delší dotaz obsahující klíčový význam, který má smysl embeddovat a k němuž mohou existovat relevantní data, pak je odpověď zřejmá, k úvaze je však otázka zpracování krátkého dotazu, často o jediném slovu, který se vyskytne v rámci přirozeného toku konverzace s LLM, kdy se uživatel v návaznosti na předchozí odpověď doptává dotazy typu „Jak?“ nebo „A kde?“ Zde je zřejmé, že embedding a vyhledávání nad takovými dotazy nemůže přinést odpovídající výsledky, proto je třeba implementovat strategii zpracování takových dotazů (například můžeme dle slovníkové analýzy

tyto dotazy zasílat rovnou LLM bez embeddingu a doplněného kontextu o nová data nebo můžeme implementovat strategii hypotetických dotazů či hypotetických odpovědí).

Strategie hypotetického dotazu (hypothetical query) nebo hypotetické odpovědi spočívá v přepsání uživatelského dotazu dotazem jiným, který svým významem lépe odpovídá tomu, co chceme hledat, nebo poskytnutím odpovědi významově blízké vyhledávanému obsahu. Příkladem budiž již výše uvedený dotaz „A jak?“. Mějme tedy kontextové okno, kde již probíhá konverzace s LLM, avšak v jeho rámci neexistuje informace, na níž se uživatel tímto navazujícím dotazem doptává. Před samotným embeddováním takového dotazu a vektorovým vyhledáváním nad ním tak zahájíme s LLM separátní konverzaci mimo uživatelovu pozornost s odlišným nastavením vůči konverzaci hlavní. Tato nová konverzace je pro každý dotaz jednorázová, tedy v jejím rámci neudržujeme kontextové okno. Pro každé volání použijeme kontext hlavní konverzace, model pro tuto paralelní konverzaci však nastavíme do velmi volného režimu systémovou zprávou typu „Jsi génius. Pokud nevíš odpověď, vymysli si ji.“ Odpověď na uživatelovu otázku pak bude sice s největší pravděpodobností objektivně chybná, ovšem svým významem se bude velmi blížit skrytému smyslu původního uživatelského dotazu. Tuto odpověď pak prohlásíme za hypotetickou odpověď na uživatelův dotaz a embedding a vyhledávání provádíme nad jejím obsahem namísto původního dotazu. Do kontextu hlavní konverzace pak přidáme původní uživatelský dotaz a dokumenty získané v paralelním vláknu, čímž simulujeme plynulý „lidský“ tok konverzace. Příklad:

Systémová zpráva: „Jsi asistent. Shrnuješ informace o společnostech pro obchodní partnery a úřady.“

Otázka (uživatel): „Kdo založil firmu JardaFedra, s.r.o.?“

Odpověď (LLM): „Společnost založili Jaroslav Novotný a Ferdinand Suchý. Každý společník má 50% podíl. Společnost je zapsána u Městského soudu v Praze.“

Otázka (uživatel): „Kde a kdy to bylo?“

Geneze hypotetické odpovědi:

1. *Systémová zpráva pro novou konverzaci: „Jsi génius, pokud nevíš odpověď, vymysli si ji.“*
2. *Zaslání stávajícího kontextu do nové konverzace s LLM.*
3. *Odpověď (LLM): „Společnost byla založena v Praze v České republice po roce 2000.“*
4. *Embedding odpovědi jako hypotetické odpovědi, vyhledávání v databázi.*
5. *Přidání získaných dat z databáze do kontextu původní konverzace.*
6. *Zaslání nového kontextu s původním dotazem do hlavní konverzace.*

Odpověď (LLM, již na základě faktických dat): Společnost JardaFerda, s.r.o. byla založena v Liberci dne 4. 3. 2005, sídlo však má na adrese Strmá 6, 140 00 Praha 4*.

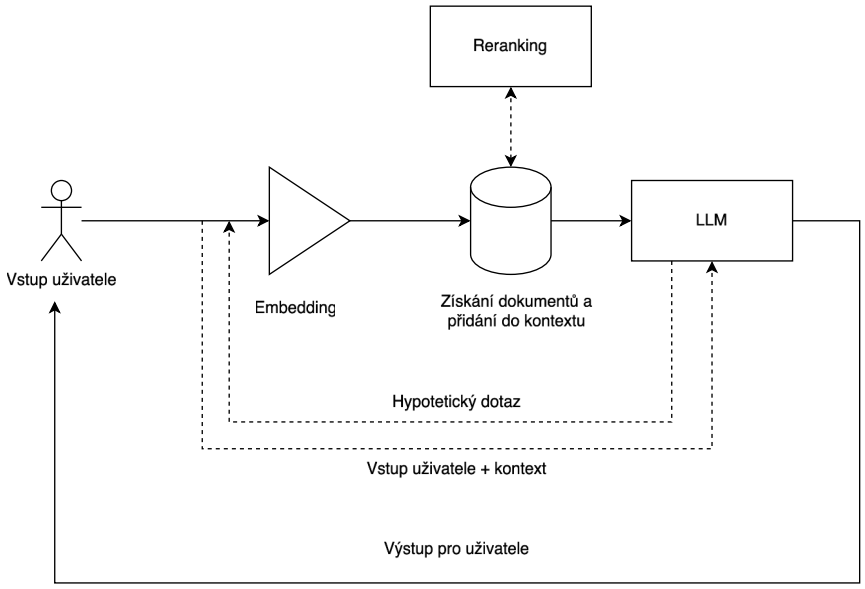
**Poznámka – data v příkladu jsou smyšlená.*

Strategie vyhledávání je dalším předmětem k rozhodnutí. Vývojář nebo architekt řešení bude volit zejména mezi strategiemi Simple retrieval, Multi-hop retrieval a Hybrid retrieval. Jak je z názvu zřejmé, Simple retrieval spočívá v embeddingu vstupního dotazu a vyhledáváním v databázi v jediném kroku. Výhodou tohoto přístupu je jeho intuitivnost a výpočetní nenáročnost vzhledem k ostatním přístupům, rychlost implementace i výkonu a dobré výsledky při vhodném dotazu založeném na dotazování na fakta. Mezi nevýhody patří možnost vynechání relevantních dokumentů se složitějším obsahem nebo vrácení dokumentů, které sice obsahují požadovanou informaci, ale nevedou k jednoznačné odpovědi.

Multi-hop retrieval pak spočívá v získávání dokumentů v několika krocích, mezi nimiž dochází k postupnému zpřesňování původního dotazu na základě nově získaných informací v každém takovém kroku. Mezi jeho výhody patří získání obsáhlejších dokumentů, jejichž porozumění vyžaduje složitější úvahu a potenciálně přesnější výsledky oproti předchozí strategii, avšak nevýhodou je složitější implementace i déle trvající vykonání kódu a náchylnost na zcela mylné odpovědi při nevhodné integraci nově získaných dat do původního dotazu.

Hybridní přístup pak obvykle spočívá v kombinaci vyhledávání nad dense a sparse vektorem a může kombinovat prvky Simple retrieval i Multi-hop retrieval. Samozřejmostí je odpovídající struktura dat na datové vrstvě. Tento přístup vykazuje vyvážené výstupy co do náročnosti běhu a přesnosti získaných odpovědí, je nicméně složitější na implementaci, vyžaduje složitější strukturu dostupných dat (sparse a dense vektory) a jemnější ladění parametrů vyhledávání.

Obrázek 3 – Schematické znázornění procesů v RAG



Zdroj: vlastní zpracování

Tabulka 2: Dopad vyhledávacích strategií na složitost implementace a kvalitu výstupu

Parametr	Simple retrieval	Multi-hop retrieval	Hybrid retrieval
Náročnost implementace	Nízká	Vysoká	Střední / Vysoká
Vyhodnocení významu	Nízké	Vysoké	Střední
Rychlost	Vysoká	Nízká	Střední / Nízká
Oblast užití	Jednoduché dotazy	Složité dotazy	Obecné dotazy se širokým spektrem významů
Nástroje	BM25, DenseRetriever	Decomp	ColBERT + BM25

Zdroj: Vlastní zpracování

2.2.4 Ladění výkonu

Současné frameworky v oblasti umělé inteligence disponují řadou nástrojů umožňujících ladění výsledné aplikace, kdy jejich využití (či nevyužití) může v konečném důsledku představovat dramatický rozdíl v jejím výkonu. Mezi obecné postupy můžeme zařadit ukládání ve vyrovnávací paměti a paralelismus, kdy ve vyrovnávací paměti lze uchovávat dotazy a odpovědi k nim, pročež pro nové dotazy s vysokou mírou podobnosti k již položeným neprovádíme nové vyhledávání, ale namísto toho vracíme „zapamatované“ odpovědi. Nevýhodou tohoto postupu je však fakt, že podobnost dotazů je třeba vyhodnotit již na aplikační úrovni a u vyrovnávací paměti je třeba určit správnou délku její platnosti. Ve věci paralelismu představuje hlavní výzvu skutečnost, že i když nám tento postup umožní například embeddovat různé části textů (chunků) souběžně, výpočetní a paměťová náročnost mohou prudce narůstat a je pak třeba určit a implementovat omezení maximálního počtu procesů / vláken / eventů spuštěných vedle sebe. Řada služeb třetích stran navíc limituje ve svém API maximální počet připojení v časové periodě, tedy přehnaným či neřízeným paralelismem můžeme snadno takového limitu docílit a aplikace tak ve výsledku může být méně stabilní.

LangChain obsahuje rozhraní k výše uvedeným úlohám, kdy pro vyrovnávací paměť jsou k použití moduly `InMemoryCache` nebo `RedisCache` a další, pro paralelismus je k dispozici rozhraní `Runnable.parallel()` nebo `RunnableSequence`. Vývojář může rovněž využít specifické strategie získávání dokumentů, jako `MultiQueryRetriever` nebo `ContextualCompressionRetriever` pro souběžné zpracování více dotazů.

LlamaIndex, dříve pod názvem GPT Index, se zaměřuje primárně na indexování a získávání dat, tedy ladění v jeho rámci spočívá zejména v dělení textu (chunking), logice získávání dokumentů, zpracování dotazů a indexaci dat, použití vyrovnávací paměti a asynchronní běh.

Dále existuje řada obdobných nástrojů s analogickými funkcionalitami (například Haystack, PromptFlow, CrewAI či Unstructured.io).

2.3 Frontend

Třetí vrstvou, a tedy nejblíže ke koncovému uživateli, je ve třívrstvém modelu frontend, který má zpravidla podobu tenkého klienta a v současné době je realizován buď formou desktopové či mobilní aplikace nebo aplikace webové, komunikující s aplikační vrstvou přes její API, kdy může jít buď o celé webové portály nebo pluginy typu AI asistentů.

2.3.1 Analýza současného stavu vývoje a trendů v oblasti uživatelské dostupnosti a zkušenosti s AI produkty

Na straně frontendu pro koncové uživatele se z hlediska RAGových aplikací a AI asistentů nejedná v pravém slova smyslu o novou oblast, jelikož u tenkých klientů bylo již dříve standardem, že komunikují s backendovou částí aplikace prostřednictvím RESTových API nebo technologie Websockets a na své straně si dočasně ponechávají pouze malé množství dat v krátkodobé vyrovnávací paměti. Na tomto postupu se nic nemění ani v případě aplikace typu RAG a vývoj této vrstvy proto z technického hlediska není v zásadním ohledu inovativní co do zasílání a získávání strukturovaných informací. Nové výzvy však může představovat z hlediska designu v souvislosti s problematikou UX a co nejplynulejšího uživatelského zážitku, včetně metod pro netechnicky orientované uživatele či uživatele s omezenými schopnostmi vnímání nebo jinými speciálními potřebami. Zde již v minulosti našly uplatnění technologie převodu textu do řeči, nyní je možné uživatelskou zkušenost obohatit o plynulou konverzaci, kdy RAGový frontend lze rozšířit o porozumění mluvenému slovu nebo adaptivní přizpůsobování chování dle potřeb uživatele. Moderní soudobé operační systémy dále obsahují grafický a hlasový frontend pro své interní AI modely a algoritmy umožňující hromadné vyhledávání v uživatelských datech (dokumentech, zprávách, emailech, fotografiích) na základě porozumění obsahu či prezentaci sumarizací a souvislostí mezi nimi, kdy aplikační logika k řešení takových úloh může použít jeden nebo více AI modelů. Na straně frontendu se nicméně stále jedná toliko o reprezentaci výstupů z aplikační vrstvy ve formátu co nejpřijatelnějším pro uživatele (řeč zde není o přímé komunikaci s AI modely trénovanými pro porozumění mluvenému slovu, kdy taková interakce neodpovídá třívrstvé architektuře).

2.3.2 HITL – Human-in-the-Loop

Princip Human-in-the-loop je založen na myšlence, že navzdory faktu, že modely umělé inteligence mohou být trénovány za rozličnými účely nad velkými datovými sadami a mohou

zpracovávat velké množství dat v reálném čase, postrádají řadu dovedností specifických pro člověka, jako je schopnost abstrakce, nadhled, emaptie, etika, svědomí či „obyčejný“ běžný rozum, což představuje zásadní překážku v rozhodovacích procesech, které by jinak bylo možné zcela automatizovat (například v případě soudů, obchodních rozhodnutích, finančních otázkách nebo otázkách národní obrany či usmrcení člověka nebo zvířete). Toto téma tak dalece přesahuje technickou otázku umělé inteligence a jedná se o mezioborový princip zasahující do oblastí práva, filozofie či do náboženských otázek. V technické rovině je pak důraz kladen na pozitivní přínos člověka do procesů nad daty umožňující lepší trénování, korekci chování, evaluaci žádoucích výsledků či v operativních rozhodovacích procesech.

V souvislosti s RAGovými aplikace pak princip HITL nalézá uplatnění při ověření platnosti a smysluplnosti získaných dat a jejich prezentace a zpětné vazby za účelem nápravy chování modelu či určení parametrů skóringu, určení míry relevance a kvality odpovědí LLM a rozeznání situací, kdy model halucinuje a nenásledování takových výstupů, což by v případě automatických rozhodovacích procesů mohlo vyústit v katastrofické scénáře.

2.3.3 Interaktivní AI asistenti

Na straně frontendu lze v horizontu posledních měsíců až jednotek let pozorovat masivní implementaci interaktivních AI asistentů. Pružnost a schopnost operovat nad daty a poskytovat žádoucí odpovědi uživatelům se napříč nasazeními velmi různí od asistentů zastupujících spíš funkci interaktivního nabízení často kladených otázek a odpovědí na ně po skutečné asistenty schopné nabízet relevantní obsah z veřejně dostupných dokumentů jako jsou všeobecné obchodní podmínky, parametry produktů či řešení potřeb a problémů klientů s možností přepojení na živého operátora. Zpravidla se jedná o pluginy již existujících aplikací s uživatelským rozhraním implementovaným pomocí popup prvků. Tento trend se navíc neuplatňuje pouze u webových a mobilních aplikací, ale užití nalézá i v automatických telefonních systémech, do té doby odkázaných na stisknutí tlačítek uživatelem jako jedinou možnou formu interakce. S rozšiřujícími se možnostmi uplatnění AI modelů a rostoucím výkonem komerčně dostupného hardware však lze do budoucna očekávat větší implementaci úplných modelů nebo jejich omezených derivátů přímo na straně frontendu a bohatší uživatelskou zkušenost danou novou mírou interakce s jejich vyspělými variantami.

Modulární RAG nad distribuovanou dokumentovou databází

3 Úvod do praktické části

Následuje popis prototypu aplikace implementující výběr ze sady výše uvedených principů. Jedná se o RAGovou aplikaci postavenou nad distribuovanou dokumentovou databází s nativní podporou embeddovaných dat navrženou pro nasazení v cloudových prostředích. Zdrojový kód [příloha A], konfigurace [příloha B] a dokumentace postupu pro cloudové nasazení [příloha C] jsou přílohami této práce.

3.1 Obecný popis

Představovaná aplikace je Proof of Concept RAG prototyp se zaměřením na testování výkonu, chování a výsledků nejrůznějších embeddovacích a LLM modelů s jejich snadnou zaměnitelností a možností přidávání nových modelů při zachování totožného přístupu a sady postupů ze strany uživatele beze změn API. Uživatelem je zde míněn datový inženýr, vývojář, analytik nebo AI specialista.

Aplikace je postavena na klasickém třívrstvém modelu. Jako datové úložiště používá distribuovanou dokumentovou databázi Elasticsearch, kdy součástí dodávané dokumentace a příkladů nastavení je rovněž definice mappingu indexů, do nichž jsou dokumenty ukládány, a embeddovacích pipelines využívající nativní Elasticový model ELSER, jakož i předpisy pro Docker a Kubernetes pro nasazení všech potřebných komponent Elastic stacku na virtualizačních a cloudových platformách.

Aplikační vrstva pak obsahuje hlavní logiku, která je rozdělena na jádro a modulární část se zaměřením na snadnou rozšiřitelnost modulární části při zachování neměnného funkčního modelu aplikace a způsobu komunikace s dalšími komponentami. Je implementována v jazyce Python verze 3.13 s použitím frameworku FastAPI pro zpracovávání požadavků formou HTTP dotazů na definované REST API. API je navrženo tak, aby bez nutnosti strukturálních změn mohlo dojít k pohodlnému rozšiřování aplikace. Dále je využito funkce automatické dokumentace a propagace těchto definic na separátním endpointu FastAPI. Kód je objektově orientován a je dokumentován přímo v deklarovaných metodách dle Python standardu a obecných zvyklostí v anglickém jazyce. Aplikační vrstva podporuje rozlišení uživatele cestou sezení, ukládání dat z několika zdrojů a formátů se snadnou rozšiřitelností o nové zdroje a formáty a jejich embedding na aplikační úrovni s možností využití nativních modelů na úrovni datové vrstvy. Dále podporuje zpracování dotazů nad již uloženými daty s možností využití principu generovaného hypotetického dotazu odvozeného z

kontextu dosavadní konverzace, ukládání historie, auditní logování a vracení odpovědi jazykového modelu spolu se sadou původních dokumentů získaných z databáze pro každý dotaz. Samozřejmostí i nutností je pak možnost definice velikosti kontextového okna uživatelem jakož i definice množství zájmových dokumentů, které se mají na dotaz získávat. Součástí konfigurace je předpis pro škálovatelné nasazení na cloudových platformách.

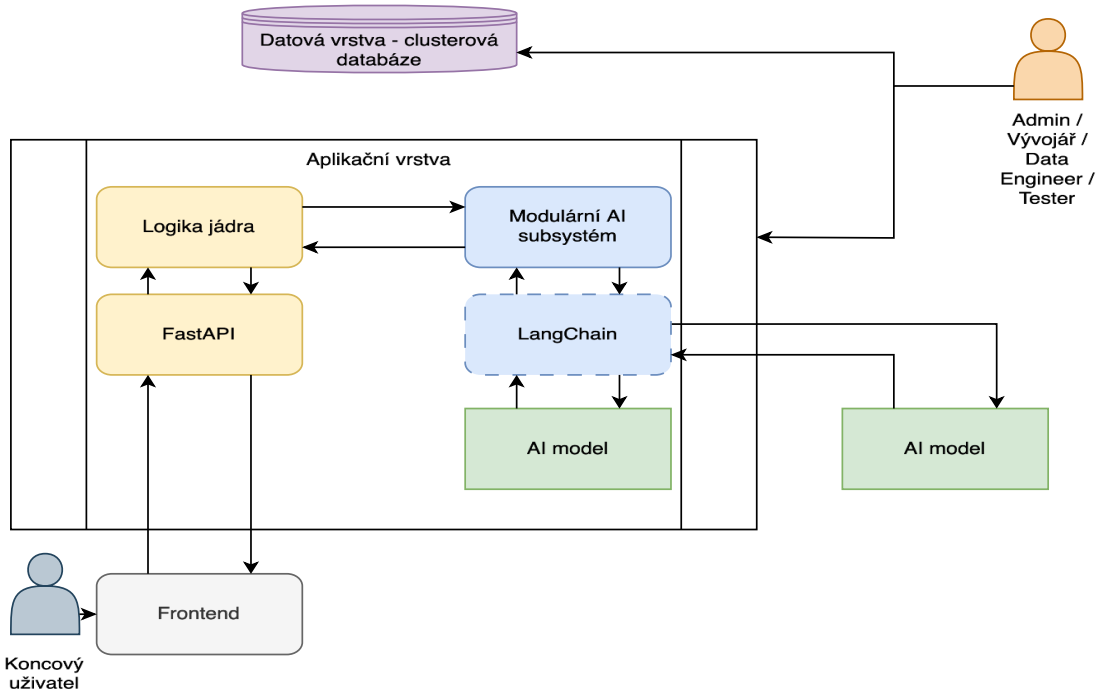
Třetí vrstvou je frontend a uživatelské rozhraní. Toto je realizováno jako separátní webová React aplikace. Nasazuje se nezávisle na aplikační vrstvě a je tak snadno zaměnitelná za zcela jiné řešení, jediným nutným spojovacím prvkem je shodná definice API. Frontend je implementován jako jednoduchá Proof of Concept chatovací aplikace a nevyužívá proto všech funkcí poskytovaných aplikační vrstvou, uživatel si volí zdroj dat v databází z nabízených možností a nad těmito klade dotazy vybranému jazykovému AI modelu. Frontend dále podporuje přihlášení konkrétního uživatele, přepínání mezi jeho konverzacemi uloženými v historii a možnost náhledu do získaných dokumentů, z nichž LLM ve své odpovědi vycházel.

3.2 Základní schematický přehled architektury, funkčností a endpointů

3.2.1 Architektura

Architektura aplikace při pohledu „shora“ je jednoduše znázorněna na následujícím obrázku, jednotlivé vrstvy jsou dále popsány v dedikovaných separátních kapitolách.

Obrázek 4: High-level přehled architektury aplikace

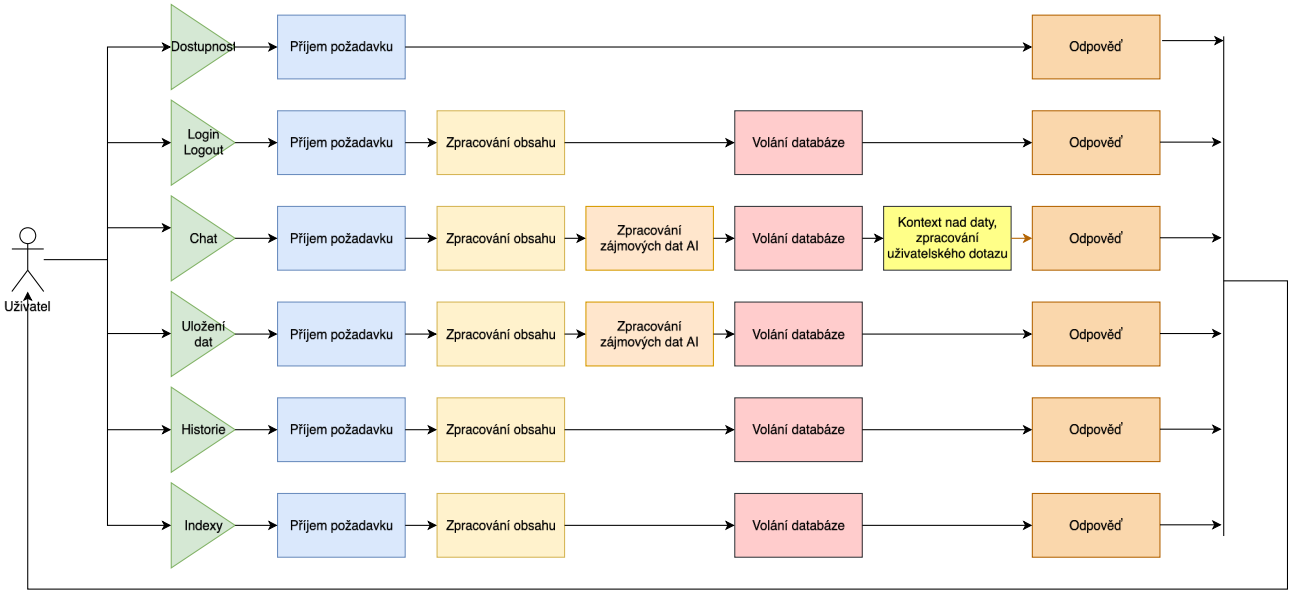


Zdroj: Vlastní zpracování

3.2.2 Funkčnosti

Na následujícím obrázku jsou schematicky popsány vybrané use casy implementované v aplikaci. Kompletní přehled backendů s popisem jejich určení je obsažen v kapitole 3.2.3.

Obrázek 5: Přehled zpracování některých use casů



Zdroj: Vlastní zpracování

3.2.3 Backendové endpointy

Backendové endpointy jsou zdokumentovány dle standardu OpenAPI a v rámci demonstrace nasazení jsou dostupné na URL https://project2.freedev.cz/docs (vizte kapitolu 3.6 pro další informace o PoC nasazení). Kompletní seznam backendových endpointů sestává z bodů:

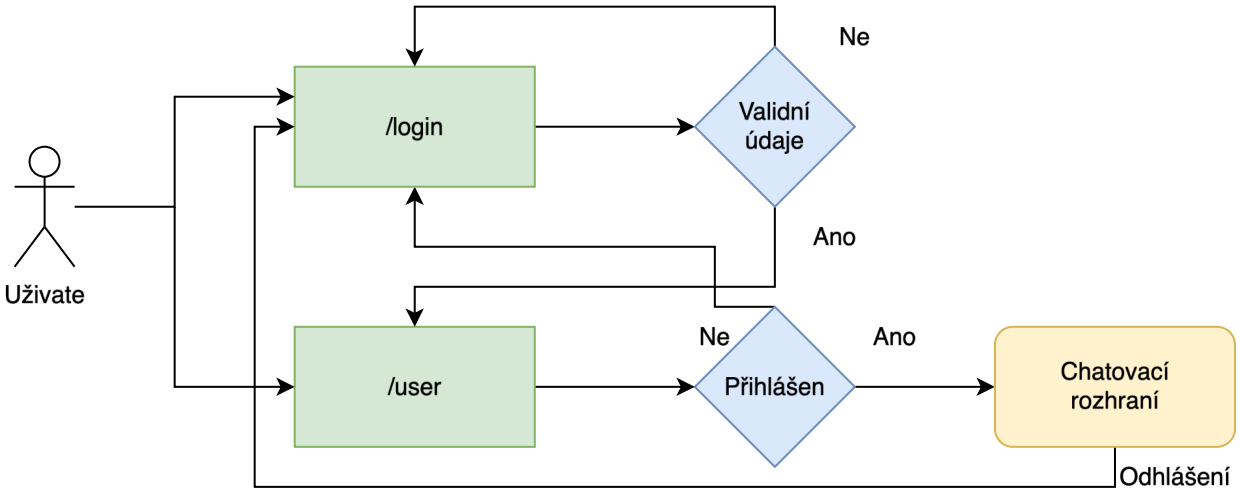
- GET /health – ověření dostupnosti / živosti aplikace
- GET /docs – dokumentace API formou Swagger dle OpenAPI standardu
- POST /login – přihlášení uživatele
- POST / logout – odhlášení uživatele
- POST /chat_query – dotaz na obsah databáze a chatování nad odpovědí s LLM v rámci kontextového okna (plus vytvoření či update historie k této konverzaci)
- POST /clean_chat – vymazání komunikace z historie chatů
- POST /store_data – uložení nových dat do databáze (data jsou buď zaslána či odkázána)
- POST /histories – získání historie všech uživatelových konverzací
- POST /indices – získání seznamu dostupných indexů s daty

- POST /logged_in – ověření, že je konkrétní uživatel přihlášen

3.2.4 Frontendové routy

Frontendová část je pojata jako PoC chatovací klient a využívá dvě routy - **/login** a **/user**, kdy první routa slouží k přihlašování uživatele a pod druhou se nachází uživatelské rozhraní samotné chatovací aplikace. Nad touto routou probíhá rovněž kontrola přihlášení a není-li uživatel přihlášen, proběhne automatické přesměrování na routu pro přihlášení. Frontendové rozhraní je pro účely demonstrace dostupné na URL https://project.freedev.cz.

Obrázek 6: Routy a funkčnosti za nimi



Zdroj: Vlastní zpracování

4 Datová vrstva

Datová vrstva aplikace je postavena na technologii Elastic stack, tedy na databázi Elasticsearch a přidružených komponentách. Databáze Elasticsearch byla zvolena z důvodu vhodné kombinace obecných předností dokumentových databází a unikátních vlastností této databáze, jmenovitě:

- Redundance a vysoká dostupnost – jedná se o clusterovou databázi se snadným horizontálním škálováním, která již ve výchozím režimu podporuje redundanci dat cestou primárních shardů a replik, kdy replika a primární shard nesdílí jediný datový node. Aplikace využívá indexy s jediným primárním shardem a jednou replikou, a to z důvodu omezeného očekávaného zápisu dat. Více primárních shardů znamená dělení příchozích dat pro paralelní zápis a hodí se zejména pro clustery s větším počtem datových nodů a intenzivním zápisem, kdy ani jedna z těchto vlastností se netýká prototypu aplikace, naopak zde je očekáván nárazový zápis, ale časté vyhledávání. Teoreticky by za tím účelem bylo možné nastavit větší počet replik, ale v rámci prototypu je nasazen minimalistický cluster, kde taková multiplikace není možná pro nedostatečný počet datových nodů. Vysoká dostupnost dat pak

vychází z již uvedeného, pokud node přestane být dostupný a z clusteru se odpojí, jiný node převezme jeho místo. Pokud došlo ke ztrátě primárního shardu, je jeho náhrada vytvořena z repliky a nová replika je vytvořena na jiném nodu, pokud je to možné. Dojde-li ke ztrátě repliky, dojde obdobným způsobem k vytvoření nové.

- Datová struktura indexů – databáze Elasticsearch pracuje mimo jiné s několika vlastními datovými typy jako s rozšířeními obecnějšího typu String. Jedná se o typy Text, Keyword či IP, nad nimiž je v Elasticsearch definována sada nadstandardních funkcí. Jmenovitě se jedná o použití analyzeru pro ukládaný text, který cestou reverzní indexace velmi zefektivňuje následné fulltextové vyhledávání, které může být nad to doplněno vyhledáváním ebmeddovaných dat, a dále datové typy přímo vyhrazené pro embeddovaná data s definovanými typy dotazů nad nimi s možností jejich kombinace, či určování geolokace přímo z IP adresy požadavku a jeho určení na mapě. Aplikace využívá většinu těchto možností se snadným případným konfigurováním dalších.
- Templating – šablonovací systém, na jehož základě jsou automaticky vytvářeny indexy v databázi, využívá aplikace nejen k definování vlastních datových typů a použitých funkcí nad nimi, ale rovněž ke specifikaci toho, co se mapovat má. Zde je využito vlastnosti databáze Elasticsearch, která mappingem zajišťuje možnost filtrovat a agregovat data. Mapping je však závazný a dokument, jehož struktura mappingu indexu odporuje, není možné do něj uložit, kromě toho je mapping náročný na operační paměť. Aplikace proto využívá indexů nastavených šablonou tak, že mapována jsou pouze zájmová data s patřičnou sadou funkcí, což vede k šetrnějšímu využití RAM jakož i k širší toleranci přijatých dokumentů (hodnota pod nemapovanými klíči nemusí být shodná napříč dokumenty).
- ILM – Index Lifetime Management – Aplikace této vlastnosti využívá pro postupné odstraňování auditních dat. Po dosažení definovaného stáří indexu se tento takzvaně „odrolluje“, což představuje vytvoření jeho následníka, do něhož se začne nově zapisovat, zatímco tento následník stejně jako odrollovaný index jsou rovněž dostupné pro čtení uložených dat. Po stanovené době se takto odrollovaný index odstraní včetně veškerých v něm uložených dat. Nad to Elasticsearch podporuje další možnosti správy těchto dat před jejich odstraněním (postupné přesouvání dat mezi životními fázemi na dedikované nody dle jejich stáří či zajištění redundance formou snapshotů na vzdálených úložištích), tyto vlastnosti však pro nedostatečně velký cluster nejsou v tomto prototypu použity, při nasazení většího clusteru je tato možnost k dispozici.
- Role a oprávnění – Databázi Elasticsearch umožňuje řízení přístupu k datům s vysokou granularitou, kdy kromě přístupu k indexům pro čtení, zápis, modifikaci a úpravu šablon na

bázi rolí je možné rovněž přístup omezit pro dokumenty s konkrétními hodnotami pod specifikovanými klíči.

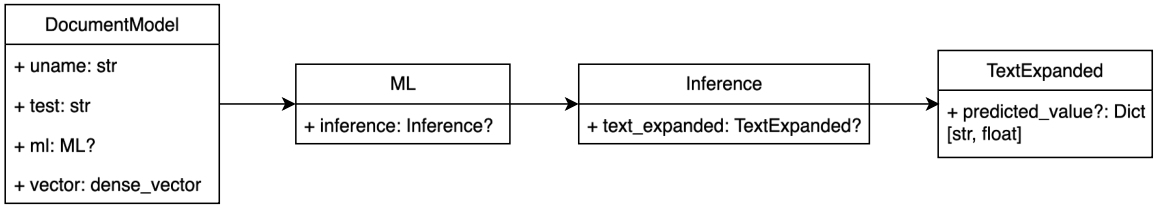
- ML – Machine Learning – Aplikace využívá nativní podpory strojového učení v databázi Elasticsearch tím způsobem, že jeden node je dedikovaný pro tuto roli. Zde je nasazen embeddovací model ELSER z díly společnosti Elastic zajišťující embeddování příchozích dat do datového typu sparse vector. K tomu je v projektu definovaná takzvaná inference pipeline, tedy postup zpracování dat před jejich konečným uložením. Inference pipeline specifikuje použití ELSER modelu nad konkrétním klíčem, výstupem je podobjekt, kdy klíče jsou zjištěná a odvozená významově zásadní slova a hodnotou je index jejich zastoupení v textu. Toto nekoliduje s jinými možnými typy embeddingu, které jsou generovány na aplikační úrovni – uložené dokumenty tak mohou obsahovat celou řadu různých embeddingů pro uložená data (vizte kapitolu 3.3 o aplikační vrstvě), nad nimiž lze vyhledávat kombinací několika dotazů v jednom s možností prioritizace výsledků určitých typů hledání nad jinými.
- APM – Application Performance Monitoring – Aplikace využívá nativního Python agenta z dílny společnosti Elastic jakož i úplného řešení pro monitorování výkonu aplikace, latence, zpracování requestů, sledování transakcí po částech, závislosti služeb a detekci anomálií. Tento systém je zejména použit pro měření a vyhodnocování rychlosti zpracování jednotlivých kroků při embeddování dat a komunikaci s LLM jakož i pro indikaci případných neošetřených chyb, pokud by se nějaké vyskytly, a indikaci případného neočekávaného chování cestou strojového učení.
- Vizualizace – Pro orientaci auditních událostí a výsledků APM je součástí nasazení rovněž aplikace Kibana umožňující přehlednou vizualizaci a hlubokou analýzu nad sebranými daty uloženými v databázi.
- Uživatelé – V rámci prototypu jsou uživatelé aplikace rovněž uloženi v databázi Elasticsearch. Důvodem je zejména nezatěžování aplikace dalšími závislostmi v této fázi vývoje.

Aplikace komunikuje s databází prostřednictvím dedikovaného Python modulu „elasticsearch[async]“, kdy tato komunikace je realizována prostřednictvím HTTPS requestů nad Elasticsearch API. Veškeré výchozí konfigurace pro databázi jsou uloženy v adresáři „configs/elasticsearch“, kde existují jako http requesty pro vytvoření uživatelských rolí, šablon indexů, indexů samotných, ingest pipeline a náhledů do dat formou Kibana data views a

Dashboardů. Tyto requesty lze přímo zaslat do databáze (s patřičným oprávněním), kde je tak zajištěno vytvoření datové struktury pro správný běh aplikace [příloha B].

4.1 Datový model aplikace

Obrázek 7: Datový model aplikace



Zdroj: Vlastní zpracování

Nepovinný atribut ml je generován databází automaticky na základě připravené embeddovací inference pipeline dostupné v přílohách [příloha B]. Její funkčnost je vázána na spuštěný nativní ELSER model a provoz databáze alespoň s trial licencí. Jeho absence nemá vliv na funkčnost aplikace jako celek.

5 Aplikační vrstva – backend

5.1 Core a modulární systém

Aplikační vrstva je implementována v jazyce Python ve verzi 3.13. S okolním světem komunikuje přes REST API, kdy za tímto účelem využívá frameworku FastAPI, jeho funkce automatické geneze dokumentace API, asynchronních volání a zpracování událostí na pozadí. Vnitřně je aplikace členěna do pěti základních okruhů – konfigurace, „core“ části – jádra, modulární části, kde je rovněž začleněna integrace umělé inteligence, routeru pro zpracování API požadavků a odpovědí a DAO části pro komunikaci s databází.

5.2 Konfigurace

Konfigurace aplikace se nastavuje v adresáři „application/config“, kde jsou připraveny konfigurační soubory používající YAML formát. V souboru common.yml je definováno obecné chování aplikace s vazbou na databázi, prefix datových indexů, jméno auditního indexu, jméno inference pipeline, a uživatelského a auditního indexu. V souboru es_connection.yml jsou k nalezení nastavení pro připojení k databázi, v souboru server.yml pak nastavení CORS pro validní komunikaci se stranou klienta. Výchozí hodnoty jsou již konfigurovány.

Dále se zde nacházejí soubory data_loaders.yml a llm.yml se specifickým určením. Jak je podrobněji popsáno v kapitole 3.3.2, vývojář modulární části aplikace určené k rozšiřování zde deklaruje globální vlastnosti a nastavení specifické pro jeho rozšíření. Veškerá logika související s

modely umělé inteligence, včetně té výchozí vytvořené již v rámci prototypu, je pojata jako modulární rozšíření, jelikož hlavním přínosem aplikace je možnost snadného vložení logiky pro nové rozličné moduly a porovnání jejich vlastností a výkonu.

5.3 Jádro aplikace

Jádro aplikace není zamýšlena jako modulární a není určena k přímému zásahu v rámci rozšiřování její AI kapacity (byť se tato možnost nevylučuje a kód je zdokumentován, logika a konfigurace nejsou pro tento účel optimalizovány). Relevantní soubory se nachází v podadresářích pod „application/abl“ a zajišťují zejména přihlašování a odhlašování uživatele a správu sezení, dále načítání nastavení uvedených v předchozí kapitole a jejich distribuci napříč konzumujícími objekty, kdy většina z nich implementuje návrhový vzor Singleton, volání metod a poskytování a získávání dat ze sekce DAO, pokročilou kontrolu dat obdržených v klientském requestu nad rámec možností FastAPI a zajištění validního zpracování dat a jejich předání routeru pro zaslání formou http response oprávněnému klientovi. Tato část také podle obsahu klientského dotazu rozlišuje, jaká volání do modulární části jsou relevantní, a provádí je.

5.4 Router

Pro zpracování klientských požadavků a poskytnutí odpovědí je použit framework FastAPI. Tento mimo jiné disponuje funkcí automatické geneze dokumentace API, čehož aplikace využívá a tuto dokumentaci zpřístupňuje na endpointu **/docs**. API je navrženo tak, aby bez změny jeho specifikace bylo možné přímo používat rozšíření vytvořená vývojářem nebo testerem libovolného modulu a rovněž specifikovat, zda má být v rámci databázových volání využít inference pipeline nad nativním ebmeddovacím modelem ELSER. Vizte kapitolu 3.3.2.1 pro příklad volání při použití konkrétních embeddovacích a LLM modelů.

5.5 DAO

Moduly v adresáři DAO slouží k překladu klientských dotazů na konečnou podobu dotazů do databáze. Jak je uvedeno v kapitole 3.2, datovou vrstvu tvoří databáze Elasticsearch. Tato má definované a zdokumentované REST API [28] jehož prostřednictvím s ní komunikují rovněž ostatní komponenty sady Elastic stack. Dotazy do databáze mají podobu HTTP dotazů, kde adresa zpravidla obsahuje endpoint ke konkrétnímu indexu, tělo pak obsahuje další dílčí parametry jako pravidla filtrování a další specifikace konkrétních dokumentů. Aplikační vrstva nad to umožňuje svázat konkrétní data k uživatelským jménem a zpřístupnit je pouze oprávněnému uživateli. Toho je docíleno prostým obohacením dat o další klíč s hodnotou specifikující vlastníka dat.

5.6 Stávající modely a rozšiřování aplikace

Modulární část aplikace je určena k jejímu rozšiřování formou implementace logiky nad modely umělé inteligence. V rámci prototypu obsahuje podporu pro konkrétní modely z komunitní platformy HuggingFace a enterprise platformy Cohere. Nachází se v adresáři „application/modules“, kde jsou umístěny 3 Pythonové moduly se třídami s rozdílným určením. Jedná se o rodičovské třídy definující minimální rozhraní, bez něhož nebude hlavní logika aplikace správně fungovat.

Hlavní třída každého rozšíření kterékoliv podsekce by měla být přímým potomkem jedné z těchto rodičovských tříd a musí být umístěna v relevantním podadresáři jako modul se stejným jménem. Jak je uvedeno v kapitole 3.3.1.1, vývojář modulární části v konfiguraci uvádí název modulu a jméno hlavní třídy. Jak je uvedeno v kapitole 3.3.1.4, v rámci těla HTTP dotazu je pak specifikováno, o jaký typ zdroje dat, embeddingu nebo LLM se jedná. Hlavní aplikační logika vychází z modulární konfigurace, pokusí se importovat modul daného jména a vytvořit instanci dané třídy. Pokud tyto neexistují, pokusí se následně importovat a vytvořit entity ze záložního nastavení (sekce default). Nezdaří-li se ani to, zpracování dotazu skončí chybou, která je odeslána klientovi. Nad to jako parametry konstruktoru předává hlavní třídě veškerá nastavení uvedená v relevantním YAML objektu v konfiguraci.

Tato sekce je založena na využití frameworku LangChain, o němž již byla zmínka v teoretické části práce. Veškeré zde implementované principy odpovídají logice tohoto frameworku a v rámci rozšiřování se předpokládá následování stejných principů. Nicméně třídy, z nichž mají rozšíření dědit, jsou navržené tak, aby následování metod LangChain nebylo nutností, v každém případě je však výhodou.

- Data loaders – Moduly pro načítání dat. Tímto je myšleno získávání dat ze zdrojů, kdy zdrojem může být prostý text, strukturovaný formát jako JSON, XML, CSV, webová stránka či rekurzivní následnictví stránek do definované hloubky a jiné. Ve výchozím stavu prototypu je implementována podpora pro CSV, JSON, PDF, text a webovou stránku či jejich rekurzi. Především je třeba přetížit asynchronní metodu load_data a zde zajistit načtení dat ze zdroje, jejich rozdělení (vizte data chunking v teoretické části práce) a vrácení seznamu takto připravených textů volající metodě z jádra aplikace.
- Embeddings – Modul pro embeddování dat. Embeddována budou buď data připravená v rámci jejich načtení ze zdroje za účelem jejich uložení nebo uživatelský dotaz za účelem získání takových dat. Zájmová metoda pro přetížení je zde asynchronní get_embedded_documents() zdokumentovaná v modulu embedding.py. Jejím výsledkem musí být výstup typu seznamu embeddovaných dokumentů ve formátu dense vector, a dále pak rovněž asynchronní metoda get_embedded_query() zdokumentovaná ve stejném

modulu, určená pro embeddování uživatelských dotazů nebo hypotetických dotazů vygenerovaných LLM. V rámci vytvoření instance se rovněž zavádí patřičný AI model definovaný v nastavení.

- LLMs – Modul pro komunikaci s jazykovým modelem. Z celé modulární logiky je tato nejobsáhlejší a řadí se sem proto nejvíce metod k případnému přetížení, není-li následována logika LangChain nebo vrací-li jazykový model odpovědi v neočekávané struktuře. Všechny metody jsou zevrubně zdokumentovány přímo v kódu.
  - expand_context_with_new_data() – metoda je volána, pokud klient v těle dotazu zadal získání nových dokumentů z databáze, tedy nikoliv pouze komunikaci nad již získaným kontextem. Zajišťuje rozšíření stávajícího kontextuálního okna s LLM o nové dokumenty, jejich seřazení, aktualizaci historie a nepřekročení definované velikosti okna.
  - expand_context_without_new_data() – metoda je volána, pokud klient v těle dotazu nezadal získání nových dokumentů z databáze, a tedy požaduje konverzaci nad dokumenty již získanými a uloženými v historii konverzace a stávajícím kontextuálním oknu.
  - get_hypothetical_query() – Pokud klient v těle dotazu zadal získání nových dokumentů, logika jádra volá tuto metodu pro genezi hypotetického dotazu před jejím faktickým embeddingem a zaslání výsledného vektoru v rámci dotazu do databáze.

Ve stávající podobě podporuje aplikace dvě AI platformy nabízející jak sadu embeddovacích, tak LLM modelů. S ohledem na finanční dostupnost a šíři nabízených možností byly zvoleny platformy Cohere a HuggingFace. Jednou z hlavních předností aplikace pro testování modelů je právě podpora HuggingFace, jelikož tuto lze vnímat jako jeden z největších uzlů nabízejících množství modelů mnoha kategorií. Vývojář či tester tak není nucen psát celou vlastní třídu, jelikož, jak lze vidět v podadresářích application/modules, moduly pro HuggingFace již existují. V rámci testování tak lze v HTTP dotazu pouze specifikovat tyto moduly klíčovým slovem „hf“ (v souladu s application/config/llm.yml) a jména konkrétních zájmových modelů uvést pod klíči `embeddingModel` a `llmModel`, což povede k přepsání výchozích jmen modelů v nastavení a jejich automatickému stažení k lokálnímu provozu. Je však třeba uvážit, že modely poskytované platformou HuggingFace nesdílí nutně formát výstupu, je tedy pravděpodobné, že vývojář bude muset takové odlišnosti ošetřit ve stávajících metodách nebo za tím účelem doplnit metody nové. Injektováním vlastního modulu, jak je popsán v kapitole 3.3.2.1, může případně změnit celý proces.

Platforma Cohere rovněž nabízí několik embeddovacích i LLM modelů, její omezení však spočívá v tom, že oproti HuggingFace obsahuje pouze modely z dílny společnosti Cohere, a v rámci

licence zdarma omezuje počet volání API v čase, což se zejména ve věci asynchronního embeddingu chunkovaných dokumentů ukazuje jako problematické. Výhodou však je konzistence výstupu, tedy v rámci testování stačí v HTTP dotazu pouze změnit jména modelů bez nutnosti dalších úprav na úrovni metod.

5.6.1 Příklady

V tomto příkladu vývojář testuje nový hypotetický embeddovací model my-embedding-model a jazykový model my-chatting-model s vlastním nastavením systémové zprávy pro klasický chat jakož i pro tvorbu hypotetických dotazů. Jeho nastavení, kód a následné volání API bude tedy vypadat následovně:

application/modules/embeddings/my-embedding-model.py

```
from modules.embedding import Embeddings
from my-model-module import MyModel

class MyEmbeddingModel(Embeddings):
  def __init__(self, config: dict, model_name: str = None):
    super().__init(config)
    if model_name is not None:
      self._model = MyModel(model_name, ……)
    else:
      self._model = MyModel(config[,model‘], ……)

  async def get_embedded_documents(self, ……) → list:
    # pretizeni metody, pokud rodicovska nevyhovuje

  async def _aembed_query(query: str) → list:
    # pretizeni metody k asynchronnimu ebmeddingu query, pokud
    # MyModel neodpovida API LangChain
```

application/modules/embeddings/my-llm.py

```
from modules.llm import LLM
from langchain_core.messages: AIMessage
from my-llm-module import MyLLM

class MyChatModel(LLM):
  def __init__(self, config: dict, model_name: str = None):
    super().__init(config)
    if model_name is not None:
      self._model = MyLLM(model_name, ……)
    else:
      self._model = MyLLM(config[,model‘], ……)

  async def expand_context_with_new_data(self, ……) → dict:
     # pretizeni metody, pokud rodicovska nevyhovuje – rozsireni kontextu s
nove ziskanymi dokumenty
  async def expand_context_without_new_data(self, ……) → dict:
    # pretizeni metody, pokud rodicovska nevyhovuje – rozsireni kontextu bez
nove ziskanych dokumentu

  async def get_hypothetical_query(self, ……) → AIMessage:
    # pretizeni metody, pokud rodicovska nevyhovuje – ziskani hypotetickeho
query
```

```
  async def _ainvoke(self, prompt: list) → dict:
      # pretizeni metody k ziskani odpovedi od LLM, neni-li dodrzeno API
LangChain
```

application/config/llm.yml

```
embeddings:
  my-embedding-model:
    class_name: „MyEmbeddingModel“
    model: „my-model-name“
    …
llms:
  my-llm:
    class_name: „MyChatModel“
    model: „my-other-model-name“
    system_message: "Your are a professional data analyst. Consider 'Extra
data' in this chat history to be your only relevant data source."
   hypo_message: "You're a genius. If you don't know an answer, make
one up.“
    …
```

Http dotaz (vizte zvýrazněné hodnoty)**:**

POST https://my-domain/chat_query

```
{
      "sessId": "session-id-here",
      "index": "my-index-here",
      "source": ["text"],
      "query": "My question for LLM and into the database?",
      "chatId": "chat-id-here-if-exists",
      "embedding": "my-embedding-model",
      "embeddingModel": "sentence-transformers/all-MiniLM-L6-v2",
      "llm": "my-llm",
      "llmSource": "text",
      "sparseVector": true,
      "sparseModel": ".elser_model_2_linux-x86_64",
      "inferenceField": "text",
      "knnBoost": 1,
      "sparseBoost": 2,
      "onlyLlm": false
}
```

Na základě výše uvedené konfigurace je po přijetí backendem dotaz zkontrolován a předzpracován jádrem aplikace. Následně jsou vytvořeny instance tříd z modulů my-embedding-model a my-llm, konkrétně jsou tyto moduly importovány a jsou vytvořeny objekty ze tříd, které vývojář specifikoval v konfiguračním souboru llm.yml. Konstruktory těchto tříd dostanou v argumentu jako parametr config celou dílčí konfiguraci, jak ji vývojář uvedl v patřičné sekci, a jméno modelu, bylo-li v http dotazu specifikováno (v opačném případě je třeba, aby vývojář tento stav ošetřil a uvedl výchozí model vlastní, jak je uvedeno v příkladu výše). Tedy vyjma jména inicializační třídy může vývojář vytvořit další specifikace zcela volně dle svých potřeb a s celým tímto nastavením může následně v kódu pracovat jako s objektem, aniž by byl nucen jej manuálně

importovat. Dodržuje-li jeho model Langchain API, nemusí navíc přetěžovat žádné metody a může využít hotové metody rodičovské, v opačném případě může metody libovolně přetížit při zachování struktury výstupních dat, s nimiž následně opět operuje logika jádra aplikace. Vývojář či tester se tak může soustředit zcela na problematiku konkrétního testovaného modelu, aniž by musel stavět celé vlastní řešení.

6 Uživatelská vrstva – frontend

Frontendová vrstva je realizována formou webové aplikace vytvořené s použitím frameworku React, respektive NextJS, a obdobně jako u ostatních komponent této práce je možné ji nasadit jako samostatnou službu v operačním systému, separátní kontejner nebo na cloudové platformě. Tato vrstva nevyužívá všech funkcionalit aplikační vrstvy, ale je zamýšlena a implementována jako PoC chatovací aplikace nad již uloženými daty v databázi. Příprava a uložení dat jsou zde chápány jako administrátorská odpovědnost, koncový uživatel v datech toliko vyhledává a vede nad nimi konverzaci v rámci kontextového okna s jazykovým modelem.

Uživatelské rozhraní je rozděleno na hlavičku v horní části stránky, seznam historických konverzací uživatele s možností zahájení nové konverzace po levé straně a konečně hlavní a dominantní prvek celé stránky představující rámec pro samotnou konverzaci. Uživatel zde vkládá své dotazy pro jazykový model a po přijetí odpovědi je tato přidána do historie. Jelikož aplikační vrstva vrací kromě odpovědi jazykového modelu rovněž dokumenty, které byly poskytnuty modelu jako zdrojová data k analýze, může uživatel zobrazit i tyto dokumenty jako popup prvky odkazované v těle konverzace.

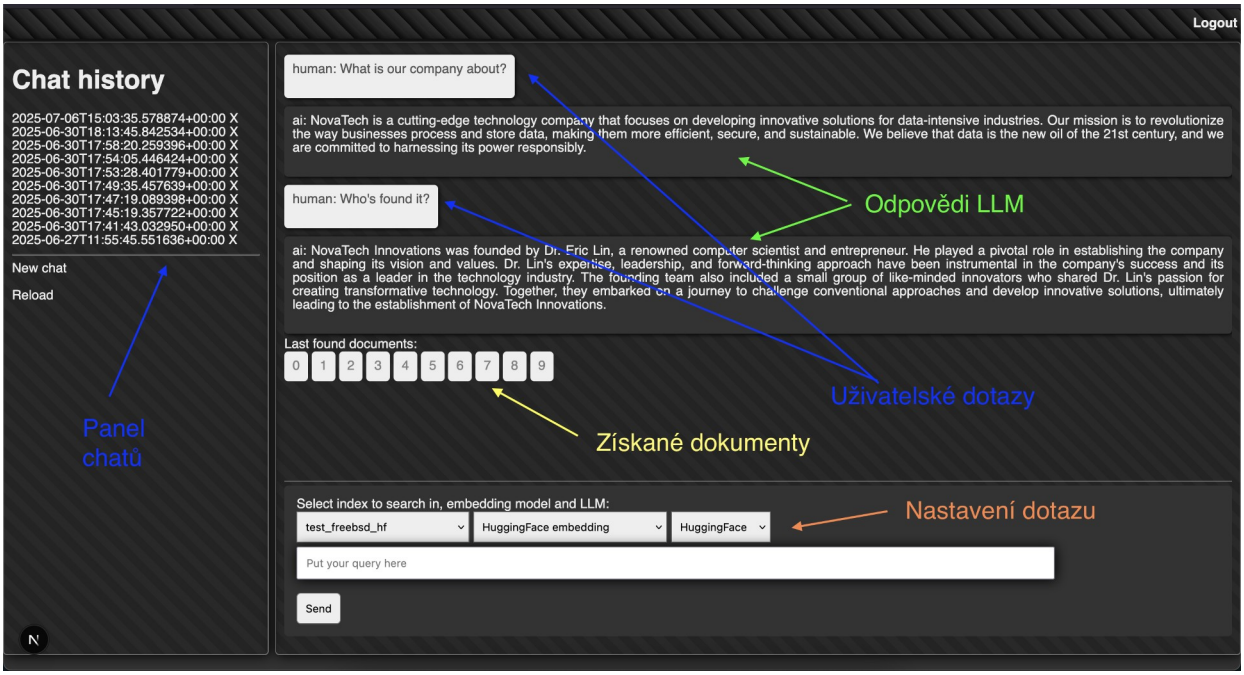
Frontend komunikuje s backendem prostřednictvím asynchronních http dotazů, kdy se nenačítá opakovaně celá stránka, ale pouze se mění obsah relevantních prvků. Toho je docíleno udržováním dílčích informací prostřednictvím komponenty ContextProvider a nativních Reactových mechanismů useContext a useState. Po kliknutí na specifikované prvky stránky jsou zahájena patřičná volání, získaná data jsou použita k nahrazení hodnot ve sledovaných proměnných, následně je komponenta překreslena s novým obsahem.

Frontend podporuje přihlašování uživatelů, což je i nezbytná funkce, jelikož součástí dotazů tak, jak jsou definovány v API backendu, je rovněž identifikátor sezení konkrétního uživatele. Tento je na straně klienta opět udržován cestou mechanismu useState a jeho hodnota je přidávána ke každému dotazu.

S ohledem na malý objem nutných konfiguračních direktiv probíhá nastavení frontendového klienta cestou proměnných prostředí NEXT_PUBLIC_BE_PROTOCOL a NEXT_PUBLIC_BE_ENDPOINT, kdy konkrétní dílčí cesty pro jednotlivá volání jsou již komponentami doplňovány fixně.

Po strukturální stránce rozlišuje aplikace dvě routy, jejichž logika a struktura je specifikována v adresáři frontend/pages. Routa **login** je použita pro stránku k přihlášení uživatele, routa **user** pak k rozhraní pro komunikaci uživatele již přihlášeného. Ostatní dílčí komponenty stránky představující samostatně definované a chovající se prvky, jakož i kontexty pro udržování proměnných dat relevantních pro celé uživatelské rozhraní, jsou definovány v patřičných podadresářích na cestě frontend/app. Vizuální stránka webové aplikace je realizována použitím kaskádových stylů CSS, s použitím tříd a unikátních identifikátorů ve smyslu HTML jakož i obecných předpisů pro komponenty dle jejich typu v adresáři frontend/app/styles.

Obrázek 8 – Screenshot a popis rozhraní



Zdroj: Vlastní aplikace

7 Nasazení vrstev v OS, kontejneru, na cloudu

Jednotlivé vrstvy projektu jsou navrženy a implementovány jako zcela oddělené komponenty, jenž jsou vzájemně propojeny pouze prostřednictvím definovaných API. Každá vrstva tak může být nasazena v libovolném prostředí jedním z následujících způsobů se z toho vyplývající sadou předností a úskalí:

- Jako služba v operačním systému.

- Jako separátní kontejnery.
- V cloudové platformě.
- Jako kombinace výše uvedených bodů.

Nasazení jako samostatné služby může být vhodné například v případě, kdy existuje dedikovaný stroj či sada strojů (serverů) pořízených či sestavených přímo za účelem provozu projektu nebo některé jeho komponenty. To lze očekávat zejména u datové vrstvy, kdy není neobvyklým případem existence sady serverů pořizovaných výhradně za účelem provozu clusterové databáze. Výhodou takového přístupu je zejména fakt, že neexistence virtualizační vrstvy znamená možnost využití všech hardwarových prostředků samotnou databází, vyjma drobného minima k provozu operačního systému. Nevýhodou je zejména nemožnost snadného horizontálního škálování, kdy rozšíření clusteru o další node představuje nutnost pořízení dalšího fyzického stroje včetně potřebného úložiště.

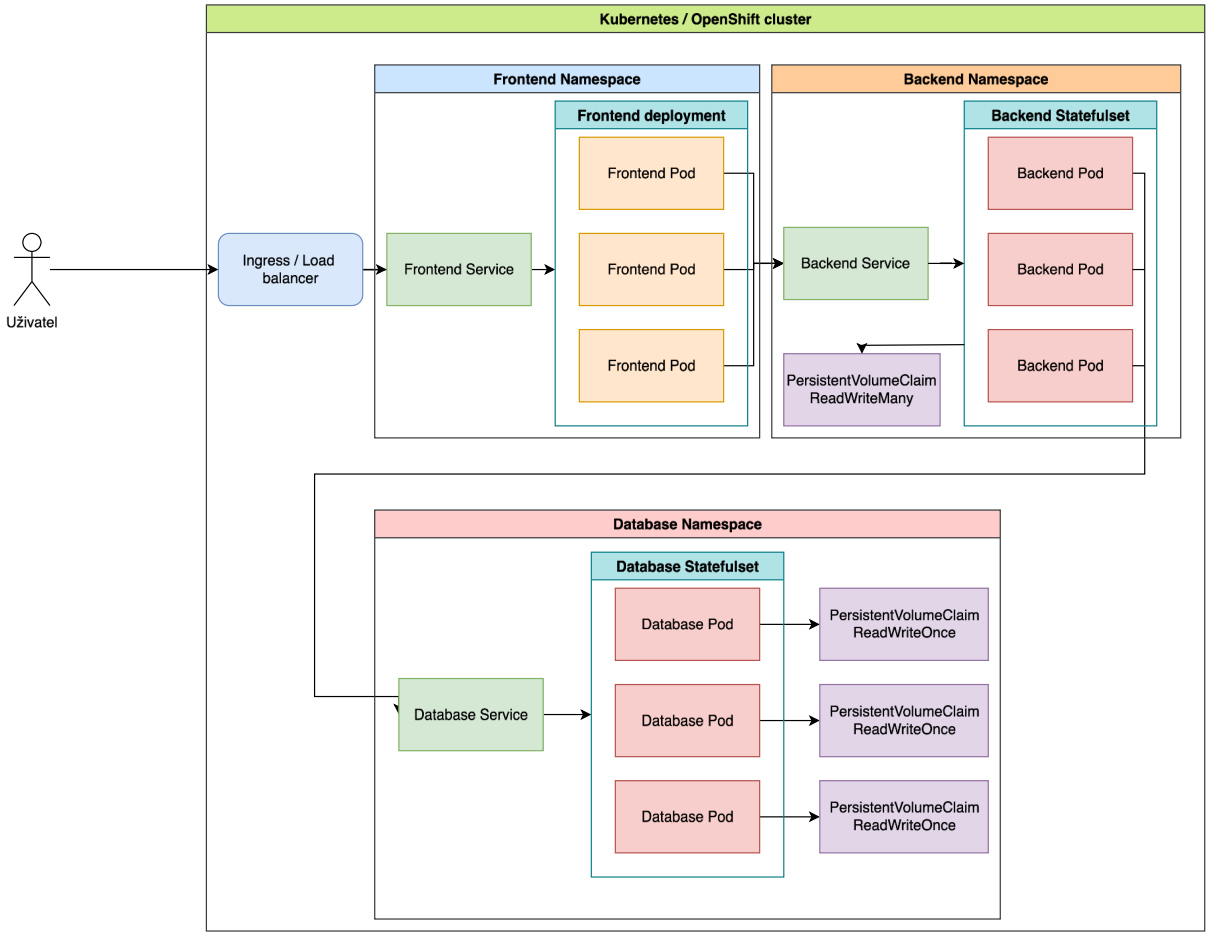
Nasazení cestou separátních kontejnerů přichází v úvahu zejména v případě, že existuje sada vyhrazených strojů, jejichž hardwarové možnosti však přesahují požadovanou kapacitu pro jednotlivé komponenty. Kontejnerizací komponent lze dosáhnout optimálního rozdělení hardwarových prostředků s možností horizontálního i vertikálního škálování v případě potřeby. Otázku lokálního úložiště pak lze řešit přidáním disků či použitím spolehlivého síťového úložiště s vysokou dostupností. Zde je třeba zejména uvážit redundanci dat, kdy některá síťová řešení tuto již obsahují a s ohledem na její nativní přítomnost v datové vrstvě je žádoucí ji na straně úložiště vypnout nebo zvolit řešení bez ní. Aplikační vrstva rovněž uchovává údaje k sezení uživatelů formou dočasných souborů, kdy je třeba zajistit jejich dostupnost pro všechny instance backendu. Součástí projektu je příklad kompletní konfigurace pro nasazení jednotlivých komponent jako kontejnerů platformy Docker cestou compose file, který je k nalezení v nejvyšším adresáři aplikace v souboru docker-compose.yml. Využívá automatickou genezi úložišť nabízenou jako nativní funkci Docker.

Nasazení aplikace na cloudové platformě vychází z možnosti popsané v předchozím bodě a sdílí tedy všechny její charakteristiky. Výhodou tohoto řešení je zejména automatizace nasazení na konkrétní stroje na základě anotací, možnost využití úložišť dle tříd a již vyřešená distribuce síťové zátěže formou služeb a Loadbalancerů, čímž se z celého řešení stává snadno škálovatelný, vysoce dostupný produkt. Součástí práce je sada YAML manifestů pro nasazení komponent na platformách Kubernetes a Openshift, která je k nalezení v nejvyšším adresáři aplikace v souboru modular-rag.yml. Úložiště jsou řešena cestou PersistentVolumeClaim s vazbou na potřebnou třídu zajišťující provisioning (tato by již měla na platformě existovat).

Za účelem demonstrace praktického nasazení a ověření funkčností jsou komponenty aplikace zpřístupněny na následujících URL:

- Datová vrstva – administrační rozhraní (definice indexů, inference pipelines, vizualizace a výsledky sebraných metrik) dostupné na https://kibana.freedev.cz/
- Aplikační vrstva – všechny endpointy dostupné na základní URL https://project2.freedev.cz (pro dokumentaci API formou Swagger vizte https://project2.freedev.cz/docs)
- Frontend – PoC chatovací aplikace dostupná na https://project.freedev.cz.

Obrázek 9 – Schéma možného nasazení v cloudu



Zdroj: vlastní zpracování

8 Měření a porovnání embeddovacích modelů a LLM

Pro měření výkonnosti jednotlivých modulů jakož i aplikace jako celku je využito její napojení na aplikační monitoring, jak je uvedeno v kapitole 3.2. APM agent (Pythonový modul) umožňuje extenzivní sledování aplikačních metrik – sleduje latenci pro jednotlivá volání, závislost

aplikace na ostatních službách (následná volání AI platforem a databáze), umožňuje trasování dotazů na jednotlivé endpointy a rozpad celého zpracování dotazu včetně indikace případných neošetřených výjimek, konkrétního těla databázových dotazů a kompletních hlaviček jednotlivých volání včetně geolokace jejich zdroje, to vše včetně vizualizací.

Při interpretaci výsledků je třeba vzít v potaz, že výkon lokálně hostovaných modelů odvisí od hardwarové kapacity infrastruktury, na níž byly provozovány, zatímco modely poskytované třetí stranou včetně jejich provozu mají kapacitu zcela odlišnou, do měření však vstupují parametry konektivity. Porovnání proto dává smysl pro větší počet modelů se shodným způsobem provozu.

Testování proběhlo na dvou datových sadách, kdy jednou byla kompletní dokumentace k operačnímu systému FreeBSD o přibližně 203895 tokenech, druhou byla série dokumentů o smyšlené technologické společnosti NovaTech Innovations o rozsahu přibližně 1250 tokenů. Ve všech případech byly modely LLM nastaveny systémovou zprávou tak, aby plnily roli datového analytika a za jediný zdroj dat byly považovány dokumenty zaslané volající stranou. Velikost kontextového okna činila 10 zpráv, sada získaných dokumentů 10 dokumentů.

8.1 Hardwarová specifikace stroje pro lokální testování modelů a nasazení aplikace

Lokální testování modelů získaných z platformy HuggingFace probíhalo na jediném stroji, kde byly jednotlivé komponenty nasazeny jako samostatné kontejnery s následujícími parametry:

Zařízení:

Tabulka 3 – Hardwarové parametry testovacího zařízení

CPU	RAM	Úložiště
Intel Xeon E3-1230 3.30 GHz 4 fyzická jádra HyperThreading	32 GB DDR3 ECC 1600 MT/s	SSD SATA 3.1 6.0 Gb/s

Zdroj: vlastní zpracování

Kontejnery:

Tabulka 4 – Hardwarové parametry kontejnerizovaných komponent

	CPU limit	RAM
Datová vrstva	limit: 2 x 2 pro datové nody, 1 x 1 pro machine learning	2 x 4 GB pro datové nody, 1 x 8 GB pro machine learning
Aplikační vrstva	4	8 GB

Frontend	0.5	2

Zdroj: vlastní zpracování

8.2 Porovnávané modely platformy HuggingFace

- Embedding
  - sentence-transformers/all-MiniLM-L6-v2
  - sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2
- LLM
  - Qwen/Qwen2.5-1.5B-Instruct
  - HuggingFaceH4/zephyr-7b-beta

8.3 Porovnávané modely platformy Cohere

- Embedding
  - embed-english-v3.0
  - embed-v4.0
- LLM
  - command-r-plus-08-2024
  - command-a-03-2025

8.4 Výsledky měření

Měření proběhlo na dvou datových sadách. Jednou datovou sadou je 203895

Embedding – výsledky embeddovacích modelů při chunkování a ukládání dat:

Tabulka 5 – Parametry měření a latence embeddovacích modelů – 20 opakování

Platforma	Model	Provoz	Dimenze	Chunking / Overlap při ukládání	Průměrná latence na asynchronní dotaz
Cohere	embed-english-v3.0	vzdálený	1024	512 / 64	273 ms
Cohere	embed-4.0	vzdálený	1536	512 / 64	326 ms
HuggingFace	sentence-transformers/all-MiniLM-L6-v2	lokální	384	512 / 64	56 ms
HuggingFace	sentence-transformers/ paraphrase-multilingual-MiniLM-L12-v2	lokální	384	512 / 64	26 ms
Elasticsearch	ELSER	lokální	N/A (sparse vector	512 / 64	32 ms

			jako dokument s proměnnou délkou)		

Zdroj: vlastní zpracování

Embedding – přesnost zpětně získaných dat (pořadí vrácených dokumentů na embeddovaný dotaz oproti pořadí pevně učenému tagy) při použití Simple retrieval metody. Měření přesnosti probíhalo na základě zasílání předem definované sady dotazů a otagování všech relevantních dokumentů, které by na takové dotazy měly být získány. Kvantifikace přesnosti je pak prostým procentuálním výpočtem.

Tabulka 6 – Přesnost získání embeddovaných dat

Platforma	Model	Provoz	Přesnost
Elasticsearch	ELSER	lokální	100 %
HuggingFace	sentence-transformers/all-MiniLM-L6-v2	lokální	96 %
HuggingFace	sentence-transformers/ paraphrase-multilingual-MiniLM-L12-v2	lokální	95 %
Cohere	embed-english-v3.0	vzdálený	96 %
Cohere	embed-4.0	vzdálený	96 %

Zdroj: vlastní zpracování

LLM – výkon chatovacích modelů

Tabulka 7 – Latence chatovacích modelů

Platforma	Model	Provoz	Hypotetické query	Hlavní query
Cohere	command-r-plus-08-2024	vzdálený	3952 ms	10213 ms
Cohere	command-a-03-2025	vzdálený	2781 ms	2153 ms
HuggingFace	Qwen/Qwen2.5-1.5B-Instruct	lokální	262 ms	1817 ms
HuggingFace	HuggingFaceH4/ zephyr-7b-beta	lokální	285 ms	1915 ms

Zdroj: vlastní zpracování

Výsledky chatovacích modelů se dále liší co do formy podání odpovědi, kdy lidově řečeno můžeme modely z platformy Cohere označit za „upovídanější“, což je obtížně kvantifikovatelná proměnná, a do jedné odpovědi zahrnují větší rozsah získaných dokumentů, zatímco testované

modely z platformy HuggingFace jsou selektivnější a jejich odpovědi jsou více faktické a méně květnaté, ve faktickém obsahu však nebyl nalezen rozdíl.

Při volbě modelu patří mezi zásadní parametry rovněž cena, kdy modely z platformy HuggingFace lze obecně označit za dostupné zdarma. Platforma Cohere nabízí modely zdarma pro vývojářské a testovací účely, pro produkční provoz je jejich použití zpoplatněno. ELSER model z dílny společnosti Elastic je rovněž vázán placenou licencí, zdarma je možné jej využít pouze v rámci 30 denní trial licence.

Na praktických testech se záměnou modelů na podporovaných platformách tedy bylo ověřeno, že aplikace je schopná správně pracovat s daty, komunikovat s externími modely a že je rozšiřitelná na modulární bázi o modely nové pouhou změnou hodnot beze změny API, a měřením prostřednictvím APM subsystému se podařilo získat konkrétní metriky o chování aplikace i specifických modelů.

9 Budoucnost projektu

V rámci dalšího rozvoje projektu je v první řadě nasnadě rozšíření výchozího počtu připravených AI platforem o další enterprise produkty jako Amazon Bedrock či OpenAI. Dále lze uvažovat o zpracování dat přijatých k uložení na pozadí. Ve stávající podobě klient vyčkává na zpracování, embedding a uložení dat, jelikož aplikační vrstva mu odpověď o výsledku zasílá až po dokončení procesu. Pokud ovšem klient zašle velké množství dat nebo odkáže na vzdálená data typu webové stránky, kterou může navíc přidávat i rekurzivně do stanovené hloubky odkazů, může takový proces trvat dlouhou dobu, což v konečném důsledku skončí indikací timeoutu na straně klienta, ačkoliv proces na straně backendu stále běží a data jsou zpracovávána a ukládána. Za tímto účelem bude vhodné spustit proces získání a přípravy dat na pozadí a klientské straně zaslat odpověď s identifikátorem úlohy, která byla spuštěna. V rámci API pak bude třeba definovat nový endpoint umožňující kontrolu stavu těchto úloh. V rámci rozvoje frameworku LangChain lze do budoucna očekávat větší důraz na splňování jeho standardního API ze strany všech podporovaných platforem, dokumenty získané z databáze a analyzované modelem pak mohou být do LLM zasílány cestou připojených metod namísto jejich vkládání přímo do konverzace. Následujícím krokem může být podpora více jazyků či překlad dokumentů a dotazů do jazyka referenčního s ohledem na omezenou podporu řady stávajících embeddovacích modelů pro méně zastoupené jazyky.

V rámci rozvoje frontendové části lze zejména uvažovat o administrátorském rozhraní, jelikož stávající verze podporuje toliko využívání běžných uživatelských funkcí jako chat nad daty a

zobrazování zdrojových dokumentů, ale neumožňuje již data ukládat. Potenciálním rozšířením může být rovněž částečná možnost zásahu přímo do databáze. Jelikož součásti nasazení je rovněž komponenta Kibana, nejsou takové funkce součástí frontendu, je však možné identifikovat minimální sadu nutných funkcí pro potřeby administrátora, tyto zapracovat do frontendu s voláním přímo do databáze skrz její REST API a Kibanu z projektu zcela odstranit, kdy zřejmou výhodou takového řešení by bylo ušetření hardwarových prostředků, nevýhodou je pak opakovaná implementace funkcí již implementovaných jinde.

Samostatnou kategorií je řešení identifikace a uchovávání uživatelů, kdy v dalších verzích aplikace lze přidat v první řadě širší možnosti specifikace uživatelských vlastností a podporu jejich uložení i v jiných databázích než pouze v Elasticsearch včetně databází relačních. Přihlášení uživatele lze dále rozšířit o možnosti jako OAuth2.0 či jiný model SSO a zcela tak odstranit nutnost udržování vlastních uživatelských účtů.

Závěr

V práci se podařilo zdokumentovat hlavní stávající trendy v oblasti vývoje aplikací pracujících s velkými daty a popsat aktuální stav a principy různých typů modelů umělé inteligence z high-level pohledu včetně jejich funkce jakož i obecně vysvětlit propojení těchto komponent do jednoho celku k vytvoření portálu či platformy umožňující rychlé vyhledávání a orientaci v datech a konverzaci o nich za účelem získání kontextu a vytěžení maximálního užitku z těchto dat. Byly popsány principy uchovávání dat v podobě čitelné pro modely umělé inteligence, metody jejich zpětného získávání včetně vlivu na přesnost výsledků a podpory těchto principů ve stávajících databázových systémech s přihlédnutím k relačním i objektovým databázím spolu s odkazy na výsledky konkrétních měření. Byly popsány stěžejní kategorie a principy fungování velkých jazykových modelů umožňující kontextuální dialog nad daty, metody jejich nastavování a využití jejich vlastností již při získávání dat jakož i k jejich analýze a vyhodnocování v rámci dat již získaných. Byly popsány metody učení AI modelů a pozornost byla věnována i etické a právní stránce této přelomové technologie. Byly popsány přední programovací jazyky, knihovny a frameworky nabízející srozumitelné API a patřičný výkon k vytváření aplikací pracujících s modely umělé inteligence, umožňující jejich učení či využívání modelů již vytrénovaných. Dále byly popsány přední současné platformy nabízející funkce AI jako hotové řešení a platformy umožňující lokální provoz modelů včetně jejich knihoven pro zpřístupnění těchto funkcí vývojářům. Teoretickou část zadání se tedy podařilo splnit.

V rámci praktické části práce se podařilo vytvořit prototyp RAGové aplikace využívající třívrstvý model. Datová vrstva je založena na objektovém databázovém systému s pokročilou nativní podporou principů zpracování a získávání dat v podobě přístupné pro embeddovací modely, kdy za účelem zajištění spolehlivosti a vysoké dostupnosti byl zvolen clusterový databázový systém Elasticsearch. Aplikační vrstva je implementována v jazyce Python s použitím frameworku FastAPI pro realizaci standardizované a bezpečné komunikace aplikace se světem prostřednictvím REST API. Tato je vnitřně rozdělena na třídy zajišťující routování klientských HTTP požadavků a relevantních odpovědí, třídy logiky jádra aplikace, třídy komunikující přímo s databází a třídy obsluhující problematiku umělé inteligence využívající framework LangChain, které jsou vyčleněny jako modulární subsystém aplikace s možností snadného rozšíření o podporu libovolné další platformy a libovolných dalších modelů bez nutnosti změn v logice jádra nebo API aplikace (ve smyslu REST API), při poskytnutí srozumitelného, flexibilního vývojářského API (ve smyslu sady rodičovských tříd a metod). Frontendová vrstva je implementována jako webová aplikace v jazyce Javascript s použitím frameworku React, kdy stránka je rozdělena na komponenty a využívá asynchronních

volání aplikační vrstvy na pozadí pro získávání nových informací a překreslování pouze těch komponent, jejichž obsah se změnil při rozdělení základního datového toku cestou rout. Jelikož práce není zaměřena na frontend, je tento pojat jako Proof of Concept chatovací funkce projektu. Dále byly vytvořeny předpisy nasazení aplikace pro virtualizovaný provoz formou separátních lokálních kontejnerů, jakož i nasazení na cloudových platformách. Do aplikační vrstvy byla integrována funkce Application Performance Monitoring zasílající provozní metriky rovněž do databáze Elasticsearch a umožňující měření výkonu a latence aplikace, a tedy potažmo výkonu a rychlosti odezvy AI modelů, ať už modelů embeddovacích nebo LLM, která byla použita k porovnání konkrétních modelů v kapitole 3.6.4. a k demonstraci snadnosti rozšíření aplikace včetně poskytnutí výkonnostních výsledků pro srovnání modelů. Praktickou část zadání se tedy rovněž podařilo splnit.

Hlavní přínos práce tak spočívá nejen v poskytnutí uchopitelného a dostatečně hlubokého náhledu do problematiky vývoje aplikací s AI funkcionalitami včetně principů umělé inteligence jako takové pro vývojáře aplikací či datové inženýry se zájmem o toto téma, ale také v poskytnutí konkrétního nástroje umožňujícího snadné porovnání specifických modelů bez nutnosti hluboké znalosti programování, který je zároveň dostatečně flexibilní a snadno rozšiřitelný a neomezuje tak pokročilejší vývojáře ve vlastní tvorbě navazující na již hotovou práci.

Seznam použitých zdrojů

1. Geeksforgeeks, *10 Best Databases for Machine Learning AI [2025]*, Online, Dostupné z https://www.geeksforgeeks.org/databases-for-machine-learning-ai [citováno 2025-03-11]
2. OpenAI, *Introducing ChatGPT*, Online, Dostupné z https://openai.com/index/chatgpt [citováno 2025-03-11]
3. Orca Security, *10 Most Popular AI Models in 2024*, Online, Dostupné z https://orca.security/resources/blog/top-10-most-popular-ai-models-2024 [citováno 2025-03-11]
4. Mendix, *What Are Different Types of AI Models*, Online, dostupné z https://www.mendix.com/blog/what-are-the-different-types-of-ai-models [citováno 2025-03-14]
5. IT Chronicles, *What is NLP*, Online, Dostupné z https://www.ibm.com/think/topics/natural-language-processing [citováno 2025-03-10]
6. Huggingface, *Models*, Online, Dospuné z https://huggingface.co/models?pipeline_tag=text-generation&sort=likes [citováno 2025-03-11]
7. SuperAnnotate, *Introduction to Diffusion Models for Machine Learning*, Online, Dostupné z https://www.superannotate.com/blog/diffusion-models [citováno 2025-03-14]
8. Arxiv, *Generative Adversartial Networks*, Online, Dostupné z https://arxiv.org/abs/1406.2661 [citováno 2025-03-14]
9. Arxiv, *Attention Is All You Need*, Online, Dostupné z https://arxiv.org/abs/1706.03762 [citováno 2025-03-14]
10. Cohere, *Cohere's Embedded Models*, Online, Dostpuné z https://docs.cohere.com/v2/docs/cohere-embed [citováno 2025-03-10]
11. Cohere, *Pricing*, Online, Dostupné z https://cohere.com/pricing [citováno 2025-03-11]
12. Symbl.ai, *An In-depth guide to Benchmarking*, Online, Dostupné z https://symbl.ai/developers/blog/an-in-depth-guide-to-benchmarking-llms [citováno 2025-13-11]
13. Huggingface, *The Big Benchmarks Collection*, Online, Dostupné z https://huggingface.co/collections/open-llm-leaderboard/the-big-benchmarks-collection-64faca6335a7fc7d4ffe974a [citováno 2025-03-11]
14. Atreum, *AI a autorské právo*, Online, Dostupné z https://atreum.cz/ai-a-autorske-pravo [citováno 2025-03-14]
15. GitHub, *pgvector*, Online, Dostupné z https://github.com/pgvector/pgvector [citováno 2025-03-21]

16. MariaDB, MariaDB Vector, Online, Dostupné z https://mariadb.org/projects/mariadb-vector/ [citováno 2025-03-21]
17. MongoDB Docs, *Run Vector Search Queries*, Online, Dostupné z https://www.mongodb.com/docs/atlas/atlas-vector-search/vector-search-stage/ [citováno 2025-03-21]
18. Upwork, *10 Best Programming Languages To Know in 2025*, Online, Dostupné z https://www.upwork.com/resources/best-ai-programming-language [citováno 2025-03-18]
19. Elitex, *JavaScript and AI*, Online, Dostupné z https://elitex.systems/blog/javascript-and-ai-is-javascript-a-good-programming-language-for-creating-artificial-intelligence [citováno 2025-03-18]
20. DZone, *Java for AI*, Online, Dostupné z https://dzone.com/articles/java-for-artificial-intelligence [citováno 2025-03-18]
21. Weka Wiki, *Use weka in your java code*, Online, Dostupné z https://waikato.github.io/weka-wiki/use_weka_in_your_java_code/ [citováno 2025-03-22]
22. Deeplearning4j, Beginners, Online, Dostupné z https://deeplearning4j.konduit.ai/multi-project/tutorials/beginners [citováno 2025-03-22]
23. BairesDev, Online, Dostupné z https://www.bairesdev.com/blog/best-java-machine-learning-libraries/ [citováno 2025-03-22]
24. GitHub, AbeelLab/javaml, Online, Dostupné z https://github.com/AbeelLab/javaml [citováno 2025-03-22]
25. Huggingface, *Libraries,* Online, Dostupné z https://huggingface.co/docs/hub/models-libraries [citováno 2025-03-24]
26. Cohere, Retrieval Augmented Generation, Online, Dostupné z https://docs.cohere.com/docs/rag-quickstart [citováno 2025-03-24]
27. Datacamp, *The Top 16 AI Frameworks and Libraries*, Online, Dostupné z https://www.datacamp.com/blog/top-ai-frameworks-and-libraries?dc_referrer=https%3A%2F%2Fwww.google.com%2F [citováno 2025-03-24]
28. Elastic, *REST APIs*, Online, dostupné z https://www.elastic.co/docs/reference/elasticsearch/rest-apis [citováno 2025-06-29]

Seznam obrázků

Seznam tabulek

Seznam grafů

Seznam příloh

Příloha A – Zdrojový kód

Zdrojový kód aplikace na přiloženém paměťovém médiu označeném RAG – data v adresářích „application“ (backend) a „frontend“.

Příloha B – Konfigurace

Dokumentace a příklady konfigurace související s nasazením aplikace na přiloženém paměťovém médiu označeném RAG – data v adresáři „configs“.

Příloha C – Nasazení

Dokumentace postupu pro nasazení aplikace na úrovni služby v operačním systému, virtualizovaného kontejneru a na cloudové platformě na přiloženém paměťovém médiu označeném RAG – data v kořenovém adresáři v souborech „docker-compose.yml“, „*.dockerfile“ (pro virtualizační platformu Docker) a „modular-rag.yml“ (pro platformy Kubernetes a Openshift).