FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Vyhledávací systém ve firemní dokumentaci
Student:	Jan Wenzel
Vedoucí:	Ing. Jiří Novák, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Cílem práce je navrhnout a implementovat backend webové aplikace, která by měla nahradit část firemní agendy (např. dotazy zaměstnanců na personálním oddělení). Aplikace bude fungovat na principu vyhodnocování textového dotazu zadaného uživatelem vůči firemní dokumentaci (např. příručka zaměstnance, interní předpisy nebo informace o firemních benefitech), která bude uložena v cloudu (např. ve formátu pdf nebo docx).

1. Nastudujte metody pro vyhledávání v textu.
2. Analyzujte existující softwarová řešení, která by bylo možné využít pro vyhledávání ve firemní dokumentaci. Zhodnoťte jejich výhody a nevýhody.
3. Navrhněte architekturu backendové části aplikace a zaměřte se zejména na její budoucí rozšiřitelnost.
4. Navrhněte jednoduchý frontend a REST API pro testování aplikace.
5. Aplikaci implementujte. Pro backend použijte jazyk Java nebo Kotlin.
6. Výsledné řešení otestujte.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 15. ledna 2024 v Praze.

Bakalářská práce

VYHLEDÁVACÍ SYSTÉM VE FIREMNÍ DOKUMENTACI

Jan Wenzel

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jiří Novák, Ph.D.
16. května 2024

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat především vedoucímu práce Ing. Jiřímu Novákovi, Ph.D. za podnětné konzultace a vedení bakalářské práce, oponentovi Ing. Jiřímu Hlásnému, MBA za vytvoření příjemného zázemí pro psaní této práce a cenné rady k implementaci. Velké díky patří mé rodině a přátelům, kteří mě neustále podporovali během celého studia. Na závěr bych chtěl poděkovat své přítelkyni, která mi byla oporou po celou dobu tvorby této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 16. května 2024

Abstrakt

Práce se zabývá návrhem a implementací webové aplikace pro vyhledávání ve firemní dokumentaci. Práce začíná analýzou stávajících metod pro vyhledávání v textu a rešerší existujících řešení. Jejich výhody a nevýhody jsou diskutovány jako základ pro návrh systému. Dále je navržena architektura backendu aplikace s důrazem na modulárnost, rozšiřitelnost a integraci s cloudovými úložišti. Implementované řešení využívá jazykové modely RoBERTa a Llama3, pro efektivní vyhledávání v rozsáhlém množství nestrukturovaných textových dokumentech. Aplikace, napsaná v jazyce Java, využívá Spring Boot framework a vystavuje REST API, pomocí kterého lze vyhledávat relevantní informace a převádět dokumenty na vektorovou reprezentaci, která se uloží ve vektorové databázi Qdrant. Implementované řešení podporuje komunikaci s dokumenty, uloženými v cloudovém úložišti Google Drive i přes UI rozhraní.

Klíčová slova webová aplikace, Retrieval Augmented Generation, sémantické vyhledávání, embeddingy, RoBERTa, Qdrant, Llama3, Java, Spring AI

Abstract

The thesis focuses on designing and implementing a web application for searching in enterprise documentation. The work begins with an analysis of existing text search methods and a review of current solutions. Their advantages and disadvantages are discussed as a basis for the system design. The application's backend architecture is also proposed, emphasizing modularity, scalability, and integration with cloud storage. The implemented solution utilizes RoBERTa and Llama3 language models to search through numerous unstructured text documents efficiently. The application, written in Java, uses the Spring Boot framework and exposes a REST API. This API enables searching for relevant information and converting documents from Google Drive into a vector representation stored in the Qdrant vector database. User interaction is also supported through a UI.

Keywords web application, Retrieval Augmented Generation, semantic search, embeddings, RoBERTa, Qdrant, Llama3, Java, Spring AI

Seznam zkratek

API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
BOW	Bag of Words
CBOW	Continuous Bag of Words
DTO	Data Transfer Object
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
LLM	Large Language Model
MLM	Masked Language Model
NLP	Natural Language Processing
NSP	Next Sentecne Prediction
ONNX	Open Neural Network Exchange
OOV	Out-of-vocabulary
RAG	Retrieval Augmented Generation
REST	Representational State Transfer
RoBERTa	Robustly Optimized BERT Pretraining Approach
TF-IDF	Term Frequency–Inverse Document Frequency
URI	Uniform Resource Identifier
XML	Extensible Markup Language

Úvod

V digitálním věku, kdy se množství nestrukturovaných textových dat neustále zvyšuje, se efektivní vyhledávání a přístup k relevantním datům stává klíčovým pro úspěšné fungování podniků a organizací. Tato práce se zaměřuje na návrh a implementaci backendové části webové aplikace, která má za cíl zefektivnit vyhledávání ve firemní dokumentaci. S rostoucím počtem dokumentů, jako jsou manuály, interní předpisy, nebo informace o benefitních programech, se stává pro zaměstnance náročné udržet si přehled a rychle nalézt potřebné informace. Řešení, které bude v práci představeno, využívá moderních technologií v oblasti strojového učení, zejména jazykových modelů RoBERTa a Llam3, aby umožnilo efektivní a přesné vyhledávání v rozsáhlém množství textových dat.

V první části práce budou představeny a zkoumány různé metody vyhledávání v nestrukturovaném textu. Dále bude provedena rešerše existujících softwarových řešení zabývajících se vyhledáváním v textu, budou zhodnoceny jejich výhody a nevýhody. Výsledná analýza bude sloužit jako základ pro návrh systému.

Ve druhé části práce bude navržena architektura backendu aplikace s důrazem na modulárnost, rozšiřitelnost a integraci s moderními cloudovými úložišti. Řešení bude využívat Qdrant vektorovou databázi pro ukládání a rychlé vyhledávání předzpracovaných embeddingů, což zefektivní práci s velkými objemy textových dat. Součástí práce bude také návrh jednoduchého frontendu a REST API, které se využije k otestování a demonstraci funkcionalit aplikace.

Implementace systému, provedená v programovacím jazyce Java, bude demonstrovat praktickou aplikovatelnost navrženého řešení a jeho přínos pro efektivní vyhledávání ve firemní dokumentaci.

Závěrem práce budou reflektovány dosažené výsledky a diskutovány přínosy webové aplikace.

Kapitola 1

Analýza

V této analytické části bude pozornost věnována možným přístupům vyhledávání v textu, které stojí v jádru moderních systémů pro zpracování a vyhledávání informací. Jedná se o způsob **lexikálního vyhledávání** a **sémantického vyhledávání**. Oba způsoby přináší unikátní přístup k interpretaci a nalezení dat ve velkých textových korpusech. Přestože se na první pohled mohou přístupy jevit jako podobné, odlišují se zásadními způsoby, které mají významný dopad na efektivitu, přesnost a relevanci vyhledávaných výsledků. V následujících oddílech budou přiblíženy teoretické základy obou těchto přístupů, dále budou zkoumány jejich klíčové vlastnosti, výhody a nevýhody.

1.1 Lexikální vyhledávání

Lexikální vyhledávání, známé také jako vyhledávání založené na klíčových slovech (keyword-based search), je základní metodou pro nalezení informací v databázích, na webech nebo v jakémkoli strukturovaném či nestrukturovaném textovém souboru. Tento proces spočívá v porovnávání klíčových slov nebo frází zadaných uživatelem s textem v dokumentové databázi k identifikaci relevantních záznamů nebo informací na základě textové shody, nikoliv však na základě kontextu či záměru dotazu uživatele [1, 2].

- **`Booleovské vyhledávání`** umožňuje uživatelům kombinovat klíčová slova pomocí Booleovských operátorů, jako jsou AND, OR a NOT, k definování složitosti a přesnosti vyhledávání. Tato metoda se opírá o Booleovu algebru, matematický systém zavedený v 19. století Georgem Boolem, který poskytuje základ pro manipulaci s logickými výrazy. Každému dokumentu v databázi je přirazena hodnota pravdivosti na základě toho, zda splňuje logické podmínky

definované ve vyhledávacím dotazu. Například použití operátoru AND mezi dvěma klíčovými slovy vrátí dokumenty, které obsahují obě klíčová slova, zatímco použití OR vrátí dokumenty obsahující alespoň jedno z klíčových slov. Operátor NOT vyloučí z výsledků dokumenty, které obsahují určité klíčové slovo. Tato flexibilita umožňuje uživatelům konstruovat vysoce specifické dotazy, což je zvláště užitečné v oblastech s velkým množstvím dat, jako jsou právní databáze a akademické databáze [3].

- **`Fuzzy vyhledávání`** je technika, která zvládne nalézt shody, jež nejsou přesně identické s hledaným výrazem. Tato metoda se opírá o algoritmy pro měření vzdálenosti nebo podobnosti mezi řetězci, což umožňuje identifikovat a vrátit výsledky, které jsou podobné, ale ne nutně identické s hledaným dotazem. Nejčastěji používaným algoritmem pro fuzzy vyhledávání je *Levenshteinova vzdálenost*, která počítá minimální počet jednoznakových operací (vložení, smazání, nahrazení), které jsou potřebné k převedení jednoho řetězce na druhý. Tato flexibilita ve vyhledávání je zvláště užitečná v případech, kdy uživatelé udělají překlep při zadávání dotazu, nebo když vyhledávají informace v databázi s nekonzistentními daty. Fuzzy vyhledávání tak zvyšuje pravděpodobnost nalezení relevantních informací tím, že toleruje malé rozdíly mezi hledaným výrazem a potenciálními výsledky [4, 5].

- **`Proximity vyhledávání`** umožňuje uživatelům najít dokumenty, obecně záznamy, kde se dva nebo více vyhledávaných termů objevuje blízko sebe v textu. Nejprve se identifikují dokumenty, které obsahují všechny hledané termíny. Poté se metoda zaměřuje na umístění těchto termů v textu a hodnotí, jak blízko se nacházejí vzhledem k sobě. Blízkost termů může být definována různě – například mohou být ve stejné větě, odstavci nebo v rámci určitého počtu slov od sebe. Například výsledkem dotazu „jablko NEAR/5 koláč“ bude pouze takový text, kde se oba hledané termy vyskytují maximálně pět slov od sebe. Tato metoda je obzvláště užitečná v situacích, kdy samotná přítomnost vyhledávaných klíčových slov v dokumentu není dostatečná pro určení jeho relevance [6, 7].

- **`Fulltextové vyhledávání`** představuje sofistikovaný přístup k vyhledávání informací, který umožňuje prohledávat kompletní texty dokumentů na rozdíl od omezenějších přístupů, jako je vyhledávání založené čistě na klíčových slovech. Tato metoda, zásadní pro funkčnost moderních vyhledávačů a databázových systémů, indexuje každé slovo obsažené v dokumentech, čímž usnadňuje rychlé a efektivní vyhledávání podle libovolných slov nebo frází obsažených v textu. Mechanika fulltextového vyhledávání zahrnuje několik klíčových kroků, začínajících procesem indexace, během kterého je text dokumentů analyzován a rozdělen na jednotlivá slova (tokeny). Tyto tokeny jsou poté normalizovány (např. převedeny na malá písmena, odstranění diakritiky) a mohou být obohaceny o další informace, jako jsou kořeny slov (stem-

ming) nebo lexikální varianty (lemmatizace). Nakonec jsou tokeny přidány do invertovaného indexu – datová struktura umožňující rychlé vyhledávání slov a jejich výskytů v dokumentech. V rámci nástrojů implementujících fulltextové vyhledávání, např. Elasticsearch, se často využívá ať už zmíněných technik (booleovské, fuzzy, proximity) nebo například použití praktik regulárních výrazů, wildcards a dalších [8, 9, 10].

Vyhledávání na základě klíčových slov si klade jako pozitivum rychlost celého procesu vyhledávání. Nicméně nedostačuje v rámci pochopení záměru uživatele, protože nebere v potaz slovní kotext [1].

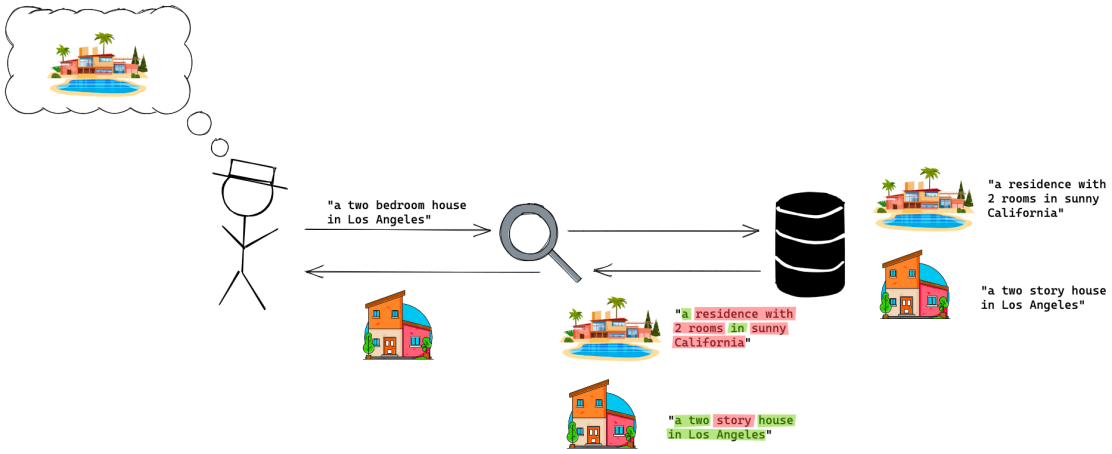
1.2 Sémantické vyhledávání

Sémantické vyhledávání se vyznačuje snahou o rozpoznání a pochopení významu uživatelských dotazů a dat, na kterých se vyhledávání provádí, namísto pouhého hledání shod s klíčovými slovy. Základním principem je pojetí, že vyhledávací systémy by měly disponovat schopností interpretace dotazů v širším kontextu a nabízet výsledky, jež jsou nejen relevantní na základě textové shody, ale také smysluplné vzhledem k záměru uživatele a sémantickému obsahu dat. To může zahrnovat porozumění synonymům, kontextu, vztahům mezi entitami a abstraktním konceptům v dotazu či zdrojových datech [11].

V článku [11] se sémantické vyhledávání popisuje jako „vyhledávání s významem“, což podtrhuje důležitost interpretace a porozumění jak dotazu, tak datům při vyhledávání. Autoři zdůrazňují, že sémantické vyhledávání vyžaduje překročení tradičních metod založených na klíčových slovech, aby bylo možné adekvátně reagovat na složitější informační potřeby uživatelů. Zahrnuje to techniky z oblasti zpracování přirozeného jazyka (NLP), ontologií a inferenčních mechanismů.

Sémantické vyhledávání přináší významný posun od tradičního způsobu, jakým vyhledávací systémy interpretují a zpracovávají uživatelské dotazy. Jak naznačuje příklad z realitního sektoru, kde uživatel hledá specifický dům s použitím dotazu „A two bedroom house in Los Angeles“, avšak relevantní článek popisuje tuto nemovitost jako „A residence with 2 rooms in sunny California“. Sémantické vyhledávání ukazuje svou schopnost rozpoznat a spojit různé způsoby vyjádření stejného konceptu [12].

Tradiční, nesémantické vyhledávače by v takovém případě neposkytly očekávaný výsledek, jelikož by nebyla nalezena přímá shoda mezi klíčovými slovy dotazu a popisem nemovitosti v databázi článků [12].

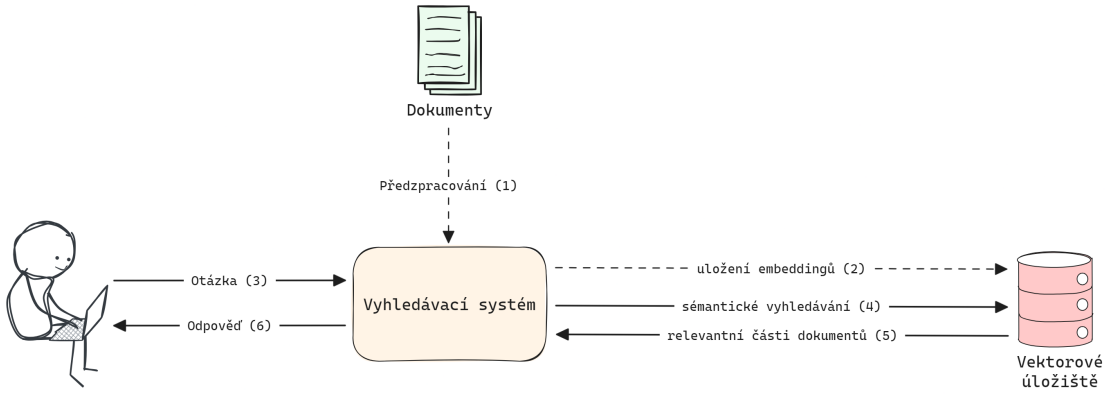


Obrázek 1.1 Workflow dotazu lexikálního vyhledávání [12]

Naopak, sémantické vyhledávače, které se soustředí na význam a kontext slov, by byly schopny identifikovat, že „residence“ a „house“, stejně jako „Los Angeles“ a „California“, jsou úzce související pojmy, a tím uživateli předložit požadovaný článek [12].

Obrázek 1.2 Workflow dotazu sémantického vyhledávání [12]

Další klíčovou vlastností sémantického vyhledávání je jeho schopnost zohledňovat kontext dotazu, což mu umožňuje poskytovat přesnější a relevantnější výsledky. To se ukazuje jako obzvláště užitečné v případech, kdy je dotaz mnohoznačný. Například slovo „Python“ může být spojeno jak s druhem hada, tak s programovacím jazykem. Tradiční vyhledávače mohou mít problém rozlišit, který význam je v daném kontextu relevantní, ale sémantické vyhledávače jsou navrženy tak, aby interpretovaly záměr uživatele. Pokud uživatel zadá dotaz „learn Python“, sémantický vyhledávač dokáže rozpoznat, že uživatel pravděpodobně hledá informace o programovacím jazyku a nikoli o hadím druhu. Tímto způsobem sémantické vyhledávání dokáže efektivně zúžit oblast vyhledávání a předložit uživateli relevantní výsledky, které co nejpřesněji odpovídají jeho informační potřebě [12].



Obrázek 1.3 Příklad workflow sémantického vyhledávače

Obrázek 1.3 demonstruje myšlenku workflow sémantického vyhledávání. Data se před vyhledáváním nejprve předzpracují a následně pomocí vybraného modelu převedou na vektorovou reprezentaci. Vytvořené vektory se uloží do vektorového úložiště. Za účelem vyhledávání uživatel pošle přes rozhraní sémantického vyhledávače dotaz, který se také převede na vektorouvou reprezentaci, avšak v poslední fázi dojde k vyhodnocení nejvíce relevantních informací za pomocí porovnávacího algoritmu. Výsledná odpověď je prezentována uživateli [13].

V následujících podsekcích budou jednotlivé fáze hlouběji analyzovány.

1.2.1 Předzpracování textových dat

Předzpracování textových dat je nedílnou součástí každého přístupu zaměřeného na extrakci užitečných informací z rozsáhlých textových dokumentů. Správně provedené předzpracování je nezbytné pro zajištění, že následné modely strojového učení budou schopny textová data správně interpretovat a využít. V této části práce budou představeny oblasti předzpracování textu, které jsou nejvíce využívány pro sémantické vyhledávání, včetně chunkování, tokenizace a specifických metod subword tokenizace, jako je algoritmus WordPiece.

1.2.1.1 Chunkování

Chunkování je technika používaná v NLP, která se zaměřuje na rozdělení textu do menších, smysluplných jednotek – „chunků“. Tyto jednotky mohou být slova, fráze nebo celé věty, které jsou nějakým způsobem související. Hlavním cílem chunkování je zjednodušit analýzu textu a usnadnit práci s rozshálými dokumenty pro modely sémantického vyhledávání. Při předzpracování textu pomocí chunkování se využívají následující metody [14, 15, 16].

- **`Tokenové chunkování`** rozděluje text na menší části na základě předem stanoveného počtu tokenů. Tento přístup umožňuje volitelně zahrnout překryv mezi jednotlivými chunky. Obecně je žádoucí udržet určité překrytí mezi chunky, aby se zajistilo, že se sémantický kontext neztratí. Metoda je výpočetně nenáročná a nevyžaduje pokročilé NLP knihovny.
- **`Sémantické chunkování`** rozděluje text na chunky na základě sémantických vztahů, což znamená, že seskupuje věty s podobným významem [14, 15, 16].
- **`Rekurzivní chunkování`** rozděluje text na menší chunky rekurzivně, dokud není dosaženo určité podmínky, například minimální velikosti chunku [14, 15, 16].
- **`Dokumentové chunkování`** vytváří segmenty, které se přizpůsobují logickým sekcím dokumentu, jako jsou odstavce, podsekce nebo tagy u značkovacích jazyků (HTML, XML), a proto je metoda vhodnější pro strukturované textové dokumenty [14, 15, 16].

1.2.1.2 Tokenizace

Tokenizace je klíčovým procesem v NLP, který převádí text na jednotky zvané **tokeny**, které následně ukládá do slovníku. Slovník slouží jako referenční databáze pro transformaci textu na tokeny a zpět. Existují tři hlavní typy tokenizace. Tokenizace založená na slovech (word), znacích (character) a podřetězcích (subword), přičemž každý z přístupů má svoje využití [17, 18].

- **`Word tokenizace`** rozděluje text na jednotlivá slova pomocí oddělovačů (delimiterů), nejčastěji mezer. Tento přístup je jednoduchý a přímočaře zachovává sémantický význam slov. Například věta „Mám rád kávu.“ bude tokenizována jako [„Mám“, „rád“, „kávu“]. Nicméně tato metoda nedokáže dobře pracovat s neznámými slovy a s variabilitou slovních tvarů, což může vést k velmi rozsáhlému slovníku a neefektivnosti při zpracování textu [17, 18].
- **`Character tokenizace`** rozděluje text na jednotlivé znaky. Tento přístup je velmi flexibilní, protože dokáže pracovat s jakýmkoliv textem bez ohledu na neznámá slova. Výhodou je malá velikost slovníku, který zahrnuje pouze znaky a několik speciálních symbolů. Na druhou stranu, tato metoda může vést k delším sekvencím a ztrátě sémantického významu, protože každý znak je zpracováván samostatně [17, 18].
- **`Subword tokenizace`** kombinuje výhody obou předchozích přístupů. Subword tokenizace rozkládá slova na menší, sémanticky významné jednotky zvané podřetězce. Tento přístup je účinný při zvládání neznámých slov a snižuje velikost slovníku. Mezi nejznámější algoritmy patří *Byte Pair Encoding, Unigram Language Model* a *WordPiece* [19, 17, 18].

WordPiece

WordPiece algoritmus je specifický druh subword tokenizace. Algoritmus začíná rozdělením textu na jednotlivé znaky a následně iterativně spojuje nejčastější dvojice znaků nebo podřetězců do nových podřetězců, dokud není dosaženo požadované velikosti slovníku. Tento proces umožňuje efektivní zpracování složených slov a redukci počtu neznámých slov, což vede k lepšímu výkonu modelů na reálných datech. WordPiece algoritmus je široce používaná technika v moderních NLP modelech díky své schopnosti zlepšit přesnost a výkonnost při zpracování textu [19].

1.2.2 Tvorba embeddingů

Každý vyhledávací systém stojí na určitém principu, který vyhledávací činnost umožňuje. Jádrem sémantického vyhledávání je fáze převodu textu na tzv. „embeddingy“. V této sekci bude vysvětleno, co to embeddingy jsou, jaké predikční modely pro jejich tvorbu existují a čím se liší.

Vnoření slov

Vnoření slov (word embeddings), neboli slovní vektory, představuje klíčovou technologii v oblasti NLP, která umožňuje počítačovým modelům zachytávat a interpretovat sémantický význam slov a frází. Tyto reprezentace transformují slova do vysokodimenzionálního vektorového prostoru, kde každé slovo je reprezentováno vektorem skutečných čísel. Důležitost embeddingů spočívá v jejich schopnosti zachytit kontext a sémantické vztahy mezi slovy, což umožňuje strojům lépe porozumět lidskému jazyku [20, 21, 22].

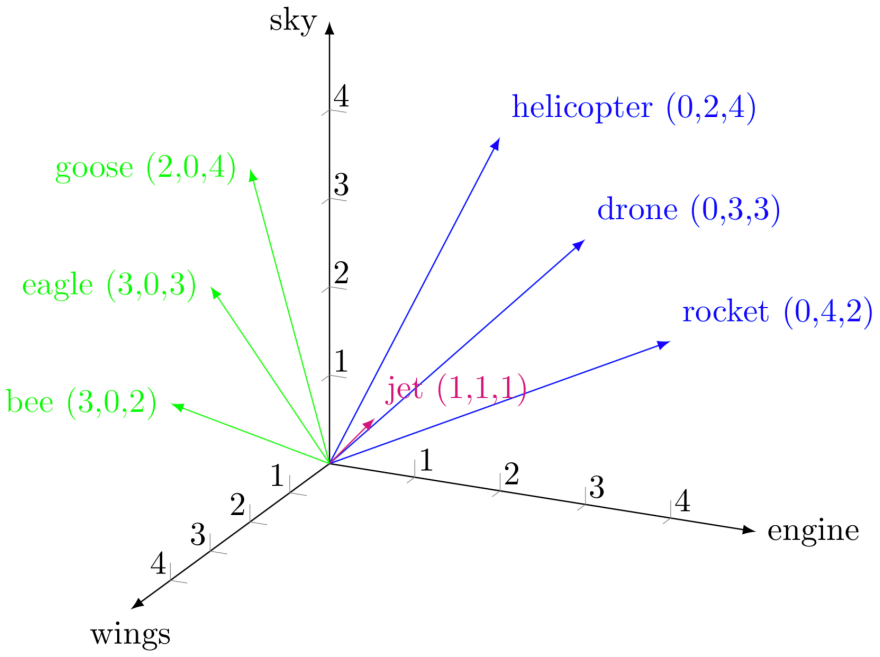
Ve světě sémantického vyhledávání mají slovní vektory zásadní roli. Pomocí nich mohou vyhledávače překonat tradiční omezení klíčových slov a místo toho interpretovat dotazy uživatelů s hlubším porozuměním jejich sémantického významu. V praxi to znamená, že vyhledávače mohou lépe rozpoznat, co uživatelé skutečně hledají. Poskytují relevantnější a přesnější výsledky tím, že analyzují sémantickou podobnost mezi slovy a frázemi v dotazech, a obsahu webových stránek [23].

Před vývojem dnešních embeddingů založených na **predikci** se NLP spoléhalo na jednodušší formy reprezentace slov, jako jsou one-hot kódování, TF-IDF vektorizaci a frekvenční matice. Tyto metody nedokázaly účinně zachytit kontext ani sémantické vztahy mezi slovy. To vedlo k neschopnosti modelů efektivně zpracovávat přirozený jazyk, protože většina lidského jazyka závisí právě na kontextu a významu, který nemohl být prostřednictvím těchto **frekvenčních** metod dosažen. Například, tradiční přístupy nedokázaly rozlišit, že slova jako „král“ a „královna“ mají podobný kontext, ale rozdílný pohlavní význam. Dnešní embeddingy byly vyvinuty

jako řešení těchto problémů. Poskytují způsob, jakým mohou být slova transformována do numerických vektorů, které účinně zachycují sémantický význam a kontext slov [22, 21, 24].

Embeddingy fungují na principu reprezentace slov jako bodů v prostoru s vysokou dimenzí, kde vzdálenosti a směry mezi vektory odrážejí sémantické vztahy. Například slova s podobnými významy jsou umístěna blízko sebe, zatímco slova s různými významy jsou dále od sebe. Tato vlastnost umožňuje modelům provádět aritmetické operace s vektory, jako je nalezení analogií tím, že se vektorové rozdíly a součty používají k odvození vztahů mezi slovy [20].

Vezměme si pro znázornění menší korpus se sedmi slovy: *bee, eagle, goose, helicopter, drone, rocket, jet* ve 3-dimenzionálním vektorovém prostoru, kde každá dimenze představuje jeden kontext: *wings, engine, sky*. Každé slovo je charakterizováno třemi složkami, které korespondují s četností výskytu určitého slova v daném kontextu. Z obrázku 1.4 je patrné, že *helicopter* není nalezeno v kontextu *wings*, ale vyskytuje se dvakrát v kontextu *engine* a čtyřikrát v kontextu *sky*. Výsledný vektor slova *helicopter* je reprezentován $\vec{v} = (0, 2, 4)$ [25].



Obrázek 1.4 Reprezentace sedmi slov ve 3-dimenzionálním vektorovém prostoru [25]

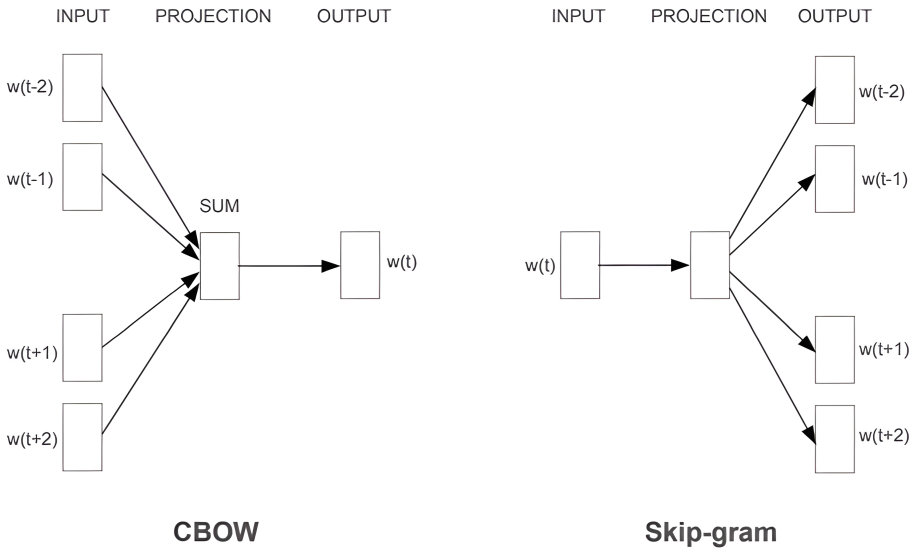
Předpokladem, který stojí za vektorovými reprezentacemi slov, je založení sémantické podobnosti na kontextové blízkosti. Například *helicopter* a *dron* jsou si blízko, protože se objevují v podobných kontextech, mají podobné vektorové profily a tudíž jsou ve vektorovém prostoru blízko u sebe [25].

Pro převádění slov na vektory byly vyvinuty modely pomocí praktik strojového učení, které využívají technoloie založené na konceptu neuronových sítí. Tyto modely je možné natrénovat s použitím vhodného datasetu a předpřipravit tak vektorový prostor pro potřeby sémantického vyhledávání[26].

V následujících podsekcích budou představeny modely od jednodušších po ty komplexnější.

1.2.2.1 Word2Vec

Word2Vec, představen Mikolovem a kol. (2013), je metoda NLP, která mapuje slova do vektorů ve vysokodimenzionálním prostoru, což umožňuje zachytit širokou škálu přesných syntaktických a sémantických vztahů mezi slovy. Word2Vec je uskupení dvou architektur **CBOW** a **skip-gram**, které jsou postaveny na technologii dvouvrstvé neuronové sítě se skrytou vrstvou navíc [20].



Obrázek 1.5 Modely Word2Vec architektur [20]

Pro představení jednotlivých architektur je třeba vysvětlit způsob **one-hot kódování**. Jedná se o základní metodu převodu slov na vektory. Velikost tohoto vektoru odpovídá počtu slov ve slovníku a každému slovu je přiřazen unikátní vektor. Pozice slova, které se reprezentuje samo sebou, je kódována jako 1, zatímco všechny ostatní pozice jsou kódovány jako 0 [27].

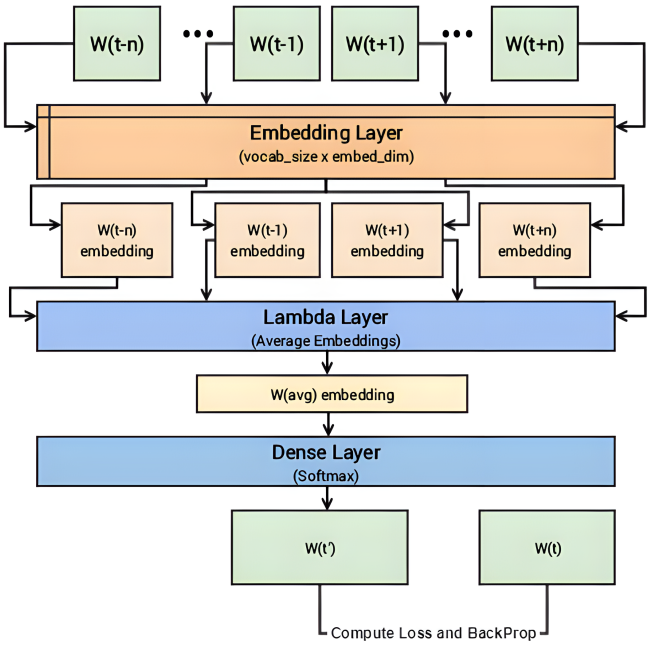
CBOW

Continuous Bag of Words (CBOW) model predikuje cílové tzv. „středové“ slovo na základě kontextu. Jednoduše řečeno se model snaží odhadnout slovo vzhledem k jeho okolí. Tento přístup

efektivně využívá informace o kontextu slova, což umožňuje rychlé a přesné trénování embeddingů, zejména pro frekventovaná slova. Díky rychlejšímu tréninku je tak CBOW vhodný pro práci s velkými datovými sadami [20, 28].

Rozdělení CBOW vrstev popisující zobrazení 1.6:

- **`Vstupní (embedding) vrstva`** přijímá kontextová slova v podobě one-hot kódování. Pro trénování neurovnové síťe slouží one-hot zakódovaná slova jakožto indexy do matice vah, které jsou na začátku procesu inicializovány náhodně.
- **`Skrytá (lambda) vrstva`** přijímá vektory všech kontextových slov. Dále vypočítá jejich aritmetický průměr, čímž vytvoří jeden „dense vektor“, který je reprezentací celého kontextu. Tento průměrový vektor snižuje dimenzionalitu problému a zároveň zachovává kontextové informace potřebné pro predikci.
- **`Výstupní (dense) vrstva`** přijímá průměrový vektor z lambda vrstvy a používá ho k predikci nejpravděpodobnějšího cílového slova. V této vrstvě se aplikuje klasifikátor softmax, který má schopnost převést vektor na distribuci pravděpodobností přes celý slovník. Funkce softmax tedy vyhodnocuje pravděpodobnost, s jakou může každé slovo ze slovníku být cílovým slovem vzhledem k danému kontextu.



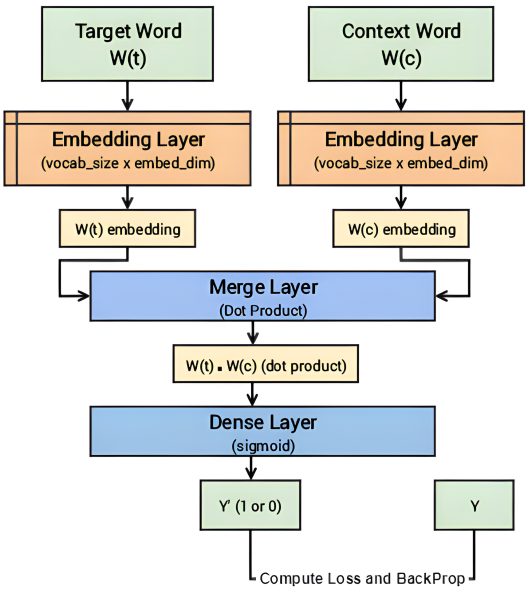
Obrázek 1.6 Vizuální zobrazení CBOW architektury [28]

Na základě distribuce pravděpodobností je vybráno slovo s nejvyšší pravděpodobností, které se považuje za výstup modelu. Během tréninku se vyhodnocuje chyba mezi predikovaným výstupem a skutečným cílovým slovem. Pomocí algoritmu zpětné propagace (backpropagation) se aktualizují váhy mezi vrstvami, aby model lépe predikoval v budoucnosti [20, 28].

Skip-gram

Skip-gram model funguje na opačném principu než zmíněný model CBOW. Na základě předloženého cílového (středového) slova predikuje slova kontextová. Tento přístup byl vyhodnocen jako vhodnější pro práci s rozmanitějším kontextem, a především se slovy, které se vyskytují méně často. Vzhledem k náročnějšímu trénování, vlivem častější aktualizace vah a nutnosti vykonávat více průchodů přes data, je skip-gram vhodnější spíše pro měnší dataset [20].

Do samotného modelu vstupuje dvojice (`X`, `Y`), kde `X` představuje další dvojici složenou z cílového a kotextového slova. `Y` poté slouží jako pozitivní (resp. negativní) číselný ukazatel (1 nebo 0) toho, zdali se jako kontextové zvolilo skutečné (resp. náhodně vybrané) slovo [20, 29].



Obrázek 1.7 Vizuální zobrarzneí skip-gram architektury [29]

Rozdělení skip-gram vrstev popisující zobrazení 1.7:

- **`Vstupní (embedding) vrstva`** zde funguje separátně jak pro cílové, tak pro kontextové slovo. Slovo je obdobně jako v CBOW modelu přijímáno v podobě one-hot kódování a následně převedeno na embedding (dense vector) pomocí váhové matice.

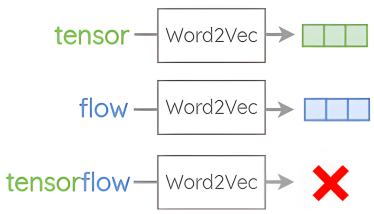
- **`Skrytá (merge) vrstva`** přijímá cílový i kontextový embedding a vypočítá jejich skalární součin (dot product). Výsledná hodnota odpovídá míře asociace mezi cílovým a kontextovým slovem.
- **`Výstupní (dense) vrstva`** zpracuje číslo z merge vrstvy v sigmoidní aktivační funkci, která transformuje skalární součin na pravděpodobnostní skóre mezi 0 a 1. Výstupem této vrstvy je výsledek transformace Y' ozančující pravděpodobnostní míru, jak moc je vstupní cílové a kontextové slovo součástí stejného kontextu.

Model se učí na základě srovnání predikovaného výstupu Y' s původním Y, a podle chyby upravuje váhy v embeddingové vrstvě pomocí procesu zpětné propagace (backpropagation). Cílem je dosáhnout takového nastavení váhové matice, kde embeddingy přiřazené slovům dobře reprezentují jejich významy a kontextové vztahy [20, 29].

1.2.2.2 FastText

FastText je open-source rozšíření architektury skip-gram populárního modelu Word2Vec pro generování vektorových reprezentací slov (embeddingů), které bylo vyvinuto výzkumným týmem Facebook AI Research (FAIR) s cílem zlepšit zpracování a analýzu přirozeného jazyka. Na rozdíl od svých předchůdců, Word2Vec a GloVe, které se zaměřují výhradně na generování embeddingů pro celá slova, FastText přináší inovativní koncept zahrnutí informací o tzv. „subword“ jednotky do svých embeddingů. Tento přístup umožňuje modelu FastText zachytit morfologii slov, čímž se stává schopnějším zvládat jazyky s bohatými slovními formami a lépe se vypořádat se slovy, které nebyly součástí trénovacích dat, a tudíž se nevyskytují ve slovníku modelu. Jedná se o tzv. „out-of-vocabulary words“ (OOV). Hlavním cílem FastText přístupu je zlepšit kvalitu textové reprezentace s důrazem na rychlost a efektivitu, což usnadňuje trénování modelů na velkých datech v rozumném časovém rámci [30, 31, 32, 33].

Na následujícím obrázku z článku [31] je znázorněn nedostatek modelu Word2Vec pro slovo „tensorflow“.

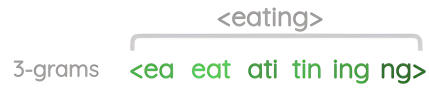


Obrázek 1.8 Word2Vec OOV nedostatek

Slova „tensor“ a „flow“ se nacházejí ve slovníku modelu Word2Vec, avšak generování embeddingu slova „tensorflow“ selže.

Pro každé slovo jsou generovány tzv. „n-gramy“ o délce od 3 do 6 znaků, které zachycují morfologický rozklad slova a případné jeho varianty (např. skloňování).

Nechť je pro demonstraci využit příklad z článku [31] na anglickém slově **eating**, kde `n = 3`. Pro toto slovo jsou generovány znakové n-gramy v podobě viz 1.9a



(a) Rozdělení slova „eating“ na 3-gramové části

slovo	n	n-gramy
eating	3	⟨ea, eat, at, ti, tin, ing, ng⟩
eating	4	⟨eat, eati, atin, ting, ing⟩
eating	5	⟨eati, eatin, ating, ting⟩
eating	6	⟨eatin, eating, ating⟩

(b) Zobrazení n-gramů délky 3-6 anlgického slova „eating“ [31]

Vypočtené n-gramy jsou reprezentovány jako embeddingy, kde každý n-gram má přiřazen svůj vlastní vektor. Výsledná vektorová reprezentace slova **eating** je dána sumací vektorů všech těchto n-gramů a vektoru reprezentujícího celé slovo. Vzniklý embedding je dále propagován do již známe architektury skip-gram [31].

Tímto způsobem je v modelu FastText zachycena nejen sémantická hodnota celého slova, ale zohledňovány jsou také vlastnosti jednotlivých morfémů a segmentů, z nichž se slovo skládá. Tento mechanismus umožňuje FastText modelu lépe zpracovávat slova s komplexní morfologií a poskytuje nástroj pro generování vektorů pro slova, která se v trénovacím datasetu nevyskytují, což je významná výhoda oproti modelům, které pracují pouze s celými slovy [30, 31].

Jazyk	skip-gram (word2vec)	CBOW (word2vec)	skip-gram (FastText)
Cs	52.8	55.0	77.8
De	44.5	45.0	56.4
En	70.1	69.9	74.9
It	51.5	51.8	62.7

Tabulka 1.1 Porovnání přesnosti syntaktických analogií mezi Word2Vec a FastText modely

Z tabulky 1.1 lze vyčíst, že oproti Word2Vec modelu FastText poskytuje větší přesnost z hlediska syntaktických úloh u morfologicky komplexnějších jazyků, jako je český nebo německý [30].

1.2.2.3 BERT

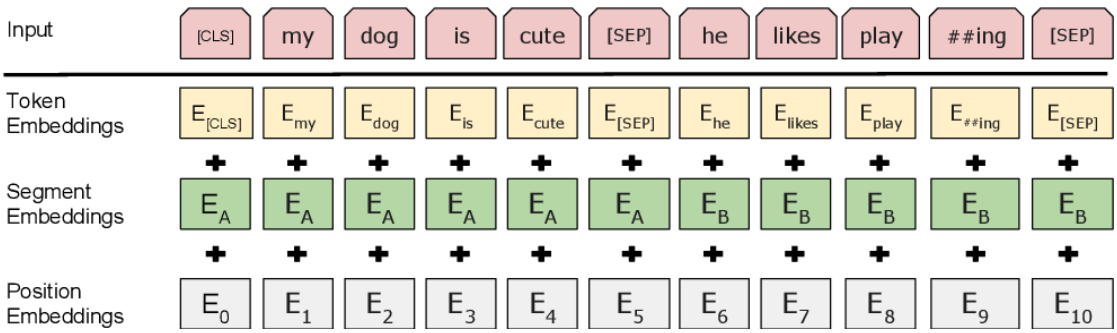
BERT (Bidirectional Encoder Representations from Transformers) je průlomový model v oblasti zpracování přirozeného jazyka, který byl představen v roce 2018 výzkumníky ze společnosti Google. Jeho základem je architektura transformátoru, která umožňuje modelu efektivně zpracovávat široké spektrum jazykových dat. Klíčovou inovací BERT je jeho schopnost trénovat obousměrné reprezentace slov, tj. každé slovo je reprezentováno v kontextu všech ostatních slov ve větě, nikoliv jen slov předcházejících či následujících, jak bylo obvyklé u předchozích modelů [34].

Existují dvě hlavní konfigurace modelu BERT, které stojí na technologii tzv. „encoder“ v architektuře transformátorů. Liší se obecně velikostí a kapacitou.

1. **BERT Base:** 12 encoderových bloků, 12 attention hlav a 110 milionů parametrů
2. **BERT Large:** 24 encoderových bloků, 16 attention hlav a 340 milionů parametrů

Vstupní reprezentace BERT modelu se skládá z jedné nebo dvou vět, které jsou rozděleny WordPiece algoritmem na jednotlivé tokeny (termy). Než však dojde k přijetí těchto tokenů, tak jsou vměstnány speciální signalizační tokeny mezi dané věty. Konfigurace, jakým způsobem a jakým algoritmem se bude tokenizovat, je obvykle uvedena v přidruženém souboru [34, 35].

- **[CLS]** představuje klasifikační token, který je přidán do každé vstupní sekvence slov.
- **[SEP]** představuje separační token, který odděluje jednotlivé konce vět vstupujících do modelu.



Obrázek 1.10 Tvorba vstupních embeddingů modelu BERT [34]

Následně se vstupní embedding sestaví jako kombinace výstpních **tokenových**, **segmentových** a **pozičních** embeddingů z embeddingové vrstvy, jak vyplývá z obrázku 1.10.

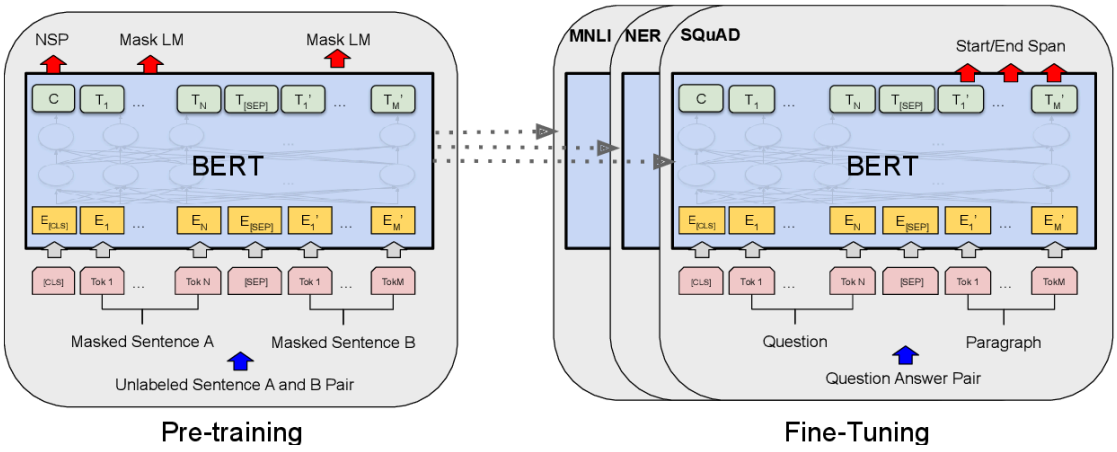
- `Tokenový embedding` vznikne převedením tokenu pomocí slovníkové tabulky (inicializované při WordPiece předzpracování).

- **`Segmentový embedding`** reprezentuje, k jaké větě se token vztahuje.
- **`Poziční embedding`** kóduje informaci o absolutní nebo relativní pozici tokenu ve větě.

Předtrénování

Zásadní pro úspěch BERT modelu je technika předtrénování využívající dvě hlavní úlohy:

- **`Masked Language Model (MLM)`** slouží k predikci náhodně zamaskovaných tokenů ve vstupních sekvencích. V průběhu tohoto procesu je přibližně 15 % tokenů v každé vstupní sekvenci zaměněno za speciální token [**MASK**], přičemž model je následně vyzván, aby na základě kontextu poskytnutého zbývajícími, nezamaskovanými tokeny, tyto chybějící tokeny predikoval. Klíčovým aspektem MLM je schopnost efektivně využívat kontext z obou stran zamaskovaného tokenu, což vede k preciznějšímu a bohatšímu učení jazykových reprezentací, neboť model není omezen pouze na informace přicházející z jednoho směru [34].
- **`Next Sentence Prediction (NSP)`** spočívá v analýze a predikci, zda je jedna věta logickým pokračováním druhé. Během trénování model obdrží dvojice vět, přičemž některé z nich jsou skutečně spojené v logickém sledu, zatímco jiné jsou kombinovány náhodně. Cílem je naučit model identifikovat, zda druhá věta ve vstupní sekvenci přirozeně následuje po první, což vyžaduje od modelu hluboké porozumění kontextu a vztahů mezi větami. Model produkuje binární výstup, který indikuje jeho domněnku, že věty k sobě logicky patří [34].



Obrázek 1.11 Architektura fáze předtrénování a následného ladění [34]

Ladění

Ladění neboli „fine-tuning“ modelu BERT spočívá v adaptaci předtrénovaného modelu na konkrétní úlohy NLP. Tento proces začíná integrací úlohově specifických výstupních vrstev do modelu, které jsou navrženy tak, aby odpovídaly konkrétním potřebám, jako je klasifikace textu,

rozpoznávání entit, či „feature extraction“. Model je poté trénován na menším, ale specificky **„labelovaném“** datasetu, přičemž se upravují a ladí všechny parametry modelu, včetně vah v úlohově specifických i původních vrstvách, aby se dosáhlo co nejlepšího výkonu na dané úloze. Rychlost učení během této fáze je obvykle nižší, aby se předešlo přepisování komplexních jazykových reprezentací získaných během předtrénování. Ladění tedy transformuje obecný model BERT na vysoce specializovaný nástroj pro příslušnou úlohu, čímž se značně zvyšuje jeho přesnost a efektivita při minimalizaci potřeby rozsáhlých trénovacích dat specifických pro úlohu [34, 36, 37].

RoBERTa

RoBERTa (Robustly Optimized BERT Pretraining Approach) představuje variaci BERT modelu. Autoři provedli rozsáhlé testy různých hyperparametrů a objemu trénovacích dat. Zjistili, že BERT byl významně podtrénován. Představili model RoBERTa, který vylepšuje BERT tím, že trénuje déle, na větších datech a s vyššími dávkami. Tento model dosáhl lepších výsledků na testovacích úlohách GLUE, RACE a SQuAD, což ukazuje na význam některých dříve přehlížených designových rozhodnutí [38].

- `intfloat/multilingual-e5-base` je příkladem multilinguální nadstavby RoBERTa *xlm-roberta-base*, která byla trénovaná na velkých datasetech (mC4, NLLB, Wikipedia …). Model se skládá z dvanácti vrstev, tvoří embeddingy s dimenzí o velikosti 768 a byl dále laděn na menších datasetech. Podporuje až 100 světových jazyků včetně češtiny. V kontextu MIRACL, multilinguálního vyhledávacího benchmarku, který pokrývá 16 jazyků, model *multilingual-e5-base* dosáhl průměrné hodnoty *nDCG@10* 62,3 a *Recall@100* 93,1. Tyto výsledky potvrzují vysokou účinnost modelu ve vyhledávání a extrakci informací v různých jazycích [39].

DistilBERT

DistilBERT je zjednodušená a komprimovaná verze modelu BERT, navržená k efektivnějšímu provozu s menším počtem parametrů a vyšší rychlostí inferenčního zpracování. DistilBERT využívá techniku zvanou „knowledge distillation“, která umožňuje „studentovi“ – v tomto případě modelu DistilBERT – učit se od „učitele“ – původního modelu BERT. Díky této metodě se podařilo snížit velikost modelu o 40 %, zachovat 97 % jeho schopnosti porozumění jazyku a dosáhnout o 60 % rychlejšího inferenčního času ve srovnání s původním BERTem [40].

SBERT

SBERT (Sentence-BERT) efektivně generuje embeddingy vět s využitím siamské a trojčlenné síťové architektury. Tato modifikace umožňuje rychleji a sémanticky přesněji porovnávat věty,

neboť SBERT redukuje čas potřebný k nalezení nejpodobnějších vět z 65 hodin (při použití BERT) na několik sekund [41].

1.2.3 Vektorová úložiště

Vektorová úložiště jsou typem databází navržených pro účinné ukládání a dotazování vysokodimenzionálních vektorů. Na rozdíl od tradičních *Online Transaction Processing (OLTP)* a *Online Analytical Processing (OLAP)* databází, kde jsou data organizována do řádků a sloupců (které se nazývají tabulky) a dotazy se provádějí na základě hodnot v těchto sloupcích. V aplikacích NLP bývají data často reprezentována jako vektory ve vysokodimenzionálním prostoru. Tyto vektory spolu s identifikátorem a dodatečnými daty jsou uloženy ve struktuře, která se nazývá kolekce ve vektorové databázi. Vektorové databáze jsou optimalizovány pro účinné ukládání a dotazování těchto vysokodimenzionálních vektorů a často využívají specializované datové struktury a techniky indexování. V rámci vyhledávání nejpodobnějších vektorů k dotazu úložiště využívají algoritmů hledání nejbližších sousedů [42, 43, 44].

Nejznámějším takovým algoritmem je *K-Nearest-Neighbor (KNN)* [45], který funguje v následujících krocích:

1. **`Výběr metriky:`** Nejdříve se zvolí vhodná metrika pro měření vzdálenosti mezi vektory. Nejvíce se využívá Euklidovská vzdálenost, Manhattanova vzdálenost, Minkowského vzdálenost, nebo kosinová podobnost, mezi dalšími.

2. **`Vyhledání sousedů:`** Pro každý dotazovaný vektor algoritmus vyhledá v datasetu k nejbližších sousedů podle zvolené metriky. To znamená, že se najde k vektorů s nejnižší vzdáleností (nebo nejvyšší podobností) vůči dotazovanému vektoru.

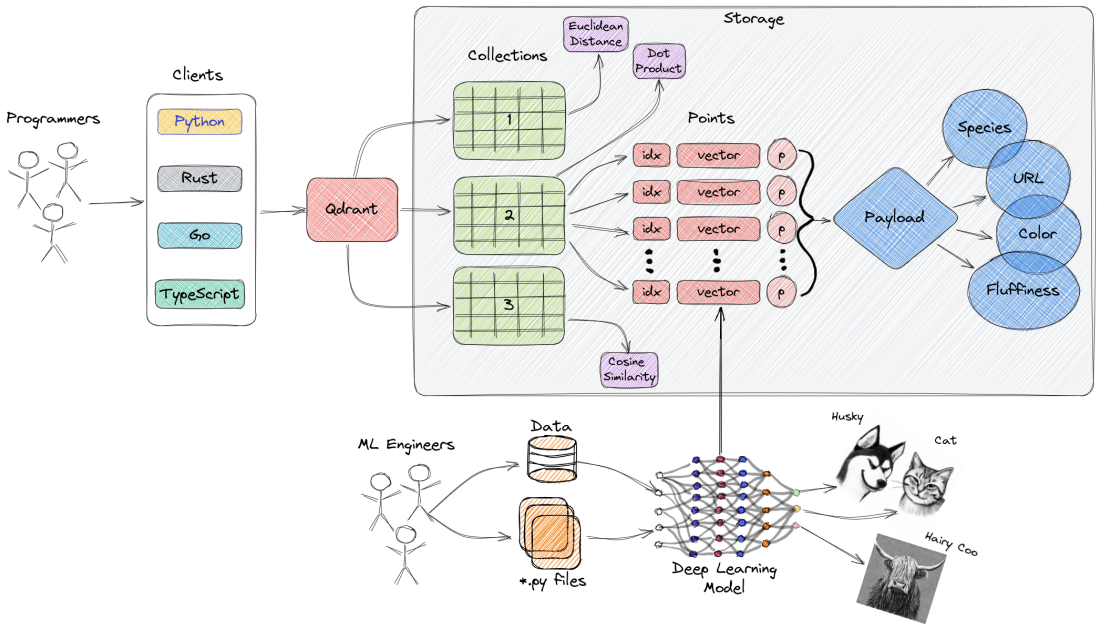
3. **`Výsledek:`** Výsledkem může být průměr sousedních vektorů, jejich majoritní třída v případě klasifikace nebo seznam sousedů pro účely vyhledávání podobnosti.

Algoritmus je uveden pro demonstraci. V praxi jsou však použity spíše náročnější techniky, které se liší především ve vyhledávání sousedů. Jedná se o techniky aproximace nejbližších sousedů (ANN) [46, 42].

Mezi nejuznávanější vektorové databáze patří Milvus, Qdrant, Weaviate, Chroma DB, Vespa a Pinecone. Pro účely této práce bude pozornost věnována detailnímu představení úložiště Qdrant.

1.2.3.1 Qdrant

Qdrant je open-source řešení vektorového úložiště, napsané v programovacím jazyce Rust, které umožňuje uživatelům přizpůsobovat databázi svým specifickým potřebám a zároveň poskytuje komplexní API pro snadnou integraci. Qdrant podporuje různé metody indexace, jako je HNSW (Hierarchical Navigable Small World), která nabízí dobrý kompromis mezi rychlostí a přesností, a techniky jako IVF (Inverted File Indexing) nebo PQ (Product Quantization), které se zaměřují na specifické úlohy pro efektivitu paměti. Pro určení vektorové podobnosti Qdrant podporuje nejznámější metriky jako skalární součin, kosinovou podobnost, Euklidovskou vzdálenost a Manhattanskou vzdálenost. Upřednostňovaná je kosinová podobnost, při vkládání vektoru do databáze se nejprve provede normalizace a při vyhledávání už dojde ke klasickému skalárnímu součinu. Qdrant také využívá binární kvantizaci, což je technika pro redukci celkové velikosti databáze tím, že komprimuje vektory do kompaktnějšího vyjádření za cenu určité ztráty přesnosti. Tato metoda může výrazně zrychlit zpracování dotazů až 40krát. Qdrant je navržen tak, aby zvládal obrovské sady dat obsahující miliardy vysokodimenzionálních vektorů. Škálovatelnost je zajištěna kombinací distribuované architektury, dynamického přidělování zdrojů, dělení dat a optimalizace. Dále nabízí lokální spuštění v Dockeru a pěkný dashboard pro vizualizaci jednotlivých bodů v prostoru. [43, 42]



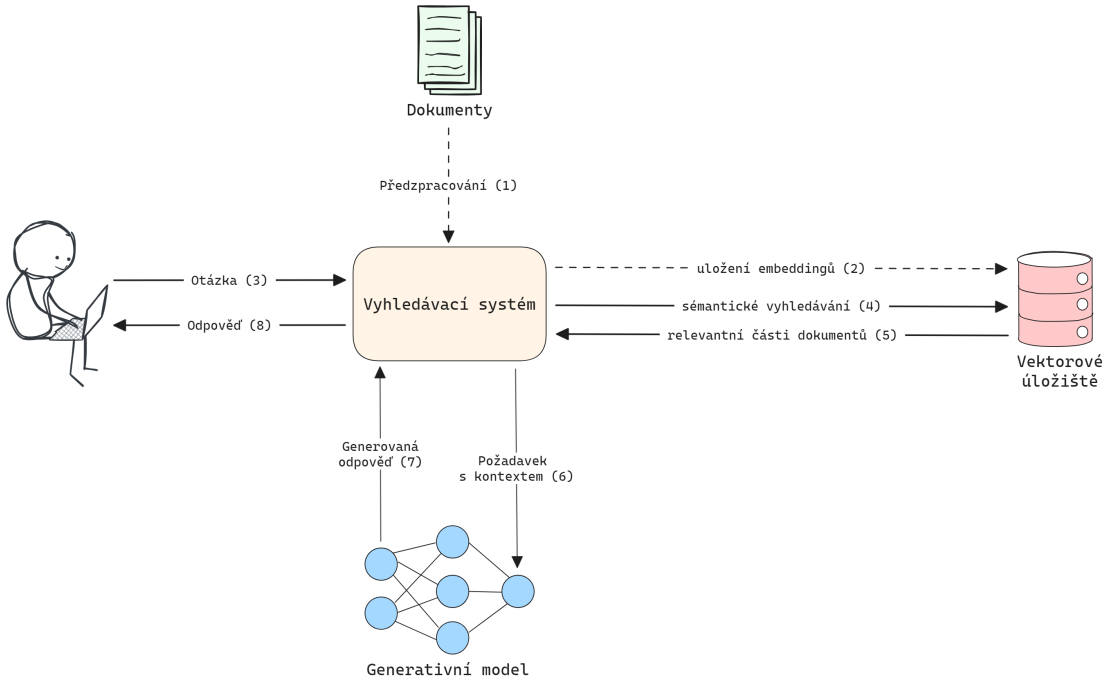
Obrázek 1.12 Architektura Qdrant vektorové databáze [43]

V obrázku 1.12 architektury jsou důležité následující pojmy:

- **`Bod (Point)`** je základní struktura, se kterou Qdrant pracuje. Obsahuje unikátní identifikátor (většinou ve formě UUID), vektor a „payload“, JSON objekt s vektorizovanými daty.
- **`Kolekce (Collection)`** je pojmenovaná množina bodů, mezi kterými lze vyhledávat. Vektor každého bodu ve stejné kolekci musí mít stejnou dimenzi a musí být porovnáván jednou metrikou.

1.3 Retrieval Augmented Generation

Doteď se práce zabývala procesem vyhledávání relevantní části textu z dokumentů na základě dotazu uživatele. Tento proces se v kontextu NLP nazývá *Retreival*. Přestože by bylo možné vyhledanou část textu přímo prezentovat uživateli, často obsahuje informace, které do kontextu sice patří, ale nejsou přímou odpovědí na otázku, což může vést k informačnímu přetížení. V dnešní době, s rozvojem generativních modelů jako je GPT, je možné tento problém vyřešit mnohem efektivněji formou *Augmented Generation*. Generativní model přijme otázku uživatele a vyhledaný kontext s pokyny k jeho zpracování. Na základě obdržených informací model generuje odpověď, která je přesněji cílená na dotaz uživatele a prezentuje ji ve srozumitelné formě [47, 48, 49].



Obrázek 1.13 Workflow procesu Retrieval Augmented Generation

Generativní large language model (LLM) potřebuje pro svůj běh velmi mnoho výpočetních prostředků, proto je téměř nemožné takový model provozovat na běžně dostupném konzumním hardwaru. Pro provoz v prostředí s omezenými zdroji modely využívají techniky zvané **kvantizace**. Kvantizace je způsob, jakým dochází ke kompromisu mezi potřebným výkonem a kvalitou. Snižuje počet bitů použitých k reprezentaci každé váhy modelu, což výrazně snižuje paměťové nároky a urychluje výpočty, přičemž se snaží minimalizovat ztrátu přesnosti. Mezi nejznámější kvantizované modely patří Llama3, Mistral a Phi3. Tyto modely lze nasadit v zařízeních s omezenými výpočetními možnostmi [50, 51].

1.4 Existující řešení

V této rešeršní části práce budou přiblížena existující komerční i open-source řešení, využívající principu sémantického vyhledávání. Budou představeny jejich funkcionality a shrnutím reflektovány jejich nedostatky, které poslouží jako klíčové podněty pro návrh vlastního systému.

1.4.1 Amazon chatbot

Amazon nabízí základní implementaci open-source chatbota, který využívá techniku retrieval augmented generation (RAG) a je napsaný v Pythonu. Tento chatbot je dostupný v rámci GitHub repozitáře [52]. Hlavními komponentami řešení jsou Amazon Bedrock, Amazon Lex, Amazon Kendra a AWS Lambda, které společně zajišťují příjem a analýzu dotazů, vyhledávání relevantních informací a generování odpovědí.

Amazon Lex

Amazon Lex je služba pro vytváření konverzačních rozhraní pomocí hlasu a textu. V tomto řešení slouží k přijímání a analýze dotazů uživatelů. Lex je zodpovědný za zpracování přirozeného jazyka a směrování dotazů k příslušným službám pro další zpracování [53, 52].

Amazon Kendra

Amazon Kendra je vyhledávací služba využívající strojové učení pro relevantní vyhledávání informací z dokumentů. Podporuje širokou škálu datových zdrojů, jako jsou Amazon S3, SharePoint, Google Drive nebo relační databáze. Kendra umožňuje vytváření vlastních modelů pro embeddingy pomocí Amazon SageMaker a integruje se s vektorovými úložišti pro efektivní sémantické vyhledávání. Funkcionality zahrnují rozšířené dotazy, kontextové vyhledávání a automatickou kategorizaci dokumentů [54, 52].

Amazon Bedrock

Amazon Bedrock je služba určená k rychlému nasazení a správě generativních modelů, určených zejména pro aplikace jako chatboty. V tomto řešení generuje odpovědi na základě výsledků vyhledávání od Kendra a informací získaných z dalších zdrojů. Podporuje širokou škálu předtrénovaných modelů, jako jsou GPT-3, Jurassic-2, a T5. Bedrock umožňuje nasazení těchto modelů ve škálovatelné infrastruktuře, přičemž nabízí robustní nástroje pro správu a monitorování výkonu modelů [55, 52].

AWS Lambda

AWS Lambda je serverless výpočetní služba, která umožňuje běh kódu v reakci na události bez správy serverů. V tomto řešení Lambda funguje jako orchestrátor, který propojuje Amazon Lex, Amazon Kendra a Amazon Bedrock. Lambda přijímá dotazy od Lexe, volá Kendru pro vyhledávání informací a následně používá Bedrock pro generování odpovědí [56, 52].

Ačkoliv je řešení chatbota open-source, jednotlivé komponenty jsou zpoplatněné. Amazon také poskytuje AWS SDK pro Javu, čímž usnadňuje integraci do existujících Java aplikací. Mezi další podobná řešení patří Google Cloud Search v kombinaci s Google AI (Vertex AI, Gemini), IBM Watson a Microsoft Azure Cognitive Services v kombinaci Azure OpenAI, které rovněž nabízejí pokročilé vyhledávací a generativní schopnosti pro vytváření nejen inteligentních chatbotů [57, 58, 59, 60].

1.4.2 Verba

Projekt Verba od společnosti Weaviate je platforma, která umožňuje zpracování, indexaci a vyhledávání textových dat skrze interaktivní UI. Verba využívá modely strojového učení pro tvorbu embeddingů od poskytovatelů jako OpenAI, Cohere a HuggingFace. Vektorová databáze Weaviate umožňuje efektivní ukládání a správu těchto embeddingů. Verba zahrnuje několik klíčových komponent, jako jsou *ReaderManager* pro načítání a strukturování dat z různých formátů (.txt, .md, .pdf a další), *ChunkerManager* pro rozdělování dokumentů na menší části, *EmbeddingManager* pro vektorizaci dat, *RetrieveManager* pro vyhledávání relevantních informací a *GenerationManager* pro tvorbu odpovědí pomocí generativních modelů. Verba inovuje zpracování dotazů díky použití sémantické mezipaměti, zajišťující vysokou rychlost. Platforma je napsána v Pythonu a je k dispozici jako open-source, přičemž provozní náklady mohou vzniknout v důsledku používání API klíčů od externích poskytovatelů, jako je OpenAI. Verba také podporuje různé

způsoby nasazení, včetně Dockeru a Weaviate Cloud Service (WCS). Verba nepodporuje nativní integraci pro Javu ani Kotlin a postrádá funkcionalitu na propojení cloudového úložiště. Podobným bezplatným řešením v Pythonu je PrivateGPT [61, 62].

1.4.3 DocGPT

DocGPT je webová aplikace pro analýzu PDF dokumentů, která kombinuje generativní AI technologii s PDF editorem a podporuje multilinguální modely. Freemium verze umožňuje nahrání jednoho dokumentu a pět dotazů měsíčně s použitím modelu GPT-3.5, zatímco premium verze za 4,99 USD měsíčně nabízí neomezený počet otázek a přístup k GPT-4 modelu. Aplikace má uživatelsky přívětivé rozhraní a není open-source, takže ji nelze integrovat. Mezi nevýhody patří omezené funkce freemium verze a absence přímé podpory pro cloudové úložiště [63].

1.4.4 Shrnutí

Žádné z popisovaných řešení nepodporuje bezplatné sémantické vyhledávání s možností integrace v rámci Java nebo Kotlin ekosystému a importování dokumentů z cloudových úložišť. Amazon Chatbot využívá placené komponenty, Verba postrádá nativní podporu pro Javu i funkcionalitu komunikace s cloudovým úložištěm a DocGPT má velmi omezenou až téměř nepoužitelnou freemium verzi bez integrace cloudových služeb, bez podpory integrace a bez možnosti vlastní správy dokumentů.

1.5 Analýza požadavků

V této sekci budou definovány **funkční** a **nefunkční** požadavky na výslednou webovou aplikaci.

1.5.1 Funkční požadavky

Funkční požadavky jsou specifikovány na základě potřeb koncových uživatelů, určují konkrétní chování a funkce, které software musí splňovat, aby jej bylo možné považovat za „funkční“. Tyto požadavky tak popisují akce či interakce se systémem a jejich očekávané výsledné chování [64].

F01 Zadávání dotazu

Webová aplikace musí umožňovat uživatelům zadávání vyhledávacího dotazu do textového pole přes grafické rozhraní.

F02 Vyhledávání informací

Webová aplikace musí na základě poskytnutého dotazu uživatelem umět vyhledat relevantní informace v textových dokumentech v českém jazyce, které jsou uložené v cloudovém úložišti.

F03 Generování odpovědi

Webová aplikace musí z vyhledaných informací prezentovat odpověď uživateli prostřednictvím grafického rozhraní.

F04 Komunikace s backendem

Webová aplikace musí zpřístupňovat komunikaci s backendem skrze REST API.

1.5.2 Nefunkční požadavky

Nefunkční požadavky, jinak známé také jako atributy kvality, určují kritéria pro hodnocení operativní schopnosti systému, a to nikoli na základě jeho konkrétního chování. Tyto požadavky sice neovlivňují přímou funkčnost systému, avšak mají klíčový vliv na uživatelskou zkušenost a celkovou kvalitu jeho provozu. Zahrnují široké spektrum aspektů, jako je výkon, bezpečnost, spolehlivost, možnost rozšíření, škálovatelnost a udržitelnost. Obvykle se definují jako minimální standardy, které musí systém splnit, aby byl považován za úspěšně implementovaný. Nicméně i v případě nesplnění těchto požadavků lze systém považovat za „funkční“ [64].

N01 Technologie

Aplikace může využívat pouze volně dostupné a bezplatné technologe.

N02 Transformace dokumentů

Veškeré dokumenty z cloudového úložiště musí být převedeny na vektorovou reprezentaci, aby bylo umožněné sémantické vyhledávání.

N03 Ukládání dokumentů

Aplikace musí transformované dokumenty ukládat ve vektorovém úložišti.

N04 Jazyk pro backend

Backend webové aplikace musí být napsán v jazyce Java nebo Kotlin.

Kapitola 2

Návrh

Zadavatel práce žádá o vyvinutí webové aplikace, která umožní uživateli ptát se na informace týkající se omezeného množství dokumentů uložených v cloudovém repozitáři. Na základě proběhlé analýzy požadavků a dostupných metod, bude navrhnut a implementován systém umožňující RAG funkcionalitu. V této kapitole bude pozornost věnována především klíčovým aspektům návrhu architektury a volbě technologií. Cílem je podrobně popsat strukturu systému, jeho komponenty a interakce mezi nimi. Zároveň dojde k představení technologického stacku, který bude využit při implementaci projektu.

2.1 Volba technologií

Volba správných technologií je základním kamenem úspěšného návrhu a implementace webové aplikace. Každý technologický stack má své specifické vlastnosti, které mohou značně ovlivnit výkon, škálovatelnost, bezpečnost a uživatelskou přívětivost aplikace. Tato sekce se zaměří na pečlivý výběr technologií, které nejlépe odpovídají požadavkům a cílům výsledné aplikace. Zahrnut bude výběr frameworku pro backend, který zajistí zpracování serverových požadavků, a frontend, který zprostředkuje komunikaci backendu s uživatelem. Dále bude diskutován výběr modelů strojového učení, neboť hrají esenciální roli ve zpracování dokumentů. Výběr cloudového a vektorového úložiště bude taktéž reflektovat potřeby aplikace zejména v oblasti dostupnosti.

2.1.1 Multilingual-e5-base

Na základě analýzy byl pro tvorbu vektorových reprezentací zvolen model *multilingual-e5-base* od autora *intfloat*, neboť vyhovuje kriteriím aplikace. Podporuje český jazyk, je volně dostupný

z portálu *HuggingFace* a obecně dosahuje state-of-the-art výsledků. Zejména z hlediska budoucí rozšiřitelnosti je vhodným kandidátem, protože se počítá i s uchováváním dokumentů v anglickém jazyce. Model je také volně distribuován ve formátu ONNX, čímž se zvyšuje jeho použitelnost.

ONNX neboli *Open Neural Network Exchange* je otevřený ekosystém, který umožňuje použití modelů strojového učení vytvořených v různých knihovnách napříč odlišnými platformami a optimalizátory bez ztráty funkcionality nebo efektivity. ONNX je zásadní pro interoperabilitu těchto nástrojů, neboť poskytuje standardizovaný formát pro modely neuronových sítí, který umožňuje jejich snadný přenos mezi frameworky. Díky ONNX mohou vývojáři efektivněji využívat hardware a zlepšit výkon neuronových sítí v produkčních prostředích, protože se nemusí omezovat na použití jedné technologie [65].

2.1.2 Llama3

Pro generativní část byl vybrán model *Llama3* ve variantě *8B-instruct* s kvantizací *Q4_0*. *Llama3* model byl vybrán jednoznačně, neboť byl publikován v době vzniku této práce a z hlediska dosaženého výkonu nemá aktuálně konkurenci. Varianta modelu byla zvolena za účelem možnosti jeho spuštění na hardweru dostupném pro běžného uživatele. *LLama3* bude zprovozněn skrze službu *Ollama*. Jedná se o otevřenou platformu určenou pro snadné nasazování a správu LLM **lokálně** na vlastním hardwaru. Lokální správa je velmi důležitý aspekt v kontextu firemních dat, jelikož se zabrání odesílání interních dat třetí straně, čímž se sníží bezpečnostní riziko. *Ollama* podporuje různé konfigurace a velikosti modelů, umožňuje jejich snadnou škálovatelnost a poskytuje rozhraní pro správu, monitorování a optimalizaci provozu [51, 66].

2.1.3 Google Drive

Pro splnění funkcionality propojení aplikace s cloudovým úložištěm byl, jako ideální řešení, zvolen Google Drive. Toto rozhodnutí bylo podpořeno několika výhodami, které Google Drive nabízí. Díky široce dostupnému API je možné snadno komunikovat s úložištěm přímo z aplikace napsané v Javě. Toto API podporuje všechny základní operace pro správu dokumentů jako je nahrávání, stahování či mazání. Dalším důležitým faktorem je vysoká úroveň zabezpečení a spolehlivost, kterou obecně Google poskytuje. Tyto aspekty činí Google Drive vynikající volbou pro cloudové úložiště v rámci navrhované aplikace. Pro napojení postačí servisní účet, čímž se aplikace vyhne přihlašování uživatele [67].

2.1.4 Spring Boot

Pro vývoj backendové části aplikace byl zvolen framework Spring Boot. Jedná se o open-source Java framework, založený na Spring frameworku. Spring Boot je vysoce ceněn pro svou schopnost rychle a efektivně vyvíjet robustní a škálovatelné webové aplikace. Jednou z klíčových vlastností frameworku je jeho „opinionated“ přístup, což znamená, že má přednastavené konfigurace, které fungují dobře pro většinu aplikací a snižují potřebu mnoha konfiguračních souborů. Tím maximalizuje produktivitu vývojářů, protože jim umožňuje se primárně zaměřit na podstatu aplikace. Pro potřeby této aplikace bude využito především webového starter balíčku s webovým serverem *Tomcat* [68].

2.1.5 Spring AI

Spring AI je inovativní projekt zaměřený na zjednodušení vývoje aplikací, které integrují funkcionality umělé inteligence (AI). Tento projekt, inspirovaný úspěšnými platformami v jazyce Python jako LangChain a LlamaIndex, si klade za cíl rozšířit možnosti využití generativní AI i mimo Python. Spring AI se vyznačuje poskytováním flexibilních abstrakcí, které tvoří základ pro rychlý a efektivní vývoj AI aplikací. Tyto abstrakce umožňují snadnou výměnu komponent bez potřeby zásadních změn v kódu, což značně zjednodušuje proces adaptace na nové požadavky a technologie. Platforma podporuje širokou škálu klientů pro integraci embedding modelů od předních poskytovatelů jako OpenAI, Microsoft, Amazon či Google. Nutno podotknout, že napojení se na modely od těchto výrobců vyžaduje API klíč, který je zpoplatněný. Pro funkci této aplikace se integruje zmíněný model ve formátu ONNX z HuggingFace přes transformerového klienta, který je součástí Spring AI. Platforma je rovněž kompatibilní s různými poskytovateli vektorových úložišť, včetně Azure Vector Search, Chroma, Milvus, Neo4j, PostgreSQL/PGVector, PineCone, Qdrant, Redis a Weaviate [69].

2.1.6 Vaadin

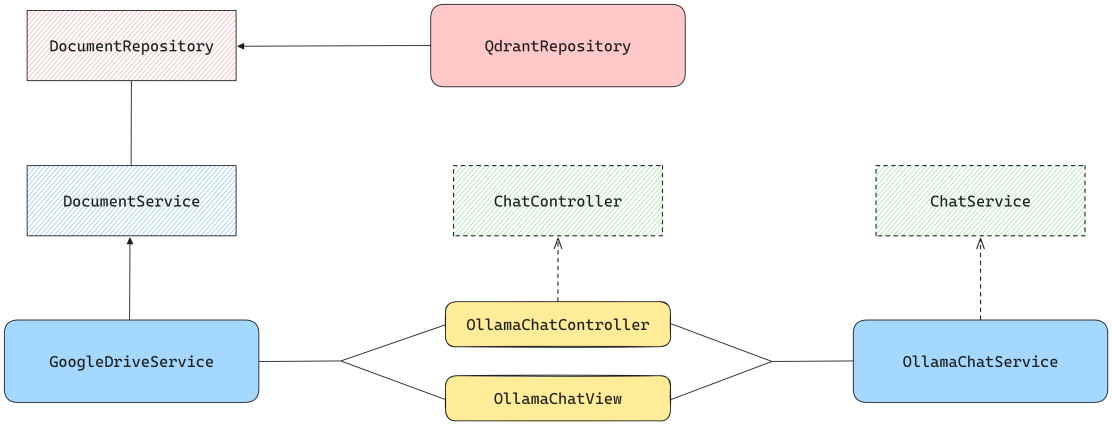
Framework Vaadin bude využit pro implementaci jednoduchého UI. Byl zvolen za účelem snadné integrace do Spring Boot ekosystému. Jeho komponenty nabízejí moderní vzhled a vysokou míru uživatelské interaktivity, což umožňuje vytvářet intuitivní a vizuálně přitažlivé uživatelské rozhraní bez zbytečné složitosti. Pro účely aplikace budou přizpůsobeny komponenty, které zobrazí chatovací okénko uživatele s asistentem a vstupní textové pole pro zadávání dotazu uživatele [70].

2.1.7 Qdrant

Za účelem ukládání embeddingů vytvořených transformačním modelem byla zvolena Qdrant vektorová databáze. Především díky jednoduchému nasazení přes Docker, kvalitní dokumentaci, možnosti využití v rámci Spring AI a vysoké efektivitě.

2.2 Architektura

Architektura výsledné aplikace bude postavena na principu uvedeném v analytické části, kde byl popsán workflow RAG systému 3.2.1. Celkový návrh aplikace bude strukturován do třívrstvé architektury. Tento model rozděluje aplikaci na tři základní vrstvy: **persistentní**, **logickou** a **prezentační**. Každá z těchto vrstev má specifické úkoly a odpovědnosti, čímž je umožněna nejen lepší organizace kódu, ale také zjednodušení údržby a možnosti rozšiřování aplikace. Výběr třívrstvé architektury podporuje vysokou úroveň abstrakce mezi jednotlivými komponentami, což vede k robustnějšímu a bezpečnějšímu designu [71].



Obrázek 2.1 Diagram návrhu tříd

2.2.1 Persistentní vrstva

Persistentní (datová) vrstva v architektuře webových aplikací slouží k ukládání dat tak, aby byla dostupná i po ukončení komunikačního spojení se serverem. Tato vrstva obvykle využívá databází nebo jiných úložišť a je zodpovědná za správu datových objektů, jejich bezpečné ukládání a manipulaci. Nejčastěji zahrnuje tzv. **CRUD** operace jako jsou vkládání (create), čtení (read), aktualizace (update), mazání (delete) [71].

V tomto projektu bude persistence řízena pomocí abstraktní třídy *DocumentRepository*, která poslouží jako základ pro specifické implementace správy dokumentů v různých vektorových úlo-

žištích. Dále uchovává instanci úložiště, kterou musí dodat konkrétní třída, jenž z této abstraktní podědí. Úložiště je pevně propojené s modelem pro transformaci embeddingů, čímž je usnadněna práce s modelem, a stačí pouze komunikovat s úložištěm. Třída *DocumentRepository* definuje následující metody:

- **`add`** umožňuje přidání seznamu dokumentů do vektorového úložiště.
- **`deleteByIds`** poskytuje základní mechanismus pro odstranění dokumentů podle jejich identifikátorů. Vrací *Optional<Boolean>* indikující úspěch operace.
- **`similaritySearch`** umožňuje vyhledávání dokumentů na základě podobnosti s poskytnutým vyhledávacím dotazem, který si bere metoda parametrem. Vrací seznam k nejpodobnějších dokumentů, kde k je určeno dotazem. Případně jeho výchozí hodnota je nastavena na číslovku čtyři.
- **`recreateCollection`** je abstraktní metoda, ve které implementující musí dodat funkcionalitu smazání celé kolekce vektorů a následně její znovuvytvoření.
- **`isCollectionEmpty`** je abstraktní metoda, která vrací *boolean* hodnotu signalizující, jestli je kolekce prázdná.
- **`isCollectionExists`** je abstraktní metoda, která verifikuje, jestli je kolekce inicializována.

2.2.2 Logická vrstva

Logická (byznys) vrstva představuje „střední část“, která zpracovává aplikační logiku a interaguje s rozhraním persistentní vrstvy. V rámci této vrstvy se realizují všechny kritické procesy, jako jsou výpočty, správa transakcí, komunikace s externími službami či rozhodování o toku dat a dalších operací [71].

Logická vrstva bude v projektu implementována prostřednictvím služeb *DocumentService* a *ChatService*.

DocumentService je abstraktní třída koordinující operace spojené se správou dokumentů. Uchovává instance tříd *DocumentRepository* a *TextSplitter*. *DocumentService* definuje chování následujících metod:

- **`downloadAndProcessAll`** orchestrálně spravuje celý proces stahování a zpracování dokumentů. Začíná tím, že smaže a znovu vytvoří kolekci dokumentů v repozitáři. Následně stáhne metadata o souborech, které se využijí pro stažení celého obsahu jednotlivých dokumentů. Účinnou extrakci textu z obsahu dokumentu obstará utilita *TikaDocumentReader*. Extrahovaný text dále využije instance třídy *TextSplitter* k rozdělení textu na menší jednotky

(chunky). Tyto jednotky textu jsou následně delegovány do repozitáře, kde se transformují na vektorovou reprezentaci a přidají do databáze.

- `retrieveContextWithMetadata` je abstraktní metoda, která na základě zadaného požadavku získá kontextový obsah a ten spolu s jeho metadaty vrátí ve formě *ContextDto*.
- `retrieveContext` je abstraktní metoda, která na základě zadaného požadavku vrátí pouze textový kontext.
- `fetchFiles` je abstraktní metoda, která vrací základní informace o souborech v cloudovém úložišti. Jedná se o *id, name, mimeType, webContentLink* a *viewContentLink*.
- `downloadFileContent` je abstraktní metoda, která na základě metadat souboru obstará jeho obsah, ten vrací v podobě *Resource*.

ChatService definuje rozhraní, prostřednictvím kterého lze komunikovat se službou zpracovávající odpověď pro uživatele. Rozhraní využívá zmíněný objekt *ContextDto* k získání otázky od uživatele, předzpracovaného kontextu a jeho metadat. *ChatService* rozhraní se skládá z následujících metod:

- `generateResponse` získá odpověď na základě kontextu a navrací volajícímu *ChatResponse*.
- `downloadAndProcessAll` získává odpověď po částech na základě kontextu a navrací volajícímu *Flux<ChatResponse>*. *Flux<>* zajistí asynchronní komunikaci podle modelu „producent & konzument“, čímž je umožněné postupné získávání odpovědi.

2.2.3 Prezentační vrstva

Prezentační vrstva je zodpovědná za interakci s uživatelem, prezentaci dat a funkcí aplikace v uživatelsky přívětivém formátu. Tato vrstva zahrnuje veškeré uživatelské rozhraní (UI), které jsou vytvářeny pomocí různých frontendových technologií jako HTML, CSS a JavaScript, a často využívají frameworky jako React nebo Angular pro dynamické a responzivní chování. Prezentační vrstva je navržena tak, aby byla vizuálně atraktivní a intuitivní pro koncového uživatele, zároveň zajišťuje, že uživatelské rozhraní je konzistentní a efektivně komunikuje s logickou vrstvou. Zahrnuje také zpracování uživatelských vstupů a prezentaci zpracovaných dat zpět uživateli. Na serverové části systému však je do prezentační vrstvy zařazena i mechanika controllerů, která řídí a zprostředkovává komunikaci s ostatními vrstvami aplikace díky využití aplikačního rozhraní (API) [71].

V rámci projektu je prezentační vrstva modelována dvěma přítupy. První možnost uživateli zobrazí grafické rozhraní (GUI), přes které uživatel napřímo může komunikovat s aplikací, pokládat otázky a číst příslušné odpovědi. Grafické rozhraní bude implementované ve třídě *OllamControllerView* a jeho podrobná implementace bude blíže představena v implementační části 3.2.1. Druhá možnost prezentační vrstvy zpřístupní REST API, které umožní použití aplikace jako backendovou (serverovou) komponentu pro frontendového klienta. Pro tuto část je připravené rozhraní *ChatController* a *DocumentController*. Rozhraní nám dává jasný předpis metod, které implementující musí dodat. Rozhraní *ChatController* využívá *RequestDto* pro získáníní vstupní otázky od uživatele. Dále využívá DTO *ResponseDto* pro mapování výstupu do příslušného formátu. Skládá se z textové odpovědi a kotextových metdat, která se serializují jen pokud byla získána. Rozhraní *ChatController* definuje následující metody:

- `chat` validuje, jestli vstupní textová sekvence od uživatele není prázdná. Získá data přes logickou vrstvu a transformuje výstup uživateli jednotně tak, že odpověď je odeslána až ve chvíli, kdy je zcela kompletní.
- `chatStream` se vstupem pracuje stejně jako *chat* metoda. Ve formě *Flux<ResponseDto>* je odeslán výstup, takže se nečeká na ucelenou odpověď, ale výstup je odesílán po částech.

Rozhraní *DocumentController* definuje komunikaci s nezpracovanými dokumenty. Definuje následující metodu:

- `processData` zřizuje komunikaci se správou dokumentů. Výsledkem této operace by mělo být připravené úložiště se zpracovanými dokumenty pro vyhledávací účely. Návratovou hodnotou je *ResponseEntity<ProcessResultDto>* obsahují informaci o úspěšném zpracování.

Schémata všech vstupních a výstupních DTO, a chování REST API koncových bodů budou volajícímu k dispozici přes OpenAPI dokumentaci.

Kapitola 3

Implementace

V této kapitole dojde k realizaci jednotlivých komponent systému podle definovaného návrhu v předchozí kapitole. Zahrnuta bude počáteční konfigurace a struktura projektu, která představí vybrané ukázky zdrojového kódu demonstrující klíčové aspekty implementace. Dále budou zmíněny nástroje, které byly využity pro vývoj aplikace a popis procesu jejího nasazení. Na závěr dojde k otestování implementovaného řešení.

3.1 Konfigurace komponent

Před implementací je nutné provést jisté konfigurační nastavení. Pro centralizaci obecné konfigurace aplikace je určen soubor *application.yaml* v adresáři resources. Zde jsou nastaveny parametry pro připojení databáze Qdrant, propojení Ollama služby, ONNX modelu a další. Následně bude více představená konfigurace Google Drive cloudového úložiště a správa závislostí.

3.1.1 Google Drive

Pro integraci Google Drive s aplikací Spring Boot bylo postupováno následujícími kroky. Nejprve došlo k vytvoření projektu **GSight** v Google Cloud Console a povolení API Google Drive. Následně byl vytvořen servisní účet, který byl k tomuto projektu přiřazen. K účtu byl vytvořen klíč, pomocí něhož lze aplikaci propojit s úložištěm a využívat API služeb. Klíč *gsight_key.json* je uložen v adresáři resources/keys a je tak dostupný pro využití aplikací.

3.1.2 Správa závislostí

Spring framework efektivně řeší správu závislostí prostřednictvím mechanismu *dependency injection*, který je implementován pomocí **Bean**. Aby se předešlo vytváření redundantních instancí tříd, Spring většinou spravuje jedinou instanci každého Beanu jako singleton, který poté injektuje do ostatních komponent, kde je potřeba. Tento přístup odděluje konfiguraci od implementace kódu, což zvyšuje modularitu a usnadňuje testování. Bean mohou být konfigurovány deklarativně pomocí anotací, jako jsou *@Component, @Service* a *@Repository*, nebo programově ve speciálních konfiguračních třídách.

Výpis kódu 3.1 Konfigurační třída GoogleDriveConfig

```
@Configuration
public class GoogleDriveConfig {

    @Value("${google.drive.key-path}")
    private Resource keyResource;

    @Value("${spring.application.name}")
    private String applicationName;

    @Bean
    public Drive googleDriveService() {
        try {
            GoogleCredentials credentials = GoogleCredentials
            .fromStream(keyResource.getInputStream())
            .createScoped(Collections.singletonList(DriveScopes.DRIVE));

            return
                new Drive.Builder
                    (
                        GoogleNetHttpTransport.newTrustedTransport(),
                        GsonFactory.getDefaultInstance(),
                        new HttpCredentialsAdapter(credentials)
                    )
                    .setApplicationName(applicationName)
                    .build();
        }
        catch (Exception ex) {
            throw new BeanCreationException(
            "googleDriveService",
            "Unable to create Google Drive Service",
            ex);
        }
    }
}
```

Ukázka 3.1 demonstruje konfiguraci úložiště Google Drive. Anotace *@Configuration* indikuje, že se jedná o konfigurační třídu, ve které mohou existovat definice Bean. *@Value* anotace přebírá parametr z konfiguračního souboru *application.yaml*. Anotace *@Bean* označuje, že metoda *googleDriveService* vrací novou instanci třídy *Drive*, kterou má Spring uchovat jako singleton pro

inejkci závislostí. V případě, kdy z nějakého důvodu nemůže být instance třídy *Drive* vytvořena, metoda vyvolá výjimku *BeanCreationException*. Tato výjimka zamezí spuštění programu.

V Adresáři config se dále nacházejí konfigurační třídy *CollectionConfigManager, EmbeddingConfig, OllamaConfig* a *QdrantConfig*.

3.2 Struktura projektu

Projekt byl inicializován pomocí nástroje Spring Initializr, který poskytl základní strukturu. Do této struktury byly postupně implementovány dílčí komponenty, čímž vzniklo funkční řešení.

3.2.1 Interakce

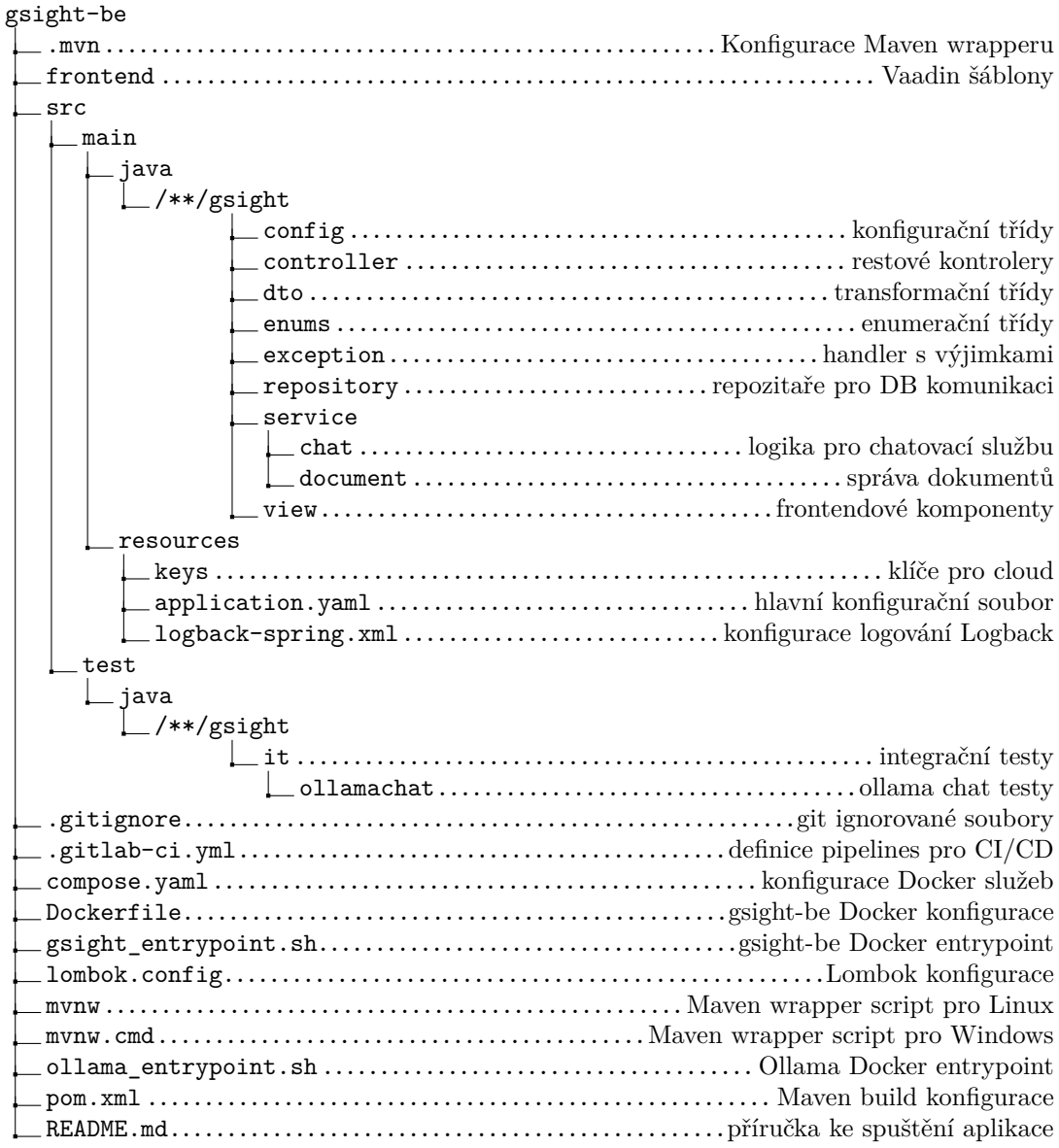
Uživatel může s aplikací interagovat přes grafické rozhraní (GUI) nebo rozhraní REST.

REST API je realizováno skrze Spring framework, konkrétně pomocí anotace *@RestController*, což umožňuje snadnou integraci HTTP požadavků a odpovědí modelu. Například, *GoogleDocumentController* přijímá požadavky pro zpracování dokumentů. Při zaslání POST požadavku na koncový bod s URI **/api/document/process-data** metoda *processData* spustí službu *googleDocumentService* pro zpracování dat, a vrátí *ResponseEntity* s výsledkem procesu.

OllamaChatController nabízí koncové body pro interaktivní chat. Koncový bod **/api/ollama/chat** přijímá POST požadavek s dotazem ve formátu JSON, který je dále zpracováván. Pokud se k dotazu nenajde žádný kontext, vrátí se předdefinovaná odpověď „I do not know the answer.“. V opačném případě se předá kontext s otázkou uživatele *OllamaChatService*, která získá vygenerovanou odpověď z jazykového modelu. Tento controller rovněž podporuje streaming odpovědí skrze endpoint **/api/ollama/stream-chat** (viz 3.2), který využívá media typ application/x-ndjson pro asynchronní posílání dat.

Výpis kódu 3.2 API koncový bod pro asynchronní komunikaci

```
@Override
@PostMapping(value = "/stream-chat", produces = MediaType.APPLICATION_NDJSON_VALUE)
public Flux<ResponseDto> streamChat(@RequestBody RequestDto requestDto) {
    ContextDto context = formContextWithNullableMetadata(requestDto.question());
    return context
            .context()
            .isEmpty()
            ? ResponseDto.emptyContextFluxResponse()
            : ResponseDto.fromChatStream(ollamaChatService.generateStreamResponse(context));
}
```



Obrázek 3.1 Struktura projektu gsight-be

Grafické rozhraní je implementováno s využitím komponent Vaadin frameworku. *OllamaChatView* představuje hlavní objekt pro uživatelskou interakci, kde uživatelé mohou zadávat dotazy prostřednictvím textového pole a následně zobrazovat odpovědi v reálném čase. Komponenta *MessageInput* slouží pro zadávání dotazů a je napojena na *submitListener*, který reaguje na odeslané dotazy. Odpovědi jsou zobrazovány jako markdown zprávy, kde barevně rozlišené avatary reprezentují zprávy uživatele a systému. V případě chyby systém zobrazí chybovou zprávu a umožní uživateli pokračovat v interakci.

Výpis kódu 3.3 Zpracování asynchronní komunikace ve view

```
Mono.defer(() -> Mono.just(googleDocumentService.retrieveContextWithMetadata(question)))
    .subscribeOn(Schedulers.boundedElastic())
    .flatMap(context -> {
        if (context.context().isEmpty()) {
            assistantMessage.appendMarkdownAsync(
                ResponseDto.emptyContextResponse().getResponse());
            return Mono.empty();
        }

        else {
            return ollamaChatService.generateStreamResponse(context)
                .map(res -> ResponseDto.fromChatResponse(res).getResponse())
                .doOnNext(assistantMessage::appendMarkdownAsync)
                .last()
                .flatMap(last -> Mono.fromRunnable(
                    () -> displayMetadata(context, assistantMessage)));
        }
    })
    .doOnError(ex -> {
        logger.error(ex.getMessage(), ex);
        messageList.remove(assistantMessage);
        messageList.add(errorMessage);
        errorMessage.appendMarkdownAsync(SYSTEM_ERROR_MESSAGE + ex.getMessage());
    })
    .onErrorResume(e -> Mono.empty())
    .subscribe();
```

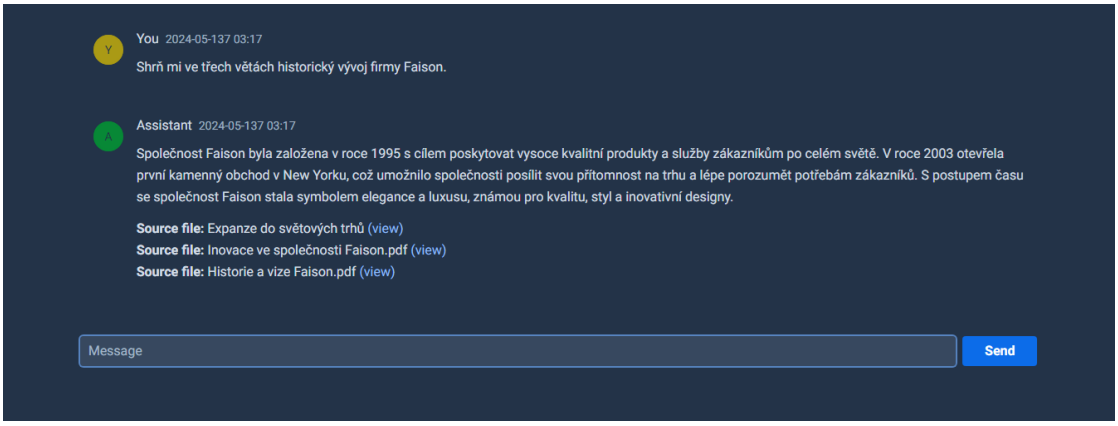
V ukázce 3.3 je znázorněna část akce *submitListener*. Zde byla použita technologie reaktivního programování s využitím Project Reactor, což umožňuje asynchronní a neblokující zpracování dat. Díky odložené inicializaci pomocí *Mono.defer* se aplikace vyhýbá zbytečnému zatížení a zahajuje zpracování dotazů až v momentě skutečné potřeby. V případě chyby nebo nedostatku dat je systém schopen uživateli okamžitě poskytnout relevantní zpětnou vazbu.

Metoda *displayMetadata* v ukázce 3.4 slouží k zobrazení metadat kontextu v uživatelském rozhraní aplikace. Funkce iteruje přes dostupná metadata, přičemž vytváří HTML elementy, které identifikují zdrojový soubor a poskytují přímý odkaz pro jeho zobrazení. Pro každý záznam je dynamicky vytvořen tučný text s názvem souboru následovaný odkazem, který umožňuje uživate-

Výpis kódu 3.4 Zobrazení metadat ve view

```
private static void displayMetadata(ContextDto context, MarkdownMessage assistantMessage ) {
    Element main = new Element(Tag.valueOf("div"), "");
    context.contextMetadata()
        .forEach(metadata -> {
            main.appendElement("b").text("Source file: ");

            String source = Optional.ofNullable(metadata.get(
                FileMetadataType.SOURCE.getValue())).orElse("").toString();

            main.appendText(source + " ");

            String viewLink = Optional.ofNullable(metadata.get(
                FileMetadataType.VIEW_LINK.getValue())).orElse("").toString();

            main.appendElement("a")
                .text("(view)")
                .attr("href", viewLink)
                .attr("target", "_blank");

            main.appendChild(new Element(Tag.valueOf("br"), ""));
        });
        assistantMessage.appendMarkdownAsync(main.outerHtml());
}
```

lům zobrazit soubor v novém okně prohlížeče. Tento proces je zabalován do div kontejneru, který je nakonec vložen do markdown zprávy asistenta. Grafické rozhraní je zachyceno na obrázku 3.2.



Obrázek 3.2 Interakce uživatele přes grafické rozhraní

3.2.2 Služby

Vnitřní logika systému je implementována pomocí služeb, které komunikují s repozitářem nebo jinými službami.

Třída *GoogleDocumentService* je specifická implementace abstraktní třídy *DocumentService*, určená pro zpracování dokumentů uložených v Google Drive. V návrhu 2.2.2 bylo představeno, že primární funkcionalitou třídy *DocumentService* je metoda *downloadAndProcessAll*, která umožňuje stahování a zpracování dokumentů podle jejich metadat. Metoda využívá dalších abstrtaktních metod, které právě třída *GoogleDocumentService* implementuje.

Výpis kódu 3.5 Načítání metadat souborů z Google Drive

```
@Override
public List<FileDto> fetchFiles() {
    List<FileDto> fetchedFiles = new ArrayList<>();
    try {
        String pageToken = null;
        do {
            FileList result = googleDriveService.files().list()
                .setQ("'" + folderId + "' in parents")
                .setPageSize(10)
                .setFields(
                    "nextPageToken, files(id, name, mimeType, webContentLink, webViewLink)")
                .setPageToken(pageToken)
                .execute();
            List<File> googleFiles = result.getFiles();
            fetchedFiles.addAll(googleFiles.stream().map(FileDto::fromGoogleFile).toList());
            pageToken = result.getNextPageToken();
        }
        while (pageToken != null);
    }
    catch (IOException ex) {
        throw new FileIOException("Error occurred during files fetching", ex);
    }
    return fetchedFiles;
}
```

Metoda *fetchFiles* z ukázky 3.5 zajišťuje výběr souborů z Google Drive dle specifikovaného ID složky. Toto ID je načteno z konfigurace *application.yaml*. Soubory jsou iterativně získávány a konvertovány na interní datový typ *FileDto*, který reprezentuje základní metadata o souboru včetně odkazů pro webové zobrazení a stahování jeho obsahu.

Pro stahování obsahu souborů z Google Drive třída *GoogleDocumentService* používá metodu *downloadFileContent* z úkázky 3.6, která na základě MIME typu souboru rozhoduje, zda bude soubor exportován nebo zda bude stažen přímo. Tento proces je nezbytný, neboť Google dokumenty *docs, sheets* či *slides* je nutné exportovat do ekvivalentního formátu Office Open XML. V případě neúspěchu operace je vyvolána výjimka *FileIOException*, signalizující problém se stahováním souboru.

Dále *GoogleDocumentService* poskytuje metody pro vyhledávání a získávání kontextu z dokumentů na základě textového dotazu. Metoda *retrieveContextWithMetadata* (viz 3.7) formuluje a následně odešle *SearchRequest* dokumentovému repozitáři, který vrátí kontextové informace

Výpis kódu 3.6 Stahován obsahu souboru z Google Drive

```
@Override
public ByteArrayResource downloadFileContent(FileDto file) {
    try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {

        Optional<String> fileMimeType = ExportFormatMapper.getExportFormat(file.getMimeType());

        if (fileMimeType.isPresent())
            googleDriveService
            .files()
            .export(file.getId(), fileMimeType.get())
            .executeMediaAndDownloadTo(baos);

        else
            googleDriveService.files().get(file.getId()).executeMediaAndDownloadTo(baos);

        return new ByteArrayResource(baos.toByteArray());
    }

    catch (IOException ex) {
        throw new FileIOException(
            String.format("File %s could not be downloaded", file.getName()), ex);
    }
}
```

Výpis kódu 3.7 Metoda získávání kontextu s metadaty

```
@Override
public ContextDto retrieveContextWithMetadata(String question) {
    var request = SearchRequest.query(question).withSimilarityThreshold(similarityThreshold);
    var documents = documentRepository.similaritySearch(request);
    return new ContextDto(
        question,
        formContext(documents),
        documents
            .stream()
            .map(Document::getMetadata)
            .peek(metadata -> metadata.remove("distance"))
            .distinct()
            .toList()
    );
}
```

s metaday, které jsou poté uceleně předány volajícímu. Metoda *retrieveContext* navrací kontext bez metadat.

Výpis kódu 3.8 Komunikace s Ollama klientem

```
@Override
public Flux<ChatResponse> generateStreamResponse(ContextDto document) {
    return ollamaChatClient.stream(formPrompt(document))
        .onErrorMap(ex -> new ChatServiceUnavailableException(OLLAMA_ERROR_MESSAGE, ex));
}
```

Služba *OllamaChatService* využívá klienta *OllamaChatClient* pro generování odpovědí na dotaz s přidruženým kontextem. Funkce *generateResponse* synchronně zpracovává odpovědi metodou *call* na *ollamaChatClient*, zatímco *generateStreamResponse* z ukázky 3.8 poskytuje odpovědi asynchronně ve formě streamu. V případě chyb v dostupnosti služby je vyvolána výjimka *ChatServiceUnavailableException*. Pomocná funkce *formPrompt* sestavuje *Prompt* z kontextových informací a šablonových zpráv definovaných v *PromptMessageHelper*, které řídí generativní model v reakcích na dotazy.

3.2.3 Persistence

Služba *QdrantRepository* je konkrétní implementací abstraktní třídy *DocumentRepository*. Slouží ke správě kolekcí dokumentů v prostředí vektorového úložiště Qdrant. Klíčovou funkcí této třídy je metoda *recreateCollection* z ukázky 3.9, která se stará o obnovu nebo vytvoření nové kolekce dokumentů.

Výpis kódu 3.9 Znovuvytvoření kolekce

```
@Override
public void recreateCollection() {
    try {
        if (!isCollectionExists())
            vectorStore.afterPropertiesSet();

        else {
            qdrantClient.deleteCollectionAsync(CollectionConfigManager.getCollectionName()).get();
            vectorStore.afterPropertiesSet();
        }
    }

    catch (Exception ex) {
        throw new ReCreateCollectionException(
            "Failed to recreate the collection due to an unexpected error", ex);
    }
}
```

Proces obnovy kolekce zahrnuje kontrolu existence aktuální kolekce pomocí metody *isCollectionExists*. Pokud kolekce neexistuje, inicializuje se nová kolekce voláním metody *afterPropertiesSet* na objektu *vectorStore*. V případě, že kolekce existuje, je nejprve asynchronně smazána a poté znovu inicializována. Tato operace může být komplikována možnými výjimkami, na které je reagováno vyvoláním *ReCreateCollectionException* s popisem chyby.

3.2.4 Řešení výjimek

Výjimky vyvolané za běhu prostředí jsou zachyceny ve třídě *RestResponseExceptionHandler*, která globálně odchytává výjimky v celé aplikaci v rámci využití REST API a dále je zpracovává. Ukázka 3.10 demonstruje případ zachycení výjimky *ChatServiceUnavailableException*. Metoda *handleError* loguje výjimky a vrací *ResponseEntity* s objektem *ExceptionDto*, který obsahuje detaily o chybě.

Výpis kódu 3.10 Zachycení výjimky *ChatServiceUnavailableException*

```
@ExceptionHandler(ChatServiceUnavailableException.class)
protected ResponseEntity<ExceptionDto> handleChatServiceUnavailableException(
    ChatServiceUnavailableException ex) {
        return handleError(ex.getMessage(), ex, HttpStatus.SERVICE_UNAVAILABLE);
}

private ResponseEntity<ExceptionDto> handleError(
    String message,
    Throwable ex,
    HttpStatus status) {
    logger.error(message, ex);
    return new ResponseEntity<>(ExceptionDto.from(status.value(), message), status);
}
```

3.3 Nasazení

Proces nasazení orchestrálně řídí Docker Compose, který spravuje jednotlivé kontejnery v síti *gsight-backend*. Konfigurace tohoto procesu je specifikována v souboru *compose.yaml*. Nasazení zahrnuje tři hlavní služby: gsight, qdrant a ollama, které odpovídají kontejnerům gsight-be, qdrant-db a ollama-ai. Inicializace začíná sestavením Docker obrazu pro službu gsight z příslušného Dockerfile. Před spuštěním kontejneru gsight-be je však nezbytné nejprve nasadit ostatní dva kontejnery. Služba qdrant se stará o spuštění vektorového úložiště Qdrant, které na portu 6333 přijímá komunikaci přes protokol gRPC a na portu 6334 prostřednictvím REST rozhraní. Služba ollama zase řídí načítání a správu generativních modelů. V rámci *ollama_entrypoint.sh*

scriptu dojde nejprve k odstranění signalizačního souboru, ke spuštění ollama serveru a k následné kontrole dostupnosti specifikovaného modelu. Pokud model chybí, je automaticky stažen. Po jeho úspěšném načtení se vytvoří signalizační soubor, který indikuje, že služba je připravena k obsluze požadavků přes REST rozhraní na portu 11434. Vzhledem k tomu, že Docker považuje službu za spuštěnou ihned po aktivaci ollama serveru, může dojít k zahájení aplikace gsight-be. Pokud by však nebyl model předem stažen, služba ollama by sice přijímala komunikaci, ale neměla by připravený model pro zpracování požadavků. Proto je ve scriptu *gsight_entrypoint.sh* zařazen mechanismus, který čeká na vytvoření signalizačního souboru *ollama_ready* ve sdíleném volume *ollama-gsight-shared*. Jakmile jsou všechny služby úspěšně spuštěné, je aplikace gsight-be připravena k interakci buď přes grafické rozhraní nebo prostřednictvím REST API přes localhost na portu 8080.

3.4 Použité nástroje

V rámci vývoje projektu byla využita řada specifických nástrojů, které značně usnadnily práci a zvýšily efektivitu vývoje. Tyto nástroje pokrývají široké spektrum úkolů od vývoje, přes verzování až po logování.

- **`IntelliJ IDEA`** je vývojové prostředí (IDE) od společnosti JetBrains, které bylo použito pro celkový vývoj aplikace. Díky své komplexní podpoře jazyka Java a množství integrovaných nástrojů byla implementace snazší.
- **`Lombok`** byl užitečný nástroj pro zjednodušení vývoje tím, že redukoval potřebu psát opakující se části kódu. Speciálně anotace *@RequiredArgsConstructor* automaticky generuje konstruktory s povinnými parametry, což eliminuje potřebu jejich ručního psaní. Anotace *@Slf4j* byla použita k automatickému injektování objektu *logger* v kombinaci s Logbabckem.
- **`Logback`** byl použit za účelem logování během vývoje a provozu aplikace. Jakým způsobem a v jakém formátu logy zaznamenávají je nastaveno v konfiguračním souboru *logback-spring.xml*.
- **`Gitlab`** byl využit pro verzování zdrojových kódů projektu. Pro test sestavení byla vytvořena pipeline v souboru *.gitlab-ci.yml*, která se spustí při každém commitu.
- **`Maven`** je nástroj pro řízení a automatizaci sestavování projektu. Maven byl využit k definování projektových závislostí, konfiguraci a sestavení aplikace. Jeho konfigurace se nachází v souboru *pom.xml*.

Kapitola 4

Testování

Testování je zásadní součástí každého implementačního procesu. Často bývá opomíjeno, protože vyžaduje dodatečný čas a zdroje, které mohou být v krátkodobém horizontu vnímány jako neefektivní vynaložení úsilí. Nicméně, až po otestování všech možných scénářů a stavů, do kterých se aplikace může dostat, lze testované funkcionality s jistotou garantovat. Úspěšné testování slouží jako důkaz, že výsledné řešení splňuje garantované funkcionality. V této kapitole bude představeno integrační testování backendové části aplikace, které je zaměřeno na ověření funkcionality vyhledávání kontextu.

4.1 Integrační testování

Hlavním cílem práce je nalezení relevantních informací z dokumentové sady na základě dotazu uživatele. Pro otestování této funkcionality byly zvoleny integrační testy, které se zaměří na testování, jestli hledaný kontext je relevantní vůči dotazu uživatele. Pro tyto potřeby bylo vygenerovnáno 20 dokumentů s pomocí ChatGPT [72]. Dokumenty jsou uložené ve složce GSight [1] v Google Drive ve formátech PDF, DOCX a Google Docs. Týkají se interní dokumentace fiktivní společnosti Faison.

Hlavní logika testování spočívá v metodách ve třídě *OllamaChatControllerIT*. V metodě *mockOllamaChatService* z ukázky 4.1 dochází k napodobení (mocking) chování služby *OllamaChatService* 3.2.2. Napodobení způsobí vrácení kontextu v metodě *generateResponse* zpět volajícímu, čímž dojde k zabránění komunikace s generativním modelem. Napodobení proběhne při každém zavolání metody.

[1] `https://drive.google.com/drive/u/1/folders/18auAJ9LU1_8k1t6B_yLQ5cxkqvG3zCX5`

Výpis kódu 4.1 Napodobení chování metody *generateResponse*

```
@BeforeEach
public void mockOllamaChatService(){
    Mockito.when(ollamaChatService.generateResponse(ArgumentMatchers.any(ContextDto.class)))
        .thenAnswer(invocation -> {
            ContextDto context = invocation.getArgument(0);
            return new ChatResponse(List.of(new Generation(context.context())));
        });
}
```

Metoda *contextSearch* v úkázce 4.2 definuje proces zasílání dotazů na koncový bod */api/ollama/chat* skrze HTTP POST požadavek. Kontrolován je stav odpovědi, zdroje souborů, ze kterých by měl být kontext nalezen a posléze samotný kontext. Ten je kontrolován na shodu klíčových frází, které by se v kontextové odpovědi měly vyskytnout.

Výpis kódu 4.2 Struktura testovací logiky

```
@SneakyThrows
private void contextSearch(
    RequestDto request,
List<String> expectedContext,
List<String> expectedSources) {
    MvcResult resultActions = mockMvc.perform(MockMvcRequestBuilders.post(OLLAMA_CHAT_POST)
            .contentType(MediaType.APPLICATION_JSON)
            .characterEncoding("UTF-8")
            .content(new ObjectMapper().writeValueAsString(request))
            .accept(MediaType.APPLICATION_JSON))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.jsonPath("$.contextMetadata[*].source")
            .value(Matchers.containsInAnyOrder(expectedSources.toArray())))
        .andReturn();

    String responseString = resultActions
    .getResponse()
    .getContentAsString(StandardCharsets.UTF_8);

    expectedContext.forEach(
        expectedItem -> assertThat(responseString, containsString(expectedItem)));
}
```

Výpis kódu 4.3 Testování kontextových informaci o benefitech

```
@Test
public void ok_Benefits(){
    contextSearch(
        RequestDto.from(
            TestContextHelper.Question.Q_BENEFITS),
        TestContextHelper.Requirement.REQ_BENEFITS,
        List.of(
            TestContextHelper.Source.SOURCE_BALANCE,
            TestContextHelper.Source.SOURCE_BENEFITS,
            TestContextHelper.Source.SOURCE_GROWTH
        ));
}
```

Test z úkazky 4.3 kontroluje dotaz „Vypiš mi v bodech, jaké mohu čerpat zaměstnanecké benefity.“. Očekává, že se v odpovědi vyskytnou fráze jako „Wellness programy“, „Flexibilní pracovní doba“ „Podpora školného a profesního rozvoje“ „Práce na dálku“ a další. Tato definice textových hodnot je spravována třídou *TestContextHelper*.

Závěr

Cílem teoretické části práce bylo analyzovat dostupné metody pro vyhledávání v textových dokumentech za účelem výběru optimální metody pro návrh a implementaci vyhledávacího systému. Dílčím cílem této části práce bylo provedení rešerše stávajících řešení, které se zabývají obdobnou problematikou a využít získané poznatky pro návrh vlastního systému.

Cílem praktické části práce bylo navrhnout a implementovat webovou aplikace v Javě nebo v Kotlinu, která by umožnila vyhledávání relevantních informací z dokumentů uložených v cloudovém úložišti. Následně bylo cílem výsledné řešení otestovat.

Všechny cíle a požadavky kladené na sytém se povedlo naplnit. Z provedené analýzy vyhledávacích metod byl vybrán RoBERTa model pro tvorbu embeddingů, Qdrant vektorové úložiště pro jejich ukládání a Llama3 model pro generování odpovědi. V rešerši existujících řešení byly identifikovány nedostatky, které podtrhují důležitost této práce. Architektura výsledného řešení byla navržena se zřetelem na budoucí rozšiřitelnost. Implementace webové aplikace GSight je řešením navržené architektury. GSight operuje s dokumenty v Google Drive cloudovém úložišti, předzpracovává je a tvoří z nich vektorovou reprezentaci s pomocí RoBERTa modelu, kterou ukládá v Qdrant vektorové databázi. Interakce s uživatelem je implementována prostřednictvím grafického rozhraní nebo skrze REST API. Při interakci se vyhledají sémanticky relevantní části dokumentů z databáze, které se předají modelu Llama3 pro generování odpovědi. Výsledné řešení bylo vhodně otestávno integračními testy a je připravené pro nasazení přes nástroj Docker.

Přínosem této práce je integrace technologií různého zaměření v rámci Java ekosystému. Použití Javy zde nabízí výhody v podobě spojení podnikových systémů s modely strojového učení bez nutnosti použití jazyka Python. Tento přístup nesnižuje výkon použitých technologií.

Bibliografie

1. BUTLER, Kevin. *Differences between Lexical and Semantic Search regarding relevancy* [online]. Pinecone, 2023 [cit. 2024-04-06]. Dostupné z: https://support.pinecone.io/hc/en-us/articles/9500075821981-Differences-between-Lexical-and-Semantic-Search-regarding-relevancy.
2. CHAUDHURI, Anshuk Pal. *Improving search relevancy powered by hybridization of semantic search and lexical search* [online]. Oracle, 2023 [cit. 2024-04-06]. Dostupné z: https://blogs.oracle.com/ai-and-datascience/post/improving-hybridization-search-semantic-lexical.
3. SKOPAL, Tomáš. *Boolean model of information retrieval* [online]. 2020. [cit. 2024-04-07]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/602570/course/section/92614/BIVWM_lecture02.pdf.
4. BRANS, Pat. *What is a Fuzzy Search?* [Online]. TechTarget, 2022 [cit. 2024-04-07]. Dostupné z: https://www.techtarget.com/whatis/definition/fuzzy-search.
5. SILVA, Eric. *What is Fuzzy Matching?* [Online]. Redis, 2022 [cit. 2024-04-06]. Dostupné z: https://redis.com/blog/what-is-fuzzy-matching/.
6. BOX, Luigi's. *Proximity Searching* [online]. Luigi's Box, 2023 [cit. 2024-04-06]. Dostupné z: https://www.luigisbox.com/search-glossary/proximity-searching/.
7. LIBRARY, Northcentral University. *Proximity Searching* [online]. Northcentral University, 2024 [cit. 2024-04-06]. Dostupné z: https://resources.nu.edu/researchprocess/proximity.
8. ACADEMY, JetBrains. *Fulltext indexes and Fulltext search* [online]. JetBrains Academy, [b.r.] [cit. 2024-04-07]. Dostupné z: https://hyperskill.org/learn/step/38541.

9. MONGODB, Inc. *Full-Text Search: What Is It And How It Works* [online]. MongoDB, 2022 [cit. 2024-04-06]. Dostupné z: https://www.mongodb.com/basics/full-text-search.

10. BAELDUNG. *Quick Intro to Full-Text Search with ElasticSearch* [online]. Baeldung, 2024 [cit. 2024-04-07]. Dostupné z: https://www.baeldung.com/elasticsearch-full-text-search-rest-api.

11. BAST, Hannah; BUCHHOLD, Björn; HAUSSMANN, Elmar. Semantic Search on Text and Knowledge Bases. *Foundations and Trends® in Information Retrieval*. 2016, roč. 10, s. 119–271. Dostupné z DOI: 10.1561/1500000032.

12. LEYS, Mathias. *Semantic search: A practical overview* [online]. ML6team, 2022 [cit. 2024-03-24]. Dostupné z: https://blog.ml6.eu/semantic-search-a-practical-overview-bf2515e7be76.

13. CASTILLO, Dylan. *Semantic Search with OpenSearch, Cohere, and FastAPI* [online]. Dylan Castillo, 2023 [cit. 2024-03-30]. Dostupné z: https://dylancastillo.co/semantic-search-with-opensearch-cohere-and-fastapi/.

14. NAVEZ, Lucie. *A Guide to Chunking Strategies for Retrieval Augmented Generation (RAG)* [online]. Sagacify, 2024 [cit. 2024-05-14]. Dostupné z: https://www.sagacify.com/news/a-guide-to-chunking-strategies-for-retrieval-augmented-generation-rag.

15. SCHWABER-COHEN, Roie. *Chunking Strategies for LLM Applications* [online]. Pinecone Systems, Inc., 2023 [cit. 2024-05-14]. Dostupné z: https://www.pinecone.io/learn/chunking-strategies/.

16. NAYAK, Plaban. Semantic Chunking for RAG. *The AI Forum* [online]. 2024 [cit. 2024-05-14]. Dostupné z: https://medium.com/the-ai-forum/semantic-chunking-for-rag-f4733025d5f5.

17. KHANNA, Chetna. Word, Subword, and Character-based Tokenization: Know the Difference. *Towards Data Science* [online]. 2023 [cit. 2024-05-14]. Dostupné z: https://towardsdatascience.com/word-subword-and-character-based-tokenization-know-the-difference-ea0976b64e17. Accessed on 2024-05-14.

18. TYAGI, Harshit. *The Evolution of Tokenization – Byte Pair Encoding in NLP* [online]. freeCodeCamp, 2021 [cit. 2024-05-14]. Dostupné z: https://www.freecodecamp.org/news/evolution-of-tokenization/.

19. HAGIWARA, Masato. *Complete Guide to Subword Tokenization Methods in the Neural Era* [online]. Octanove Institute, 2021 [cit. 2024-05-14]. Dostupné z: https://blog.octanove.org/guide-to-subword-tokenization/.

20. MIKOLOV, Tomas; CHEN, Kai; CORRADO, G.s; DEAN, Jeffrey. Efficient Estimation of Word Representations in Vector Space. *Proceedings of Workshop at ICLR*. 2013, roč. 2013.

21. NSS. *Understanding Word Embeddings: From Word2Vec to Count Vectors* [online]. Analytics Vidhya, 2017 [cit. 2024-03-30]. Dostupné z: https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/.

22. HUILGOL, Purva. *4 Sentence Embedding Techniques One Should Know* [online]. Analytics Vidhya, 2020 [cit. 2024-03-30]. Dostupné z: https://www.analyticsvidhya.com/blog/2020/08/top-4-sentence-embedding-techniques-using-python/.

23. SERRANO, Luis. *What is Semantic Search?* [Online]. Cohere, 2023 [cit. 2024-03-30]. Dostupné z: https://txt.cohere.com/what-is-semantic-search/.

24. TENSORFLOW. *Word embeddings* [online]. TensorFlow, 2022 [cit. 2024-03-30]. Dostupné z: https://www.tensorflow.org/text/guide/word_embeddings.

25. DESAGULIER, Guillaume. *Word embeddings: the (very) basics* [online]. Hypotheses, 2018 [cit. 2024-03-31]. Dostupné z: https://corpling.hypotheses.org/495.

26. SCHWABER-COHEN, Roie. *Vector Embeddings for Developers: The Basics* [online]. Pinecone Systems, Inc., 2023 [cit. 2024-03-31]. Dostupné z: https://www.pinecone.io/learn/vector-embeddings-for-developers/.

27. BROWNLEE, Jason. *How to One Hot Encode Sequence Data in Python* [online]. Machine Learning Mastery, 2019 [cit. 2024-04-01]. Dostupné z: https://machinelearningmastery.com/how-to-one-hot-encode-sequence-data-in-python/.

28. SARKAR, Dipanjan. *Implementing Deep Learning Methods and Feature Engineering for Text Data: The Continuous Bag of Words (CBOW)* [online]. KDnuggets, 2018 [cit. 2024-04-01]. Dostupné z: https://www.kdnuggets.com/2018/04/implementing-deep-learning-methods-feature-engineering-text-data-cbow.html.

29. SARKAR, Dipanjan. *Implementing Deep Learning Methods and Feature Engineering for Text Data: The Skip-gram Model* [online]. KDnuggets, 2018 [cit. 2024-04-01]. Dostupné z: https://www.kdnuggets.com/2018/04/implementing-deep-learning-methods-feature-engineering-text-data-skip-gram.html.

30. BOJANOWSKI, Piotr; GRAVE, Edouard; JOULIN, Armand; MIKOLOV, Tomas. *Enriching Word Vectors with Subword Information*. 2017. Dostupné z arXiv: 1607.04606 [cs.CL].

31. CHAUDHARY, Amit. *A Visual Guide to FastText Word Embeddings* [online]. Amit Chaudhary, 2020 [cit. 2024-04-03]. Dostupné z: https://amitness.com/2020/06/fasttext-embeddings/.

32. V, Krithika. *Introduction to FastText Embeddings and its Implication* [online]. Analytics Vidhya, 2023 [cit. 2024-04-03]. Dostupné z: https://www.analyticsvidhya.com/blog/2023/01/introduction-to-fasttext-embeddings-and-its-implication/.

33. THAPLIYAL, Hari. *Embedding with FastText* [online]. Hari Thapliyal, 2023 [cit. 2024-04-03]. Dostupné z: https://dasarpai.github.io/dsblog/Embedding-with-FastText.

34. DEVLIN, Jacob; CHANG, Ming-Wei; LEE, Kenton; TOUTANOVA, Kristina. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. Dostupné z arXiv: 1810.04805 [cs.CL].

35. WOLF, Thomas; DEBUT, Lysandre; SANH, Victor; CHAUMOND, Julien; DELANGUE, Clement; MOI, Anthony; CISTAC, Pierric; RAULT, Tim; LOUF, Rémi; FUNTOWICZ, Morgan; DAVISON, Joe; SHLEIFER, Sam; PLATEN, Patrick von; MA, Clara; JERNITE, Yacine; PLU, Julien; XU, Canwen; SCAO, Teven Le; GUGGER, Sylvain; DRAME, Mariama; LHOEST, Quentin; RUSH, Alexander M. Transformers: State-of-the-Art Natural Language Processing. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, 2020, s. 38–45. Dostupné také z: https://www.aclweb.org/anthology/2020.emnlp-demos.6.

36. KUMARI, Kajal. *Step-by-Step BERT Implementation Guide* [online]. Analytics Vidhya, 2023 [cit. 2024-04-09]. Dostupné z: https://www.analyticsvidhya.com/blog/2023/06/step-by-step-bert-implementation-guide/.

37. WOLFE, Cameron R. *The Basics of AI-Powered Vector Search* [online]. Substack, 2024 [cit. 2024-04-09]. Dostupné z: https://cameronrwolfe.substack.com/p/the-basics-of-ai-powered-vector-search.

38. LIU, Yinhan; OTT, Myle; GOYAL, Naman; DU, Jingfei; JOSHI, Mandar; CHEN, Danqi; LEVY, Omer; LEWIS, Mike; ZETTLEMOYER, Luke; STOYANOV, Veselin. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR*. 2019, roč. abs/1907.11692. Dostupné z arXiv: 1907.11692.

39. WANG, Liang; YANG, Nan; HUANG, Xiaolong; YANG, Linjun; MAJUMDER, Rangan; WEI, Furu. Multilingual E5 Text Embeddings: A Technical Report. *arXiv preprint arXiv:2402.05672*. 2024.

40. SANH, Victor; DEBUT, Lysandre; CHAUMOND, Julien; WOLF, Thomas. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *ArXiv*. 2019. Dostupné také z: https://arxiv.org/abs/1910.01108.

41. REIMERS, Nils; GUREVYCH, Iryna. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2019. Dostupné také z: https://arxiv.org/abs/1908.10084.

42. KUKREJA, Sanjay; KUMAR, Tarun; BHARATE, Vishal; PUROHIT, Amit; DASGUPTA, Abhijit; GUHA, Debashis. Vector Databases and Vector Embeddings-Review. In: *2023 International Workshop on Artificial Intelligence and Image Processing (IWAIIP)*. 2023, s. 231–236. Dostupné z DOI: 10.1109/IWAIIP58158.2023.10462847.

43. *Documentation* [online]. Qdrant, 2023 [cit. 2024-05-09]. Dostupné z: https://qdrant.tech/documentation/.

44. SCHWABER-COHEN, Roie. *What is a Vector Database & How Does it Work? Use Cases + Examples* [online]. Pinecone, 2023 [cit. 2024-05-09]. Dostupné z: https://www.pinecone.io/learn/vector-database.

45. SKOPAL, Tomáš. *Approximate similarity search* [online]. 2020. [cit. 2024-05-09]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/602570/course/section/92614/BIVWM_lecture10.pdf.

46. TRIPATHI, Rajat. *What is Similarity Search?* [Online]. Pinecone, 2023 [cit. 2024-05-09]. Dostupné z: https://www.pinecone.io/learn/what-is-similarity-search.

47. LEWIS, Patrick; PEREZ, Ethan; PIKTUS, Aleksandra; PETRONI, Fabio; KARPUKHIN, Vladimir; GOYAL, Naman; KÜTTLER, Wen-tau; ROCKTÄSCHEL, Heinrich; LEWIS, Mike; YIH, Tim; RIEDEL, Sebastian; KIELA, Douwe. Retrieval-augmented generation for knowledge-intensive NLP tasks. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. Vancouver, BC, Canada: Curran Associates Inc., 2020. NIPS '20. ISBN 9781713829546.

48. MCELROY, Joe. *Retrieval Augmented Generation: Refine LLM Responses with RAG* [online]. Elastic, 2023 [cit. 2024-05-14]. Dostupné z: https://search-labs.elastic.co/search-labs/blog/retrieval-augmented-generation-rag.

49. CHOI, Nicole. *What is retrieval-augmented generation, and what does it do for generative AI?* [Online]. GitHub, 2024 [cit. 2024-05-14]. Dostupné z: https://github.blog/2024-04

-04-what-is-retrieval-augmented-generation-and-what-does-it-do-for-generative-ai/.

50. NEVES, Miguel Carreira. *What are Quantized LLMs?* [Online]. TensorOps AI, 2023 [cit. 2024-05-14]. Dostupné z: https://www.tensorops.ai/post/what-are-quantized-llms.

51. *Ollama* [online]. GitHub, 2024 [cit. 2024-05-07]. Dostupné z: https://github.com/ollama/ollama.

52. AMAZON.COM. *Amazon Bedrock Kendra Lex Chatbot* [online]. GitHub, 2023 [cit. 2024-05-14]. Dostupné z: https://github.com/aws-samples/amazon-bedrock-kendra-lex-chatbot.

53. SERVICES, Amazon Web. *Amazon Lex Documentation* [online]. Amazon Web Services, [b.r.] [cit. 2024-05-14]. Dostupné z: https://docs.aws.amazon.com/lex/.

54. SERVICES, Amazon Web. *Amazon Kendra Documentation* [online]. Amazon Web Services, [b.r.] [cit. 2024-05-14]. Dostupné z: https://docs.aws.amazon.com/kendra/.

55. SERVICES, Amazon Web. *AWS Bedrock Documentation* [online]. Amazon Web Services, [b.r.] [cit. 2024-05-14]. Dostupné z: https://docs.aws.amazon.com/bedrock/.

56. SERVICES, Amazon Web. *AWS Lambda Documentation* [online]. Amazon Web Services, [b.r.] [cit. 2024-05-14]. Dostupné z: https://docs.aws.amazon.com/lambda/.

57. GOOGLE. *Google Cloud Search Reference* [online]. Google, [b.r.] [cit. 2024-05-14]. Dostupné z: https://developers.google.com/cloud-search/docs/reference.

58. GOOGLE. *Google Cloud AI and ML Documentation* [online]. Google, [b.r.] [cit. 2024-05-14]. Dostupné z: https://cloud.google.com/docs/ai-ml.

59. IBM. *IBM WatsonX Documentation* [online]. IBM, [b.r.] [cit. 2024-05-14]. Dostupné z: https://www.ibm.com/docs/en/watsonx.

60. MICROSOFT. *Azure AI Services Documentation* [online]. Microsoft, [b.r.] [cit. 2024-05-14]. Dostupné z: https://learn.microsoft.com/en-us/azure/ai-services/.

61. WEAVIATE. *Verba: The Golden RAGtriever* [online]. 2023. [cit. 2024-05-14]. Dostupné z: https://github.com/weaviate/Verba.

62. AI, Zylon. *PrivateGPT* [online]. GitHub, 2023 [cit. 2024-05-14]. Dostupné z: https://github.com/zylon-ai/private-gpt.

63. DOCGPT. *DocGPT: AI-Powered Document Management* [online]. 2023. [cit. 2024-05-14]. Dostupné z: https://docgpt.io.

64. MLEJNEK, Jiří. *Analýza a sběr požadavků* [online]. 2023. [cit. 2024-04-14]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/780099/mod_resource/content/10/03.prednaska.pdf.

65. DEVELOPERS, ONNX Runtime. *ONNX Runtime* [online]. 2021. [cit. 2024-05-07]. Dostupné z: https://onnxruntime.ai.

66. HUANG, Wei; MA, Xudong; QIN, Haotong; ZHENG, Xingyu; LV, Chengtao; CHEN, Hong; LUO, Jie; QI, Xiaojuan; LIU, Xianglong; MAGNO, Michele. How Good Are Low-bit Quantized LLaMA3 Models? An Empirical Study. In: 2024. Dostupné také z: https://api.semanticscholar.org/CorpusID:269293766.

67. DEVELOPERS, Google. *Google Drive API Overview* [online]. Google, [b.r.] [cit. 2024-05-08]. Dostupné z: https://developers.google.com/drive/api/guides/about-sdk.

68. *Spring Boot* [online]. Spring, 2024 [cit. 2024-05-08]. Dostupné z: https://spring.io/projects/spring-boot.

69. *Spring AI Reference* [online]. Spring, 2024 [cit. 2024-05-08]. Dostupné z: https://docs.spring.io/spring-ai/reference.

70. *Overview | Vaadin Docs* [online]. Vaadin, 2024 [cit. 2024-05-08]. Dostupné z: https://vaadin.com/docs/latest/overview.

71. MLEJNEK, Jiří. *Architektonické vzory* [online]. 2023. [cit. 2024-05-06]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/600986/mod_resource/content/5/06.prednaska.pdf.

72. OPENAI. *ChatGPT* [online]. 2022. [cit. 2024-05-16]. Dostupné z: https://www.openai.com/chatgpt.

Obsah přiloženého archivu

- `README.md` ... stručný popis obsahu média
- `gsight-be` ... zdrojové kódy implementace
- `text` ... text práce
  - `wenzejan-thesis.pdf` ... text práce ve formátu PDF
  - `thesis` ... zdrojová forma práce ve formátu LaTeX