VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

RAG SYSTÉM PRACUJÍCÍ SE STRUKTUROVANÝMI DATY

RAG SYSTEM WORKING WITH STRUCTURED DATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MAREK TENORA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ONDŘEJ KLÍMA, Ph.D.**
SUPERVISOR

BRNO 2025

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

164066

Ústav:	Ústav počítačové grafiky a multimédií (UPGM)
Student:	**Tenora Marek**
Program:	Informační technologie
Název:	**Question-answering systém pracující se strukturovanými daty**
Kategorie:	Umělá inteligence
Akademický rok:	2024/25

Zadání:

1. Seznamte se s problematiku question-answering systémů; zaměřte se na přístupy pracující s tabulkovými daty.
2. Navrhněte systém, který uživateli umožní interaktivní dotazování na informace obsažené v dodaných strukturovaných datech.
3. Implementujte navržený systém v podobě webové aplikace s využitím dostupných nástrojů a knihoven.
4. Proveďte experimenty s dodanými daty z oboru historie, vyhodnoťte dosažené výsledky i vlastnosti celého řešení.
5. Prezentujte vytvořené řešení a diskutujte jeho využitelnost v praxi.

Literatura:
Dle doporučení vedoucího.

Při obhajobě semestrální části projektu je požadováno:
Body 1, 2 a částečně 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Klíma Ondřej, Ing., Ph.D.**
Vedoucí ústavu:	Černocký Jan, prof. Dr. Ing.
Datum zadání:	1.11.2024
Termín pro odevzdání:	14.5.2025
Datum schválení:	12.11.2024

Abstrakt

Tato práce popisuje návrh a implementaci prototypu aplikace využívající paradigma Retrieval-Augmented Generation (RAG) pro efektivní práci s relačními SQL databázemi. Systém postavený na LangChain/LangGraph kombinuje relační databázi a vektorový index a obsahuje hybridního agenta, který iterativně využívá sémantické vyhledávání a SQL dotazy, ověřuje výsledky a generuje odpovědi se zdroji. Prototyp byl nasazen jako webová aplikace a manuálně otestován, přičemž prokázal vyšší faktickou přesnost oproti jednozdrojovým přístupům, větší transparentnost díky zobrazení interní logiky agenta a dobrou rozšiřitelnost. Identifikována byla také omezení včetně výpočetních nároků, nestability knihoven a bezpečnostních rizik. Přínosem je propojení teorie s praktickou implementací, otevřený prototyp pro další výzkum a doporučení pro výběr nástrojů při vývoji RAG systémů.

Abstract

This thesis presents the design and implementation of a prototype application leveraging the Retrieval-Augmented Generation (RAG) paradigm for efficient interaction with relational SQL databases. The system, built on LangChain/LangGraph, combines a relational database with a vector index and features a hybrid agent that iteratively applies semantic search and SQL queries, verifies retrieved results, and generates source-backed responses. The prototype was deployed as a web application and manually tested, demonstrating higher factual accuracy compared to single-source approaches, greater transparency through visible agent reasoning, and solid extensibility. The work also identifies limitations, including high computational demands, the instability of rapidly evolving libraries, and security risks related to database access. Its contribution lies in bridging theoretical foundations with practical implementation, offering an open prototype for further research and providing recommendations for selecting tools in the development of RAG based systems.

Klíčová slova

RAG, umělá inteligence, AI, zpracování přirozeného jazyka, text-to-SQL, agentní systémy, SQL databáze, vektorová databáze, LangChain, LangGraph, FastAPI, Vue.js

Keywords

RAG, artificial intelligence, AI, natural language processing, text-to-SQL, agentic systems, SQL database, vector database, LangChain, LangGraph, FastAPI, Vue.js

Citace

TENORA, Marek. *RAG systém pracující se strukturovanými daty.* Brno, 2025. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ondřej Klíma, Ph.D.

RAG systém pracující se strukturovanými daty

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Ondřeje Klímy, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Marek Tenora
13. května 2025

Poděkování

Chtěl bych poděkovat vedoucímu práce – panu Ing. Ondřeji Klímovi, Ph.D., za pomoc, rady, přípravu testovací databáze a časté konzultace.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Vyhledání správných informací ze strukturovaných (i nestrukturovaných) dat je klíčové pro všechna odvětví, která si dokážeme představit. Tento proces je často zdlouhavý, vyžaduje znalosti o struktuře dat a nástroje, které data zobrazí. S nástupem umělé inteligence se nám otvírají nové cesty, jak informace efektivně a srozumitelně doručit i uživateli, který se ve struktuře nevyzná a pouze ví, co potřebuje zjistit.

Architektura Retrieval-Augmented Generation (RAG) kombinuje velké jazykové modely s externím vyhledáváním, čímž umožňuje modelu odpovídat na dotazy na základě konkrétních dat. Samotný velký jazykový model tak nespoléhá pouze na předem naučené znalosti, ale pracuje i s aktuálním kontextem získaným z dotazování na externí datové zdroje.

Tato práce se zaměřuje na návrh a implementaci systému využívající RAG pro práci s daty uloženými v relační SQL databázi. Součástí řešení je i vektorová databáze, která umožňuje obohatit kontext vyhledáváním pomocí sémantické podobnosti. Tento krok zajišťuje přesnější výběr relevantních informací, které se následně použijí při generování odpovědi.

Navržený systém se skládá z několika částí — backendové komponenty, která zajišťuje API rozhraní, autentizaci uživatelů a koordinaci mezi jednotlivými částmi. Na backend je napojen RAG systém jako služba. Poslední částí je frontendová webová aplikace, která uživateli nabízí přívětivé rozhraní pro kladení dotazů a práci s výsledky. Cílem této práce je ukázat praktické využití propojení relačních dat, vektorového vyhledávání a velkých jazykových modelů při práci s informacemi. Celý systém, včetně RAG komponenty, je plně lokalizován do českého jazyka, čímž je zajištěna jeho přístupnost pro uživatele v českém jazykovém prostředí, zejména pro české historiky.

Kapitola 2

RAG systémy

Tato kapitola se zaměřuje na RAG systémy, jejich architekturu, principy fungování, výhody, možnosti implementace a praktické využití.

2.1 Úvod do RAG systémů

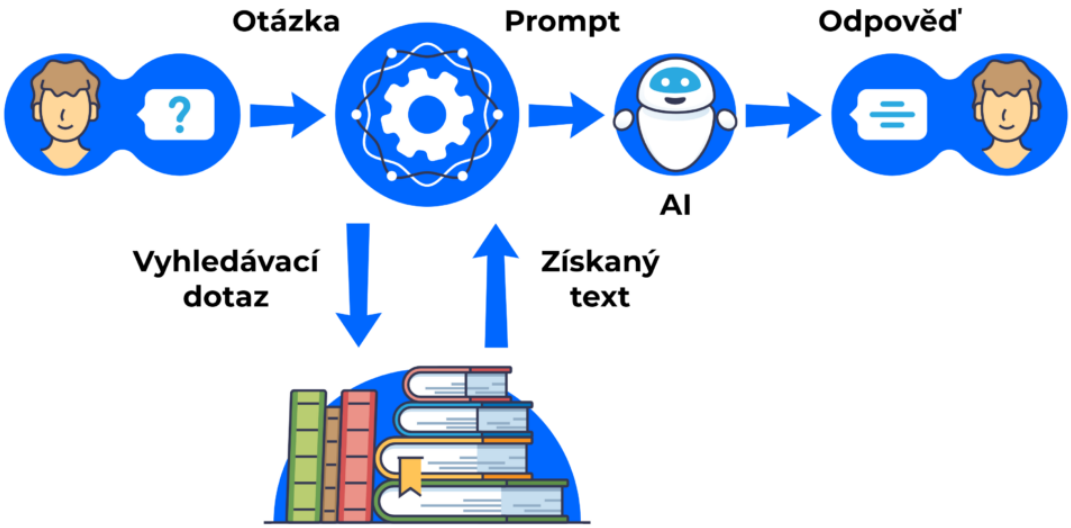
RAG[1] je metoda používaná v oblasti zpracování přirozeného jazyka, která kombinuje generování textu pomocí modelů strojového učení s externím vyhledáváním informací. Tato technika umožňuje generovat odpovědi, které jsou přesnější a aktuálnější, protože model využívá externí databáze nebo zdroje informací pro podporu svého generování. Kombinuje mechanismy pro vyhledávání informací s generativními jazykovými modely, díky tomu je schopen generovat odpovědi na základě externích znalostních bází. Tento přístup zvyšuje přesnost a relevanci generovaných odpovědí, čímž překonává omezení klasických jazykových modelů, které mají všechny informace předem naučeny. Na základě vstupního dotazu se nejprve vyhledají relevantní informace, ověří se jejich správnost v daném kontextu a poté se předají jazykovému modelu, který díky nim odpovídá přesněji[19].

Proč a kdy použít RAG

RAG je vhodné použít pro situace vyžadující aktuální a přesné informace.
Hlavní výhody RAG:

- **Zdroje:** K odpovědím je možné přiřadit konkrétní zdroje ze kterých se informace čerpaly a tím se dá jednoduše ověřit jejich správnost.
- **Snížení halucinací:** pokud se odpověď ve zdrojích nenajde, jazykový model se nebude snažit vymyslet věrohodně znějící nesprávné odpovědi, ale odpoví že informaci nenalezl.
- **Vylepšená přesnost a relevantnost:** Správným načítáním relevantních dokumentů a dat se zajišťuje, že výstup bude přesnější.
- **Škálovatelnost a kapacita pro zpracování velkých znalostních bází**: Systém RAG dokáže efektivně vyhledávat a načítat relevantní informace z obrovských datových sad, což z něj činí škálovatelné řešení vhodné pro aplikace vyžadující rozsáhlý

[1] **Retrieval Augmented Generation (RAG)**: Přeloženo jako "Generování podpořené vyhledáváním".



Obrázek 2.1: Jak funguje RAG [12].

přístup ke znalostem. Díky tomu je možné generovat kontextově rozšířené a přesné odpovědi i při práci s velkým množstvím dat.

- **Flexibilita a přizpůsobení:** Systémy RAG jsou přizpůsobitelné a dají se vyladit pro konkrétní oblasti, což vývojářům umožňuje vytvářet specializované nástroje pro konkrétní odvětví nebo úkoly.
- **Aktualizace znalostí v reálném čase:** Informace jsou čerpány z externích zdrojů, které mohou být aktualizovány v reálném čase
- **Všestrannost a více-modální integrace:** Je možné systém rozšířit tak, aby podporoval více-modální data (text, obrázky, strukturovaná data)[1].

Historie a vývoj RAG systémů

Před vznikem RAG používaly generativní systémy statickou znalostní bázi, která byla obsažena již v tréninkových datech, to značně omezovalo možnost poskytovat aktuální informace. Díky tomu byly zavedeny metody, které umožnily dynamicky získávat data z externích zdrojů. RAG byl poprvé představen v roce 2020 výzkumníky z Meta[13].

S pozdějším vývojem byly zavedeny pokročilé techniky pro efektivní získávání informací, jako je například vektorová databáze se sémantickým vyhledáváním.

V roce 2024 byl představen nový přístup **GraphRAG**, který rozšiřuje tradiční RAG o

využití znalostních grafů. Tato integrace umožňuje modelům propojit různé zdroje informací, provádět složitější úvahy a potencionálně snižovat riziko halucinací. Metoda klade velký důraz na indexaci dat, což je klíčový krok před samotným generováním odpovědi. GraphRAG využívá velké jazykové modely LLM[2] k vytvoření grafového indexu, kde uzly reprezentují entity a hrany vztahy spolu s dodatečnými atributy. Tímto přístupem můžeme dosáhnout efektivnější zodpovídání dotazů vyžadující globální porozumění nad daty (předchozí metody se soustředily hlavně na lokální vyhledání konkrétních informací bez použití globálních znalostí o datech), a to díky hierarchické struktuře, ze které se poté generuje shrnutí pro zodpovězení dotazů[10].

2.2 Velké jazykové modely (LLM)

Velké jazykové modely jsou pokročilé neuronové sítě navržené ke zpracování a generování přirozeného jazyka. Jsou trénovány na velkém množství textu pomocí samo-učení nebo částečného učení s učitelem. Jejich hlavním úkolem je předpovídat následující slovo na základě předchozího textu, to jim umožňuje vytvářet smysluplné věty. Zásadním průlomem v této oblasti se stal **mechanismus pozornosti**, který umožňuje efektivní zpracování velkého množství dat pro učení modelu a vylepšené schopnosti porozumění složitým jazykovým strukturám[8][17].

Základní princip fungování

1. Zadání vstupního řetězce: každý velký jazkový model má svůj kontextový limit udávající maximální délku vstupního řetezce.

2. Tokenizace: zadaný řetězec je rozdělen na malé části na základě jejich významu, těmto částem se říká tokeny. Tokeny nemusí být nutně celé slovo, například slovo "Doorbell"[3] je rozděleno na 2 tokeny "door"a "bell".

3. Vektorizace: každý token je převeden na jeho vektorovou reprezentaci. Cílem je aby vektory podobných slov byly v blízkém prostoru u sebe, díky tomu je možné kategorizovat jejich sémantickou a syntaktickou podobnost. Slova "pán"a "paní"budou tedy blíže než slova "pán"a "sova".

4. **Tranformery** a mechanismus pozornosti:

   - Transformery jsou typ neuronové sítě, která dokáže paralelně zpracovávat vstup a tím je mnohem efektivnější než předchozí typy.
   - Mechanismus pozornosti určuje, které části vstupního textu jsou důležitější než ty ostatní. To umožňuje hledání souvislostí a významů i ve velmi dlouhých větách.

5. Generování výsledku: Po zpracování pozornosti a vztahů mezi tokeny model předpovídá, jaké slovo (token) by mohlo následovat. Tuto předpověď opakuje token za tokenem, až nakonec vytvoří celou odpověď[18][17].

[2] Large Language Models - česky velké jazykové modely
[3] Doorbell - česky zvonek u dveří

Klíčové vlastnosti

Klíčové vlastnosti modelu určují jeho schopnosti, efektivitu a použitelnost v praxi. Níže jsou uvedeny hlavní faktory, které ovlivňují jeho výkon a vhodnost pro konkrétní aplikace:

- **Velikost modelu:** Udává se počtem parametrů, které model obsahuje. Jednotlivé parametry jsou číselné hodnoty, které ovlivňují, jak model zpracovává vstupní data a generuje výstupy.
- **Rozsah trénovacích dat**: Kvalitu modelu velmi ovlivňuje to, na kolika a jak kvalitních datech byl model trénován.
- **Výpočetní náročnost**: Míra potřebných výpočetních zdrojů pro trénink a provoz modelu.

Uplatnění pro RAG systémy

Velké jazykové modely jsou klíčové pro tvorbu RAG systému nejen při generování konečných odpovědí na základě získaných dat, ale mohou být použity i ve fázi získávání dat pro zapojení kritického myšlení, vylepšení dotazů na externí zdroje nebo procesu rozhodování, který zdroj dat použít pro získání nejrelevantnějších informací.

Vhodné modely pro použití v RAG

Použití správného velkého jazykového modelu je klíčové hlavně po optimalizační stránce. Každý model je svým způsobem unikátní a liší se parametry a datovými sadami, na kterých byl naučen. To může v RAG ovlivnit cenu, výsledky a rychlost odpovědí. Pro některé úkony bude dostačující použít menší model (ideálně specializovaný), který je rychlý a má nižší náklady. Naopak pro úkony, kde je zapotřebí více logického zapojení, je vhodné použít model silnější.
Příklady některých modelů a jejich klíčových vlastností:

1. **Uvažující modely** Modely umělé inteligence schopné adaptivního rozhodování a řešení komplexních problémů. Liší se od ostatních modelů tím, že před odpovědí nějakou dobu "přemýšlí".
    - GPT-o1 (OpenAI): První uvažující model, vysoká cena.
    - R1 (DeepSeek): Čínská verze modelu od OpenAI, která zaujala hlavně nízkými náklady.
2. **Velké univerzální modely**
   Tyto modely jsou trénovány na rozsáhlých a různorodých datech, což jim umožňuje zvládat širokou škálu úkolů. Jsou ideální pro generování složitých odpovědí.
    - GPT-4o (OpenAI): Vhodný pro obecné dotazy i složité úkoly díky vysoké přesnosti a kontextovému porozumění.
    - Gemini 1.5 Pro (Google): Model s největším kontextovým limitem (2 miliony tokenů)
    - Grok-2 (xAI): Vhodný pro pokročilé kódování a zpracování textu. Model je optimalizovaný pro rychlé odpovědi

3. **Středně velké modely**
   Modely v této kategorii nabízejí kompromis mezi výkonem a náklady. Vhodné jsou pro úkoly kde není nutná extrémní univerzálnost.

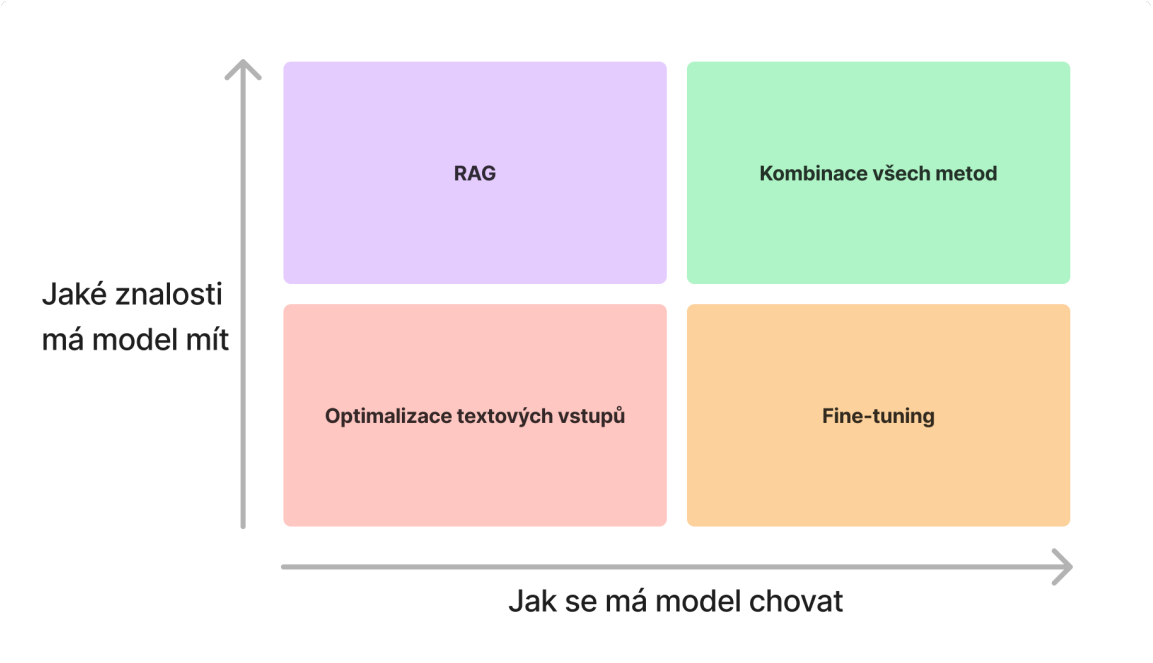
   - Llama 3.3 (Meta): Nabízí velké kontextové okno (128 000 tokenů) a nízké provozní náklady.
   - Claude 3.5 Sonnet (Anthropic): Exceluje v analýze vizuálních dat a složitých textových úlohách.
   - GPT-4o-mini (OpenAI): Cenově efektivní varianta s nízkou latencí. Ideální pro aplikace vyžadující rychlé odpovědi při nižších nákladech.

4. **Malé modely**
   Nízká náročnost na zdroje a rychlá odezva. Vhodné pro jednoduché rozhodovací procesy nebo rozpoznání vzorů v textu.

   - BERT (Google): Zaměřený na pochopení kontextu slov v textu. Ideální pro vyhledávání nebo sentimentální analýzu
   - Phi-3 Mini (Microsoft): Poskytuje vysokou přesnost a při velmi nízkých nákladech. Vhodný pro sumarizaci dokumentů.

2.3 RAG nebo Fine-tuning

Zatímco **Fine-tuning**[4] modifikuje parametry modelu pro specializované úkoly (např. formátování textu nebo gramatická korekce), **RAG** zachovává obecné schopnosti LLM a rozšiřuje je o dynamický přístup k externím zdrojům.



Obrázek 2.2: Metody pro přizpůsobení modelu.

[4]**Fine-tuning** (doladění) je proces přizpůsobení předtrénovaného modelu na konkrétní úkol pomocí dalšího cíleného trénování.

Klíčové aspekty pro volbu přístupu

- **Aktualizace znalostí**
  - Fine-tuning vyžaduje přeučení pro aktualizaci znalostní báze
  - RAG stačí aktualizovat externí databázi bez zásahu do modelu
- **Cena**
  - Trénování velkých modelů může být velmi nákladné
  - Provoz RAG je relativně úsporný
- **Implementace**
  - Pro implementaci fine-tuningu je zapotřebí datová sada a hostování doučeného modelu
  - Pro RAG je potřeba napsat program, který chod RAG řídí, LLM se obvykle napojuje na externí API (OpenAI API, Mistrail API), nebo na vlastní hostovaný model.
- **Kontextová relevance**
  - Fine-tuning exceluje v úkolech s pevnými vzory (např. automatizace formulářů)
  - RAG lépe zvládá komplexní dotazy v kategorii na kterou je zaměřen

Limitační faktory

1. Pro historický výzkum je zásadní **schopnost citovat primární zdroje** - zatímco RAG explicitně uvádí původ informací, Fine-tuning integruje znalosti přímo do modelu bez možnosti zpětného dohledání
2. Při práci s kronikami obsahujícími protichůdné záznamy:
   - Fine-tuning by vedl ke generování "průměrných"odpovědí
   - RAG umožňuje identifikovat a prezentovat různé verze událostí

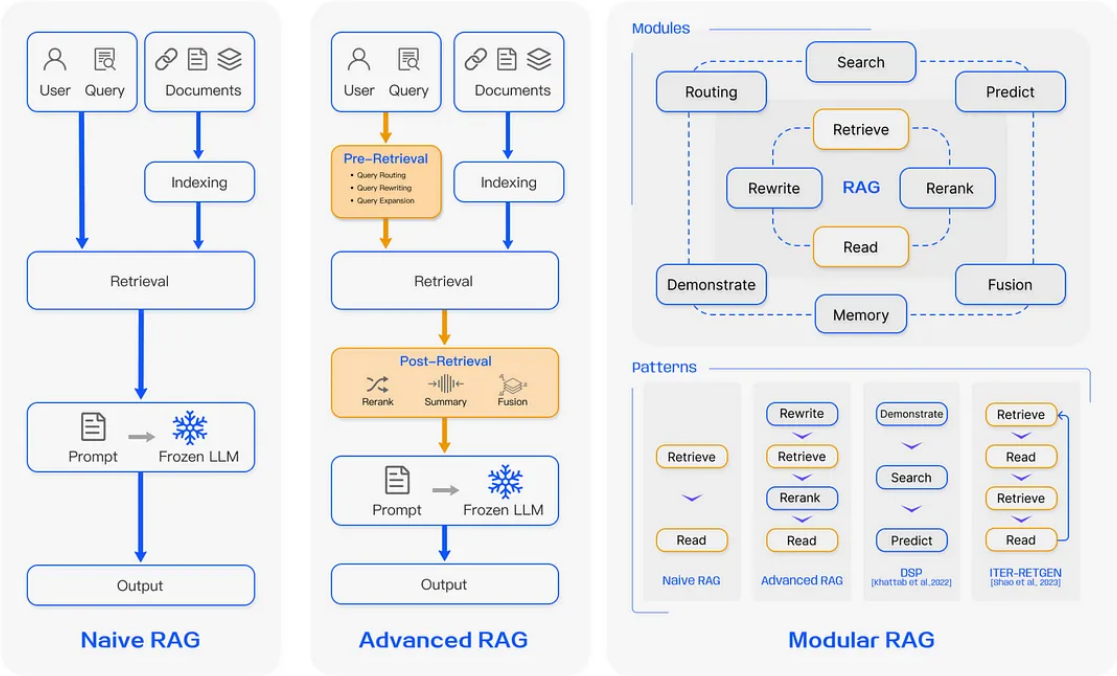
Praktická aplikace v historickém výzkumu

Analýza srovnávající RAG a finetune přístupy[9] ukazuje, že RAG systémy:

- Často **dosahují nižší hodnoty perplexity**[5], oproti modelům bez externích znalostí
- Na základě lidského hodnocení dosahují lepších výsledků, hlavně v úkolech, kde je zapotřebí **faktická přesnost**.
- Jsou vhodné pro systémy, ve kterých je potřeba častá aktualizace znalostní databázi.

Tato analýza potvrzuje, že pro práci s historickými prameny představuje RAG lepší řešení. **Hybridní přístup** (RAG + částečný fine-tuning) by mohl být relevantní v případě potřeby specifické interpretace historického jazyka použitého v kronice.

[5] **Perplexita** je způsob, jakým měříme, jak "zmatený"je model umělé inteligence, když se snaží předpovědět další slovo nebo frázi v textu. Čím nižší je hodnota perplexity, tím lépe se model v textu orientuje a chápe ho.



Obrázek 2.3: Porovnání mezi paradigmaty RAG systémů [11].

2.4 Rozdělení RAG paradigmat

Implementace RAG systému v praxi probíhá prostřednictvím různých paradigmat, která se odlišují jak složitostí architektury, tak přístupem k integraci znalostních bází. Správná volba paradigmatu je zásadní, protože přímo ovlivňuje přesnost výsledného systému, rychlost odpovídání na dotazy a celkovou kvalitu výstupních dat. Na obrázku 2.3 je graf popisující jednotlivá paradigmata.

Naivní RAG

První a nejjednodušší RAG paradigma, získalo popularitu krátce po globálním rozšíření ChatGPT. Přístup následuje tradiční proces zahrnující 3 fáze: indexování, získávání dat a generování odpovědí.

- **Indexování:** Začíná čištěním a extrakcí surových dat různých formátu do obyčejného textu. Text je poté segmentován na menší části (chunks[6]). Segmenty jsou následně zakódovány do vektorové reprezentace pomocí **embeddovacích modelů**[7] a uloženy do vektorových databází, které obsahují původní hodnoty segmentů a jejich vektorovou reprezentaci. Krok je klíčový jelikož umožňuje sémantické vyhledávání za pomocí podobnosti vektorů.

[6] V oblasti umělé inteligence se termín **"chunks"** často používá k popisu menších, logicky oddělených částí dat nebo informací, které jsou zpracovávány modelem.

[7] **Embedding modely** převádějí slova, fráze nebo jiná data do číselných vektorů v nižším rozměrném prostoru, které zachycují jejich význam a vztahy, což umožňuje strojům lépe porozumět a pracovat s těmito daty.

- **Získávání dat:** Po získání uživatelského dotazu, je použit stejný **embedovací model** k převodu dotazu na vektorovou reprezentaci. Vektor dotazu je poté porovnán s ostatními vektory segmentů uložených v databázi. Systém vybere **top K**[8] nejpodobnějších segmentů, které přidává k originálnímu dotazu jako rozšířený kontext.

- **Generování odpovědí:** Dotaz obohacený o rozšířený kontext je předán jako jeden **prompt**[9] do jazykové modelu. Tento prompt může být dále obohacen o historii konverzace. Jazykový model na základě formátu promptu, dotazu, rozšířeného kontextu a historii konverzace generuje odpověď. Model může čerpat informace i ze svých předem naučených znalostí nebo se omezit pouze na informace poskytnuté v promptu, to záleží na specifikaci programátora promptu.

Zhodnocení: Naivní RAG nabízí velmi jednoduchou implementaci, která může být v mnoha případech dostačující, ale potýká se s několika klíčovými problémy:

- **Problémy při získávání dat:** Často dochází k problémům s přesností a úplností získaných dat, to se děje kvůli výběru nesprávných nebo nerelevantních segmentů, a to vede ke generování nepřesných odpovědí.

- **Problém s generováním:** Při generování může dojít k **"halucinacím"**, které způsobí nekonzistenci oproti získanému kontextu. Může dále způsobit irelevanci, toxicitu nebo zkreslení informací, což zhoršuje kvalitu a spolehlivost systému.

- **Problémy s augmentací**[10] **informací:** Integrace získaných dat s konkrétním dotazem může být obtížná. Tento proces může vést k nejednotným odpovědím. Rovněž může dojít k redundanci, kdy se podobné informace opakují z více zdrojů, to způsobí opakování odpovědí.

Díky těmto komplikacím nemusí vždy jedno získávání dat na základě dotazu stačit k získaní kvalitního kontextu, který je vhodný pro generování správné odpovědi[11].

Rozšířený RAG

Cílem rozšířeného RAGu bylo vyřešit problémy, se kterými se potýkal naivní RAG. Soustředí se na vylepšení kvality získávání dat, přináší nové strategie **pre-retrieval**[11] a **post-retrieval**.[12] Rozšířený RAG používá nové metody pro vylepšení indexace:

- **Metoda posuvného okna:** Postupné procházení textu v pevně daných krocích, které pomáhá zachytit důležité informace v dlouhých textech, aniž by se přehlédli klíčové detaily.

- **Jemnozrnná segmentace:** Oproti klasickému segmentování rozděluje text na menší částí (věty, fráze) a tím umožňuje přesnější analýzu a zachycení kontextu při indexování. Vyžaduje větší výpočetní náklady.

[8] **Top K** označuje výběr K nejlepších nebo nejpravděpodobnějších výsledků (např. slov, odpovědí nebo možností) z většího seznamu na základě určité metriky, jako je pravděpodobnost nebo skóre.
[9] **Prompt** je textový vstup nebo instrukce, která řídí chování AI modelu, aby generoval požadovanou odpověď.
[10] **Augmentace** je umělé rozšíření nebo úprava existujících dat (např. obrázků, textu).
[11] **pre-retrieval** - před-vyhledání, krok provedený před získáním dat
[12] **post-retrieval** - po-vyhledání, krok provedený po získání dat

- **Metadata:** Přidání doplňujících informací k vektorovým záznamům pro lepší organizaci.

Pre-retrieval

Soustředí se na optimalizaci indexovací struktury a původního dotazu. Cíl je optimalizace indexování pro vylepšení kvality získaného obsahu. K dosáhnutí se používají tyto strategie: rozdělení dotazu na více pod-dotazů, optimalizace indexové struktury a přidání atributů. Správný pre-retrieval by měl tedy upravit původní dotaz tak, aby měl zřetelnější záměr a lépe fungoval při vyhledávání dat. Mezi často využívané metody patří **přepisování dotazu**, **transformace dotazu** a **rozšíření dotazu**.

Post-retrieval

Po získání relevantních informací je klíčové jejich efektivní zařazení do dotazu. Hlavní používané metody: **přehodnocení segmentů a komprese kontextu**. Přehodnocování získaných segmentů za účelem přemístění nejdůležitějších segmentů na okraje dotazu se ukázalo jako klíčové. Pokud jazykovému modelu jednoduše vložíme všechny získané segmenty, může dojít k informačnímu přetížení což vede k odvrácení od důležitých detailů. Proto je zároveň důležitá selekce opravdu relevantních segmentů[11].

Modulární RAG

Využívá různé moduly a techniky rozšířeného RAGu

Modulární RAG představuje evoluci oproti předchozím paradigmatům díky své flexibilní architektuře založené na samostatných modulech. Tento přístup umožňuje kombinovat specializované komponenty pro vylepšení kvality vyhledávání, zpracování dotazů i generování odpovědí. Modulární RAG využívá techniky z Rozšířeného RAGu i zcela nové moduly a vzory interakce, které zvyšují adaptabilitu systému pro komplexní úlohy.
Příklady modulů:

- **Vyhledávací modul:** Kromě klasického vyhledávání ve vektorové databázi integruje data z externích zdrojů (vyhledávače, tabulky, znalostní grafy) pomocí LLM-generovaného kódu a dotazovacích jazyků.
- **Paměťový modul:** Vytváří "nekonečnou paměťovou bázi"uchovávající předchozí dotazy a odpovědi. LLM iterativně využívá tuto paměť k upřesňování a ladění odpovědí.
- **Fúze** Řeší limity tradičního vyhledávání pomocí paralelního zpracování. Generuje rozšířené dotazy (z původního dotazu) a ty poté zaráz vykonává. Kombinuje výsledky z různých perspektiv pomocí inteligentního přehodnocování a optimalizace hodnocení.
- **Směrování (Routing):** Dynamicky rozhoduje o dalším kroku pro uživatelský dotaz, zda provést sumarizaci, vyhledat v konkrétní databázi, sloučit informační toky nebo generovat finální odpověď.

Modulární RAG překonává své předchůdce díky flexibilitě a škálovatelnosti. Implementace je však náročnější, je třeba správně nakonfigurovat jednotlivé moduly a komunikaci mezi nimi. Vyžaduje hlubší porozumění jak jednotlivých komponent, tak synergiím mezi nimi[11].

2.5 Vektorové vyhledávání informací

Představuje moderní přístup získávání relevantních dat, který je založen na porovnávání vektorů v n-dimenzionálním prostoru. To nám umožňuje zachytit sémantické vztahy mezi daty a provádět vyhledávání na základě podobnosti vektorů, což přináší výrazně lepší výsledky oproti tradičním metodám založeným na klíčových slovech. Pro RAG systémy hraje vektorové vyhledávání klíčovou roli při procesu získávání relevantních dokumentů, které jsou použity pro obohacení odpovědi.

Embedding modely

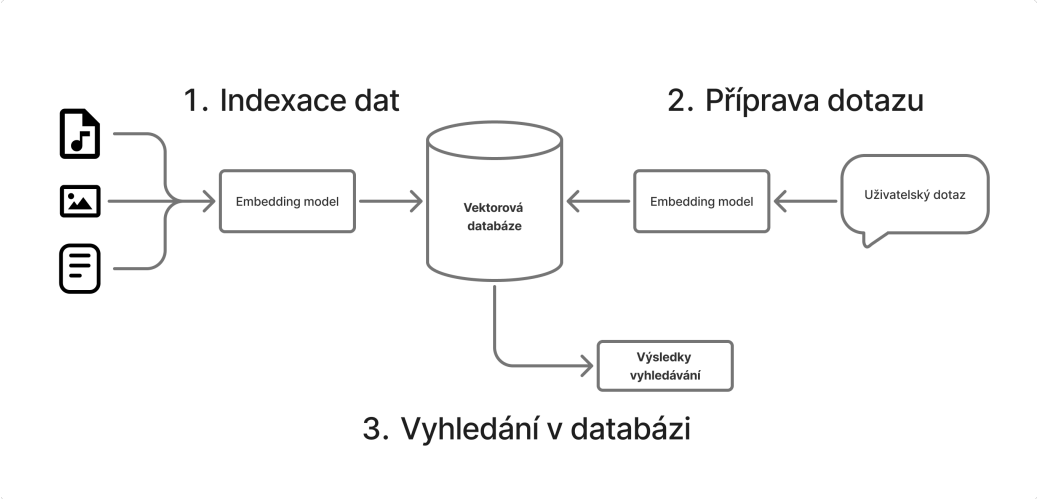
Embedding modely jsou základním nástrojem pro převod nestrukturovaných dat (text, obrázky, zvuk) do vektorové reprezentace, která zachycuje sémantické a kontextové vztahy mezi jednotlivými prvky dat. Modely umožňují pracovat s různorodými datovými formáty díky transformaci do jednotného vektorového prostoru.
Vlastnosti:

- **Sémantická podobnost:** Embedding zajišťuje, že podobné dokumenty[13], reprezentované vektory, jsou v prostoru blízko sebe, zatímco odlišné jsou od sebe dál. To umožňuje efektivně vyhledávat podobné dokumenty na základě jejich významu.
- **Dimenzionalita:** Vektory mají předem nastavený pevný počet dimenzí (nejnovější modely od OpenAI mají 1536 pro menší a 3072 pro větší verzi modelu[7]), kde každá dimenze zachycuje určitý aspekt významu. Vyšší počet dimenzí umožňuje zachytit jemnější nuance, ale zároveň zvyšuje výpočetní náročnost při zpracování. Například vektor slova "student"bude v prostoru blízko slovu "studentka"díky podobnému sémantickému významu.
- **Kontext:** Novější modely podporují generování kontextově závislé vektorové reprezentace, což znamená že význam jednotlivých částí dokumentu jsou ovlivněny jeho okolím. Například slovo "instituce"bude mít odlišnou reprezentaci v kontextech "školní instituce"a "státní instituce".

Postup vektorového vyhledávání:

1. **Vektorizace dat:** Pomocí embedding modelu jsou vstupní dokumenty převedeny na vektorovou reprezentaci.
2. **Indexace:** Vygenerované vektory jsou uloženy spolu s původními dokumenty ve vektorové databázi.
3. **Vektorizace dotazu:** Uživatelský dotaz je převeden do vektoru stejným embedding modelem.
4. **Vyhledávání podobnosti:** Pomocí metody kosinové podobnosti jsou nalezeny vektory v databázi, které jsou nejblíže vektoru dotazu. Existuje více způsobů vyhledávání, ale metoda kosinové podobnosti je často preferována, protože je nezávislá na velikosti vektorů a zaměřuje se na jejich směr, což umožňuje efektivně porovnávat význam dokumentů s různou délkou[14].
5. **Seřazení výsledků:** Výsledky hledání jsou seřazeny od nejbližších (nejpodobnějších) po nejvzdálenější.

[13] **dokument:** nositel informace o různém datovém formátu a velikosti (např.: věta, obrázek)



Obrázek 2.4: Proces vektorového vyhledávání.

Vzorec pro výpočet kosinové podobnosti vektorů, kde A, B jsou vektory[2]:

$$\frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

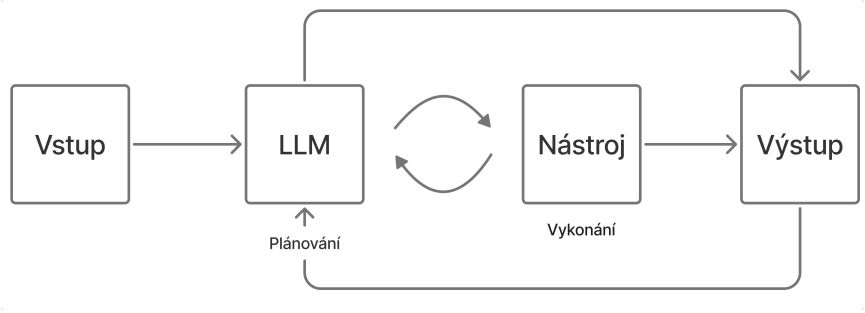
Bilinguální podpora Při použití embedding modelu s podporou více jazyků je možné vyhledávat bilinguálně bez potřeby jakéhokoliv překladu dotazu.

Existující modely:

- **OpenAI text-embedding-3**: Model nabízí lepší výkon a sémantické porozumění. Vyniká ve specializovaných oblastech a při práci s mezijazyčným obsahem. Je navržen jako efektivní a škálovatelné řešení. K dispozici jsou 2 verze: "small"(1536 dimenzí) a "large"(3072 dimenzí)[7][15].
- **Cohere Embed v3**: Je významným modelem pro mezijazyčné schopnosti, podporující více než 100 jazyků s výjimečným výkonem. Zachycuje složité jazykové vzorce napříč jazyky a je efektivní i ve specializovaných úlohách. Model je optimalizován pro produkční prostředí[15].
- **Google Universal Sentence Encoder (USE)**: Toto je všestranné řešení se dvěma variantami - jedna optimalizovaná pro přesnost a druhá pro rychlost zpracování. Má robustní výkon na kratších segmentech textu a silné mezijazyčné schopnosti, podporující 16 jazyků[15].

2.6 Agenti, agentské systémy a nástroje

Chytré systémy AI agentů se charakterizují schopností provádět akce, které postupně po čase vedou k dosažení cíle, aniž by postup, pořadí a způsob provedení akce byly předem definovány. Agenta si můžeme představit jako chytrého pomocníka, který pro svého uživatele plní zadané úkoly. Díky **schopnosti rozhodovat se iterativně za chodu** o dalším postupu je agent schopný dosahovat lepších výsledků, než při použití pouze čistého LLM, kde jsou přesně dané kroky k dosažení cíle stanoveny při položení dotazu. Jelikož se však



Obrázek 2.5: Diagram systému agenta.

agent rozhoduje o postupu sám, **zvyšuje se razantně riziko halucinací a nepředvídatelného chování**[16].

Agentičnost

Stupeň toho, do jaké míry je systém schopný adaptace a dosáhnutí komplexního cíle v komplexním prostředí s omezeným přímým dohledem.

Agentičnost se dále rozděluje do několika podsekcí:

- **Složitost cíle:** Jak složité by bylo pro člověka splnit úkol zadaný agentovi a v jakém rozsahu je schopný agent tohoto cíle dosáhnout? Mezi parametry vyhodnocení patří **spolehlivost, rychlost a bezpečnost**.
- **Složitost prostředí:** Jak složité je prostředí, ve kterém má agent dosáhnout cíle. V jakých oborech oborech agent pracuje, zda se jedná o krátkodobý úkol nebo úkol vyžadující delší časový horizont a jaké externí nástroje má agent k dispozici.

  Příklad: Systém který je schopný odborně hrát jakoukoliv deskovou hru má vyšší složitost prostředí než umělá inteligenci která umí hrát pouze šachy, jelikož dokáže uspět v mnohem větším rozsahu prostředí.
- **Schopnost se adaptovat:** Jak dobře se dokáže systém adaptovat a reagovat na neočekávané okolnosti.
- **Nezávislé provádění akcí:** Do jaké míry dokáže systém spolehlivě provádět akce a dosahovat cíle bez nutnosti intervence člověka.

  Příklad: Auto s 3. úrovní autonomního řízení, které zvládne samostatně operovat na silnici za každé situace má vyšší stupeň nezávislého provádění akcí než tradiční auto ovládané člověk[16].

Nástroje

Některé LLM modely podporují používání nástrojů. Nástroje jsou rozšiřující mechanismy, umožňující agentům přístup k externím zdrojům informací, jako jsou databáze, API rozhraní a další systémy mimo samotný jazykový model. Díky nástrojům může agent vykonávat úkony, které by samotný model bez přístupu k aktuálním datům a funkcím nezvládl.

Nástroje jsou agentům předávány jako funkce se vstupními parametry a popisem funkcionality.

Příklady nástrojů:

- **Webový vyhledávač**
- **Nástroje pro práci s databází**
- **API rozhraní**

2.7 Knihovny a metody implementace

Pro implementaci RAG systémů existuje více různých knihoven, většina z nich je však kvůli velmi rychlému vývoji nestabilní nebo neposkytuje kvalitní dokumentaci. Vybrány byly knihovny z ekosystému LangChain[5]. LangChain obsahuje 3 hlavní knihovny, které pokrývají většinu aspektů vývoje RAGu, komponenty a jejich napojení, vytvoření stavových grafů a monitorování a evaluace.

LangChain

LangChain je open-source framework navržený pro vývoj aplikací využívajících velké jazykové modely. Poskytuje jednotné rozhraní pro práci s LLM, vektorovými databázemi, nástroji pro parsování dat, retrievery[14], pamětí a agenty. Primárním cílem je usnadnit sestavování složitých systémů prostřednictvím znovupoužitelných komponent a integrací[4].

LangGraph

LangGraph je nástroj pro řízení toků založený na stavových grafech, který umožňuje vývojářům vytvářet agentní systémy s kontrolou toku a stavu. Je určen pro případy, kdy je třeba mít přesnější kontrolu nad rozhodovací logikou agenta, včetně návratu k předchozím stavům, paměti nebo spolupráce člověka a agenta[6].

LangSmith

LangSmith je nástroj pro ladění, sledování a vyhodnocování LLM aplikací. Nabízí nástroje pro trasování běhu, logování vstupů a výstupů modelu, verzování promptů a experimentální evaluace. Cílem nástroje je vytvoření prostředí, kde je možné efektivně testovat a optimalizovat chování AI aplikací před jejich nasazením[3].

2.8 Přehled existujících řešení

NotebookLM

NotebookLM je webová aplikace od Googlu, do které můžete nahrát své dokumenty a poté o nich můžete s umělou inteligencí komunikovat. Aplikace umožňuje vygenerovat podcast, který dokumenty shrnuje. Velmi dobrý příklad použití RAG, momentálně nepodporuje SQL databáze jako zdroj.

[14] **Retriever**: V rámci ekosystému LangChain je pojem definován jako rozhraní, které na základě neformulovaného dotazu vrací relevantní dokumenty.

AskYourDatabase

AskYourDatabase je možné použít jako desktopovou nebo jako webovou aplikaci, která se připojuje k SQL databázi. Při použití desktopové verze zůstávají přístupy k databázi na straně klienta. Systém postupně buduje znalostní bázi na základě používání a tím se postupně vylepšuje. Od řešení prezentovaného v této práci se aplikace liší v použití znalostní vektorové databáze, v této práci je vektorová databáze inicializována ihned po napojení a aktuálně nepodporuje učení na základě uživatelských interakcí.

Kapitola 3

Návrh

3.1 Analýza požadavků a specifikace systému

Tato část shrnuje základní požadavky na systém, a to z uživatelského a technického hlediska. Definují se zde hlavní cíle, které musí systém plnit.

Funkční požadavky

V této podsekci jsou popsány klíčové funkční požadavky, a to ve formě uživatelských příběhů (user stories). Uživatelské příběhy popisují očekávání, které jednotlivé role od systému očekávají a jaké potřeby má aplikace plnit.

Jako	**chci**	**abych**
uživatel	položit dotaz v přirozeném jazyce	rychle získal relevantní informace z databáze
uživatel	vidět, z jakých zdrojů aplikace čerpala odpověď	mohl ověřit její správnost
uživatel	aby umělá inteligence generovala přesné informace	na ni mohl při práci spolehnout
uživatel	aby umělá inteligence generovala přehledné a srozumitelné odpovědi	snadno a rychle našel klíčové informace bez zbytečného hledání
uživatel	vidět historii svých dotazů a odpovědí	se k nim mohl vrátit později
uživatel	mít účet zabezpečený (např. heslem)	zabránil ostatním v přístupu k mým dotazům
uživatel	aby bylo používání aplikace cenově efektivní	mohl službu udržitelně využívat
administrátor	spravovat uživatelské účty	zajistil bezpečný provoz systému
administrátor	nastavovat modely a API klíče	mohl spravovat napojení systému na externí služby
administrátor	spravovat napojení SQL databáze	mohl agentovi zajistit přístup ke správným datům
administrátor	indexovat SQL databázi do vektorové podoby	mohl agentovi umožnit vektorové vyhledávání
administrátor	uzavřenou aplikaci dostupnou jen schváleným uživatelům	zamezil neoprávněnému přístupu k datům
administrátor	mít sadu funkcí pro kontrolu aplikace	je mohl spouštět přímo z příkazového řádku
systém (agent)	zpracovat uživatelský dotaz, získat relevantní dokumenty a vygenerovat odpověď s využitím RAG	poskytl přesné a srozumitelné výsledky

Tabulka 3.1: Přehled funkčních požadavků systému ve formě uživatelských příběhů.

Nefunkční požadavky

Nefunkční požadavky určují obecné vlastnosti systému, a to například v oblastech, jako je výkon, bezpečnost, styl psaní kódu a údržba. Vzhledem k omezenému času nebyly všechny běžné nefunkční požadavky implementovány.
Při dalším vývoji pro produkční nasazení by se mělo zaměřit na tyto požadavky:

- Bezpečná autentizace a autorizace podle průmyslových standardů (např.: dvoufaktorové ověření)

- Optimalizace výkonu a nasazení na produkčním serveru
- Monitorování a logování
- Vytvoření CI/CD pro nasazování nových verzí

Tyto aspekty byly při implementaci identifikovány jako důležité, ale nebyly prioritizovány kvůli zaměření na funkčnost a návrh systému.

Omezení a předpoklady

V této části jsou shrnuty hlavní technologická omezení, která ovlivnila návrh a realizaci projektu. Zároveň definuje klíčové předpoklady pro správné fungování aplikace. Jelikož se stále jedná o prototyp a nikoli plně produkční verzi, některé aspekty byly méně prioritní na úkor důležitějších částí, jako je například optimalizace architektury RAG a funkční propojení jednotlivých komponent systému.

Technologická omezení

Práce byla realizována s následujícími technologickými omezeními:

- **Experimentální knihovny**: Použité knihovny jako Langchain a Langgraph jsou nové a rychle vyvíjené knihovny, to může způsobovat nestabilitu, problémy s výběrem správné verze a nedostatečnou dokumentací.
- **Výpočetní zdroje**: Kvůli omezeným zdrojům nebylo možné otestovat provoz při velké zátěži.
- **Bezpečnost**: Díky velkému rozsahu práce nebylo implementování rozšířených bezpečnostních postupů prioritou.
- **Nedokonalost modelů**: Na umělou inteligenci by se nemělo brát vždy spolehnutí. Bezpečnostní test prokázal, že je možné pomocí škodlivých dotazů měnit chování a instrukce modelu. Z tohoto důvodu se velmi doporučuje napojovat databázi pomocí specializovaného účtu s právy pouze pro čtení.

Předpoklady pro funkčnost aplikace

Pro správné fungování aplikace je potřeba splnit tyto body:

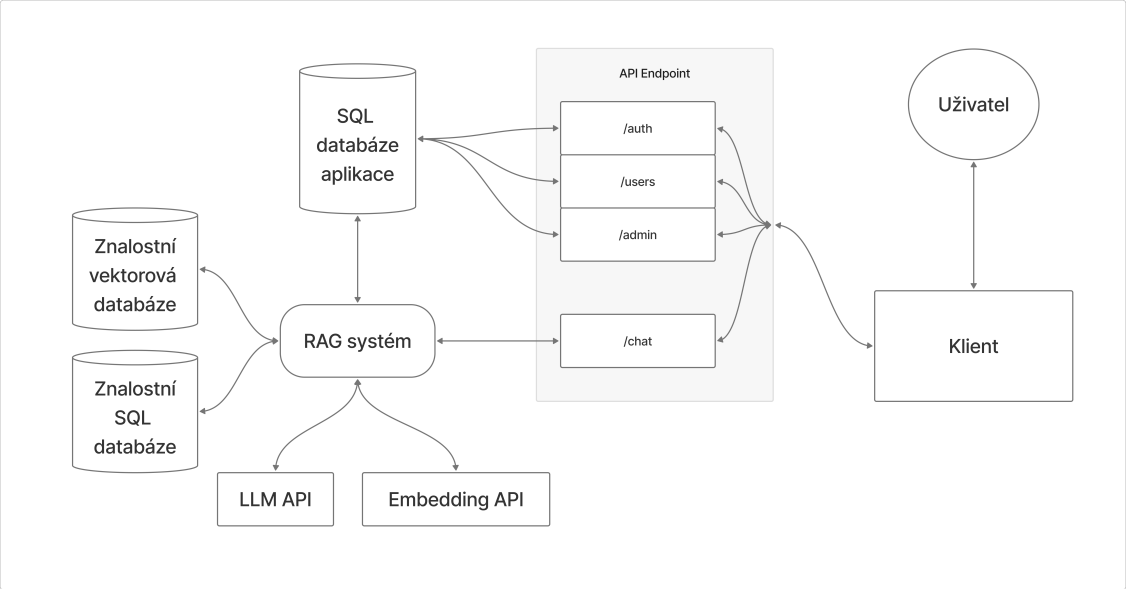
- **Backend**: Server musí mít nainstalovaný Docker a správce musí správně nakonfigurovat proměnné prostředí. Nastavení CORS pravidel musí povolovat přístup z adresy frontend aplikace.
- **LLM a embedding**: Aplikace musí mít nastavený model a API klíč, a to jak pro jazykový model a embedding model. Napojení bylo testováno na poskytovatelích OpenAI API a OpenRouter API.
- **Přístup do SQL databáze**: V aplikaci musí být nastaven přístup k SQL databázi, ze které systém hledá informace. Systém momentálně podporuje MySQL databáze.
- **Indexace SQL databáze**: Pro správné fungování vektorového hledání je důležité indexovat napojenou databázi
- **Endpoint**: Frontend aplikace musí mít nastavenou adresu backend API.

3.2 Architektura systému

Celkový návrh architektury

Aplikace je rozdělena na dvě hlavní části: klientskou část (frontend) napsanou ve Vue.js a serverovou část(backend) postavenou na Python FastAPI, které mezi sebou komunikují. Autentizace uživatele je implementována pomocí JWT tokenu[1].

RAG systém poté funguje jako samostatný modul, napsaný s využitím frameworků LangChain a LangGraph, který je integrován do backendu a jeho funkce jsou přístupné prostřednictvím třídy `RAG`.



Obrázek 3.1: Vysoko-úrovňový diagram architektury aplikace.

Systém pracuje celkem se třemi typy databází, z nichž každá plní odlišnou roli(viz. obrázek 3.1):

- **SQL databáze aplikace**: Slouží pro ukládání dat aplikace, správu uživatelů, konfigurace RAG a historii konverzací.
- **Externí znalostní SQL databáze**: Napojena administrátorem, databázi využívá RAG jako znalostní bázi.
- **Znalostní Vektor databáze**: Vytvořena pomocí indexace **Externí znalostní SQL databáze**. Umožňuje vyhledávat sémanticky.

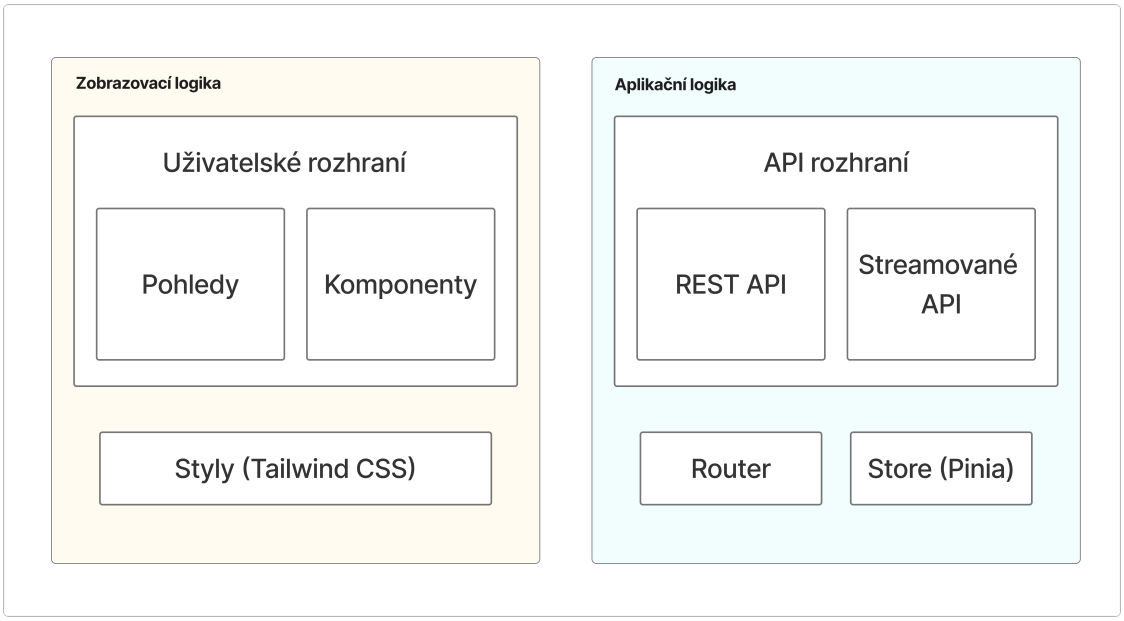
Frontend architektura

Frontendová architektura je navržena jako komponentová webová aplikace využívající framework Vue.js. Aplikace je rozdělena do několika pohledů, využívá komponenty pro přehlednější strukturu kódu a obsahuje servisní vrstvu pro komunikaci s backendem a správu

[1] **JWT (JSON Web Token)** je standard pro bezpečné předávání autentizačních informací mezi stranami.

aplikačního stavu. Hlavním pohledem aplikace je `ChatView` ve kterém uživatel komunikuje s botem. Aplikace není kvůli omezenému času optimalizována pro mobilní zařízení. Návrh frontendu kladl důraz na jednoduchost, přehlednost, modulárnost a škálovatelnost.

Pohledy jsou odděleny pomocí routeru[2], který řídí směrování mezi stránkami (např.: nepřihlášený uživatel bude při pokusu přejít na stránku chatu vždy přesměrován na přihlášení).



Obrázek 3.2: Vysoko-úrovňový diagram architektury frontendu.

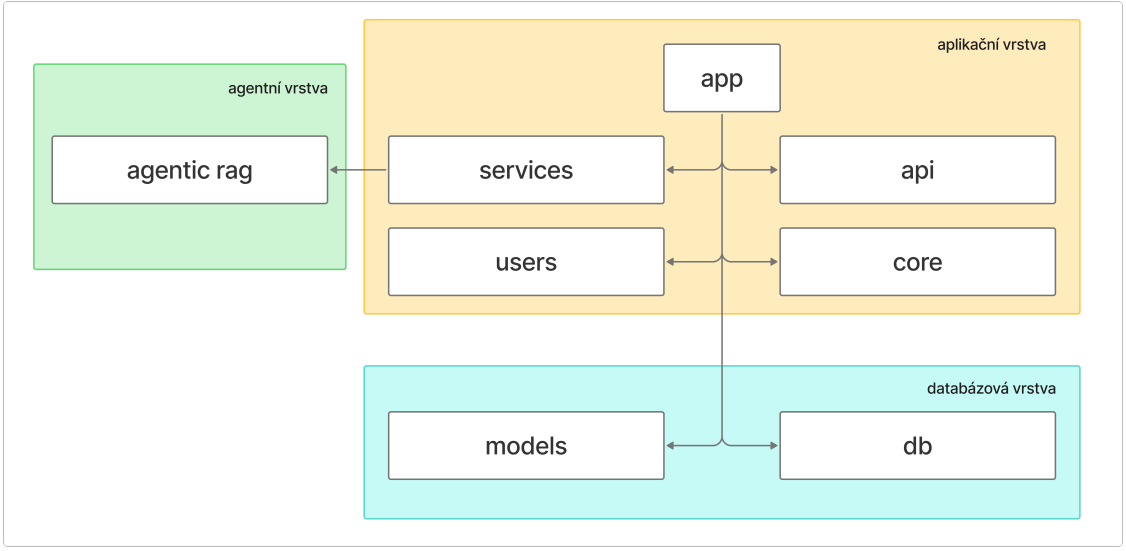
[2]**Router** je nástroj ve webových aplikacích, který řídí navigaci mezi jednotlivými komponentami na základě URL adresy. Díky němu mohou být na různých stránkách aplikace zobrazeny odlišné prvky podle toho, na jakém odkazu se uživatel nachází.}

Pohled	Popis
HomeView	Úvodní stránka aplikace, kde uživatel vidí základní informace a může přejít k přihlášení nebo registraci.
LoginView	Přihlašovací stránka pro uživatele s formulářem pro zadání přihlašovacích údajů.
RegisterView	Stránka pro registraci nových uživatelů s příslušným registračním formulářem.
ChatView	Hlavní chatovací rozhraní, kde uživatel komunikuje s RAG agentem.
AdminView	Administrátorské rozhraní pro správu uživatelů a konfigurace RAG.
AboutView	Stránka s informacemi o aplikaci, jejím účelu a autoru.

Tabulka 3.2: Přehled hlavních pohledů frontend aplikace.

Backend architektura

Backend je postaven jako kombinace jednoduché serverové aplikace postavené na frameworku FastAPI a pokročilého RAG systému využívající LangGraph. Architektura je navržená modulárně, aby bylo možné aplikaci snadně rozšiřovat a udržovat.



Obrázek 3.3: Vysoko-úrovňový diagram architektury backendu.

Celý systém je rozdělen do tří hlavních vrstev:

- **Aplikační vrstva**: Poskytuje REST API pro frontend a rozhraní pro příkazový řádek. Zajišťuje směrování požadavků a použití služeb.

- **Agentní vrstva**: Koordinuje jednotlivé komponenty RAG systému (např.: LLM modely, databáze, nástroje).
- **Databázová vrstva**

Použité technologie

Backendová část systému využívá několik technologií, které byly vybrány na základě jejich vhodnosti pro daný účel a efektivitu implementace v rámci projektu.
Parametry pro výběr ideálních řešení byly:

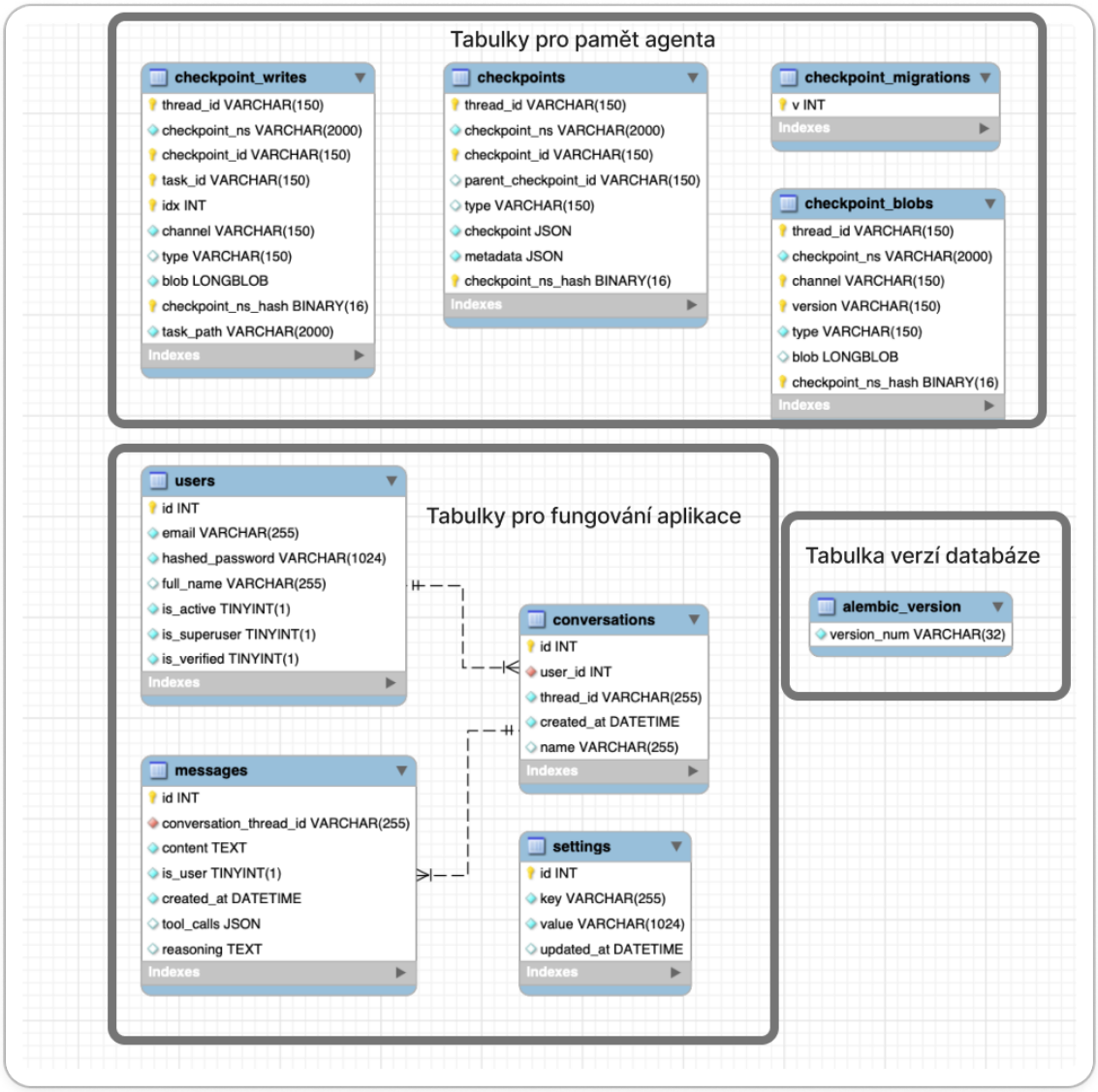
- **Složitost implementace**: Zda je možné technologií za omezený čas správně implementovat.
- **Stabilita**: Preferovány byly knihovny a frameworky, které jsou udržované a jejichž chování je předvídatelné.
- **Osobní preference**: Použití technologií, se kterými má programátor již nějaké zkušenosti je vždy přínosné a zkracuje výrazně čas potřebný pro vývoj.
- **Aktuálnost**: Moderní technologie nabízejí oproti zastaralým řešením jednodušší způsoby implementace běžných funkcionalit (např. správa uživatelů, reaktivita) a lépe odpovídají současným vývojovým standardům.

Název	**Popis**
Python	Programovací jazyk použitý jako hlavní prostředek pro vývoj backendové části aplikace. Nabízí rozsáhlý ekosystém knihoven a frameworků vhodných pro práci s webovými službami i umělou inteligencí.
FastAPI	Moderní webový framework pro Python, který umožňuje rychlý vývoj API s vysokým výkonem. Podporuje automatickou dokumentaci a validaci dat na základě typových anotací.
FastAPI Users	Rozšiřující knihovna pro FastAPI, která zjednodušuje správu uživatelů, autentizaci a autorizaci, včetně práce s JWT tokeny a správou hesel.
SQLAlchemy	Knihovna pro Python, která zajišťuje pohodlnou práci s relačními databázemi prostřednictvím objektově orientovaného přístupu.
Docker	Nástroj pro kontejnerizaci, který umožňuje balit aplikaci a její závislosti do přenositelného a konzistentního prostředí, což usnadňuje nasazení a správu.
Alembic	Nástroj pro správu databázových migrací v Python projektech používajících SQLAlchemy. Umožňuje postupnou úpravu schématu databáze bez ztráty dat.

Tabulka 3.3: Použité backend technologie.

Databázový návrh

Na obrázku 3.4 je zobrazen ER diagram databáze použité v aplikaci. Návrh databáze je rozdělen do několika logických bloků podle jejich funkce:



Obrázek 3.4: ER diagram aplikační databáze.

Tabulky lze rozdělit do tří hlavních kategorií:

- **Tabulky pro fungování aplikace**:
  - `users`: Uchovává informace o uživatelích systému, včetně e-mailu, hesla (uloženého v hashované podobě), jména a příznaků (aktivní, admin, ověřený účet). Funkcionalita ověřování účtů však zatím nebyla implementována a příznak `is_verified` je tedy připraven do budoucna.

  - `conversations`: Reprezentuje jednotlivé konverzace uživatelů s agentem. Každá konverzace má přiřazeného uživatele, identifikátor vlákna (`thread_id`) a čas vytvoření.
  - `messages`: Obsahuje jednotlivé zprávy v rámci konverzace, včetně samotného textového obsahu, informace, zda šlo o vstup od uživatele nebo odpověď systému, případně informací o volání nástrojů a doplňkových výstupech modelu (`reasoning`).
  - `settings`: Uchovává aplikační nastavení ve formě klíč-hodnota, například konfigurační údaje nebo parametry ovlivňující chování systému.

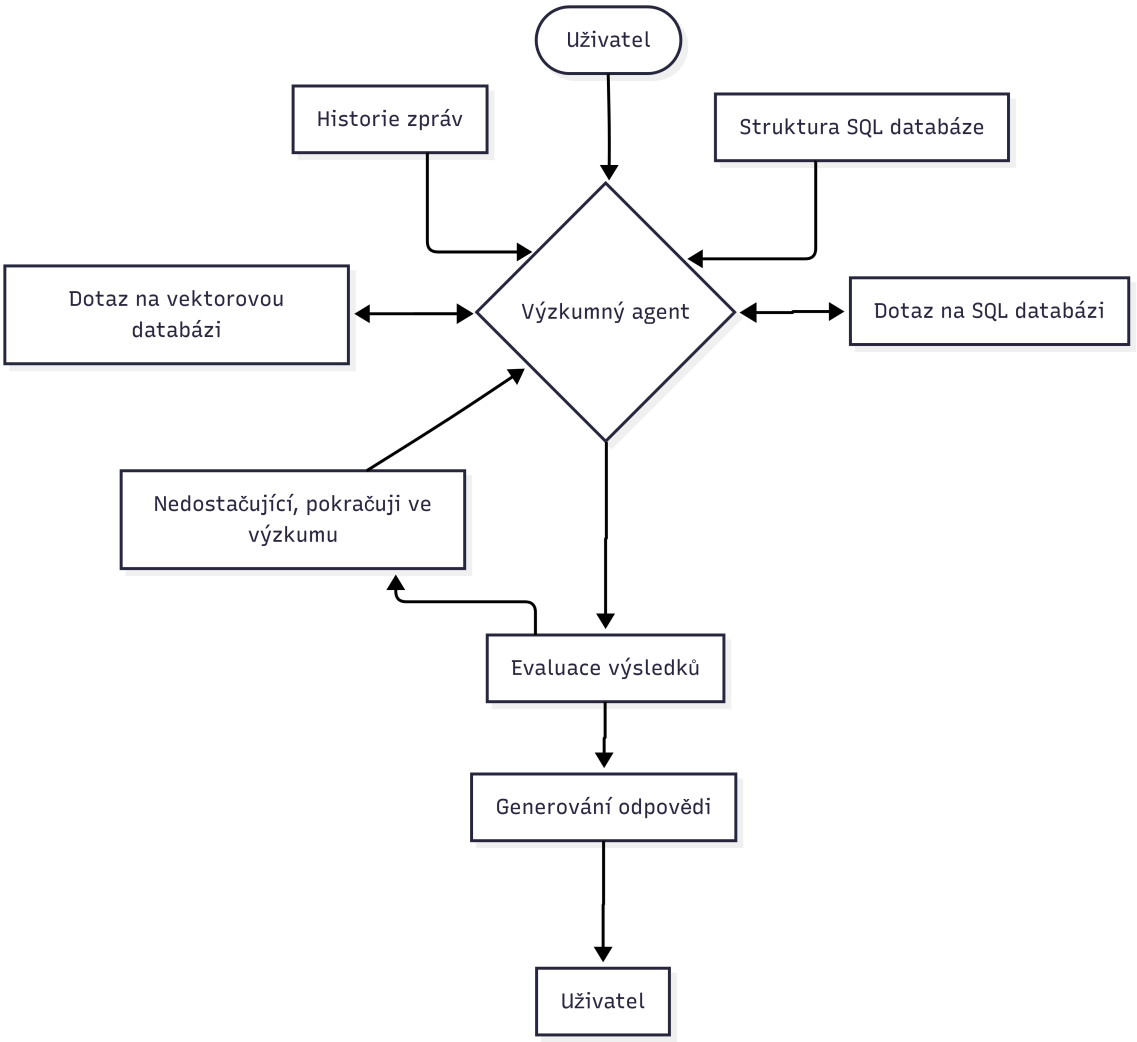
- **Tabulky pro paměť agenta**: Tyto tabulky slouží jako úložiště pro paměťový mechanismus Langchainu, který uchovává kontext předchozí komunikace. Jejich obsah spravuje přímo framework a zahrnuje různé pomocné datové struktury potřebné pro správné fungování agenta.

- **Tabulka verzí databáze**:

  - `alembic_version`: Sleduje aktuální verzi schématu databáze spravovaného nástrojem Alembic.

3.3 Návrh RAG agenta

Pro návrh RAG agenta jsem se rozhodl využít pokročilý přístup postavený na architektuře výzkumného agenta, který dokáže iterativně vyhledávat relevantní informace ze dvou zdrojů: z vektorové databáze (pro sémantické hledání) a ze strukturované SQL databáze (pro přímé dotazování).



Obrázek 3.5: Architektura RAG agenta.

Hlavní komponentou je **výzkumný agent** který je zodpovědný za získání správných dat, které jsou předány do fáze generování odpovědi. Agent má k dispozici 2 hlavní nástroje, dotazování na SQL databázi a dotazování na vektorovou databázi. Agentu jsou zároveň poskytnuty i pomocné nástroje pro práci s SQL, které může využít pro získání informací ohledně databáze (struktura tabulky, seznam tabulek v databázi, kontrola správnosti SQL dotazu). Jelikož se jedná o agenta, tak nelze staticky nastavit, jaké nástroje má kdy a kolikrát použít, proto je zpracování každého dotazu unikátní a nelze jej přesně predikovat. Agent má nastavenu vyladěnou vstupní instrukci s pokyny, které mu určují roli, formát odpovědi, jak a k čemu má nástroje používat a hlavní úkoly, které by měl vždy plnit. Jedním z funkčních požadavků je, aby byly odpovědi dobře strukturované, z toho důvodu byl tento úkol oddělen od hlavního výzkumného agenta a provádí se samostatně po konci výzkumu na základě zjistěných informací a uživatelském dotazu.

Pravidla pro výzkum:

- Informace získané z vektorové databáze jsou označeny jako **neověřené** a nemohou být poskytnuty uživateli bez dalšího ověření v SQL databázi.
- Informace získané z SQL databáze jsou označeny jako **ověřené**, tyto data můžou být použity k vytvoření odpovědi pro uživatele.
- Pokud se nepovede nalézt odpověď na dotaz, výzkum by měl pokračovat dalším iterativním dotazováním, dokud nejsou získány relevantní a ověřené informace.
- Pokud je uživatelský dotaz nejednoznačný, agent by měl aktivně pokládat upřesňující otázky, aby získal kontext potřebný pro správný výzkum.
- Agent nikdy nesmí používat nástroje pro úpravu databáze (např. INSERT, UPDATE, DELETE), má právo data pouze číst.
- Agent by měl při návrhu SQL dotazů využívat pokročilé konstrukce jako jsou JOINy, CTEs, agregace a třídění, aby maximalizoval kvalitu a úplnost získaných dat.
- Při zjišťování trendů nebo četnosti musí agent použít vhodné agregační funkce (COUNT, MAX) místo ručního vyhodnocování výsledků.
- Odpověď s výsledky výzkumu by vždy měla obsahovat strukturovaný výstup s položkami: ověřené informace, citace, neověřené informace, historie SQL dotazů a interní poznámky. Tento strukturovaný výstup pomáhá generovat komplexní odpovědi.

Při návrhu systému bylo důležité zohlednit nejen přesnost odpovědí a schopnosti agenta, ale také faktory jako rychlost odezvy, náklady na jeden dotaz a efektivní využití dostupných nástrojů. Architektura byla záměrně postavena tak, aby využívala jednoho centralizovaného výzkumného agenta, místo paralelního nasazení více specializovaných agentů. Přestože by bylo technicky možné rozdělit úlohy mezi 10-20 různých agentů (například samostatného agenta pro vektorové hledání, pro SQL dotazy, pro slučování výsledků), tento přístup by výrazně prodloužil dobu odpovědi kvůli nutné koordinaci a předávání výsledků mezi agenty. Navíc by hrozilo, že během této mezi-krokové komunikace dojde ke ztrátě kontextu původního dotazu, což by mohlo vést k halucinacím nebo nesprávným závěrům.

Na druhou stranu přílišná komplexnost jednoho agenta může vést k jeho přetížení, kdy se snaží řešit příliš mnoho úkolů naráz, což se může projevit snížením kvality odpovědí nebo chybným používáním nástrojů. Z tohoto důvodu byla agentovi odebrána úloha generování finální odpovědi, která probíhá až v samostatné fázi po dokončení výzkumu.

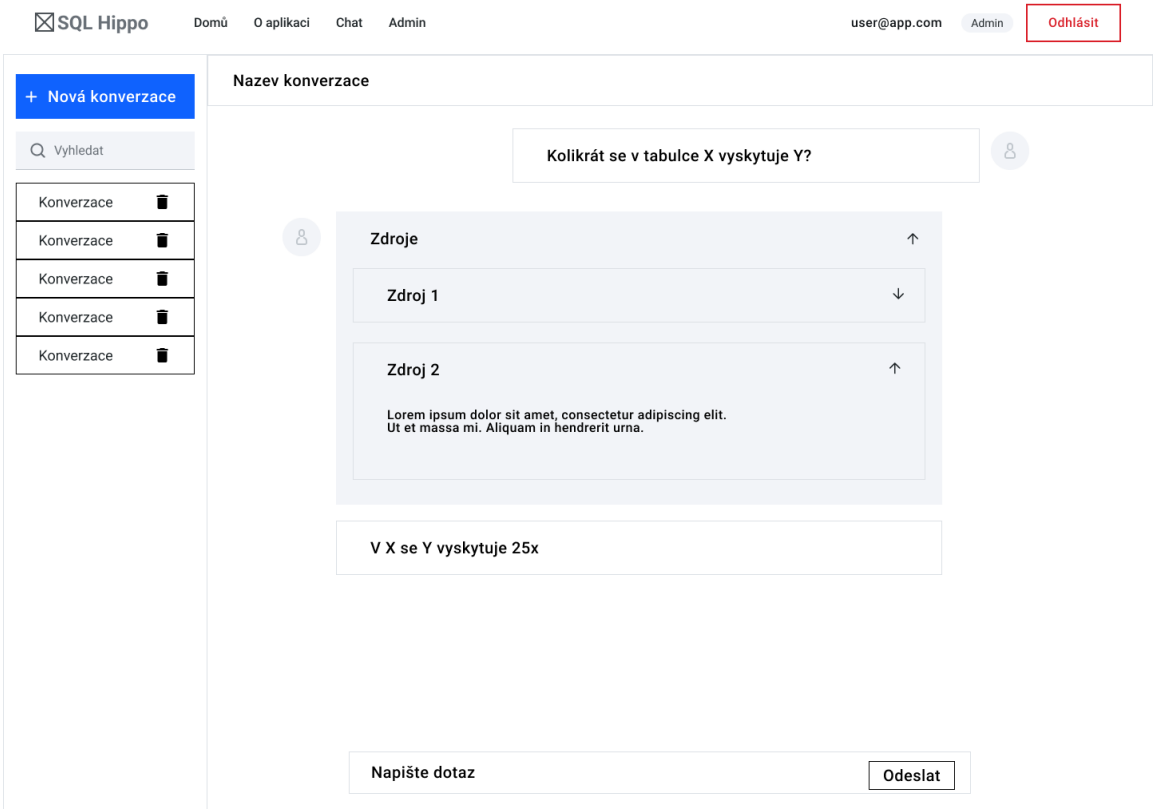
Během vývoje jsem nejprve testoval jednodušší přístupy, například LLM bez agentní architektury, řešení postavená jen na SQL nebo jen na vektorovém vyhledávání. Tyto varianty však nedosahovaly požadované kvality odpovědí a ukázalo se, že klíčovým přínosem je schopnost agenta "rozmýšlet se"nad dotazem a iterativně kombinovat různé strategie. Hybridní přístup, který propojuje práci s SQL i sémantickým vyhledáváním, se tak stal hlavním pilířem návrhu. Tento koncept vyvažuje pokrytí různých typů dotazů, rychlost odezvy a konzistenci výsledků a ukázal se jako optimální řešení, jehož podrobnější srovnání s dalšími variantami bude popsáno v následujících kapitolách.

3.4 Návrh uživatelského rozhraní

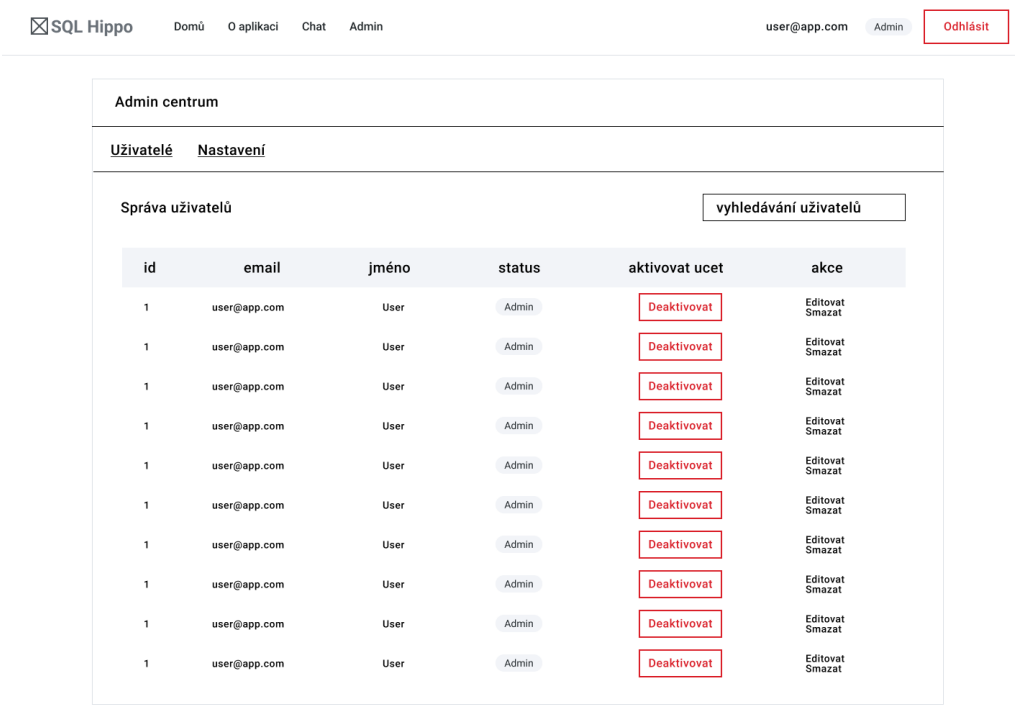
Při návrhu uživatelského rozhraní byl kladen důraz především na vytvoření moderního a přehledného vzhledu chatovacího okna, zatímco ostatní pohledy, které nebyly pro hlavní účel práce zásadní, byly navrženy co nejjednodušeji.

Wireframy a prototypy

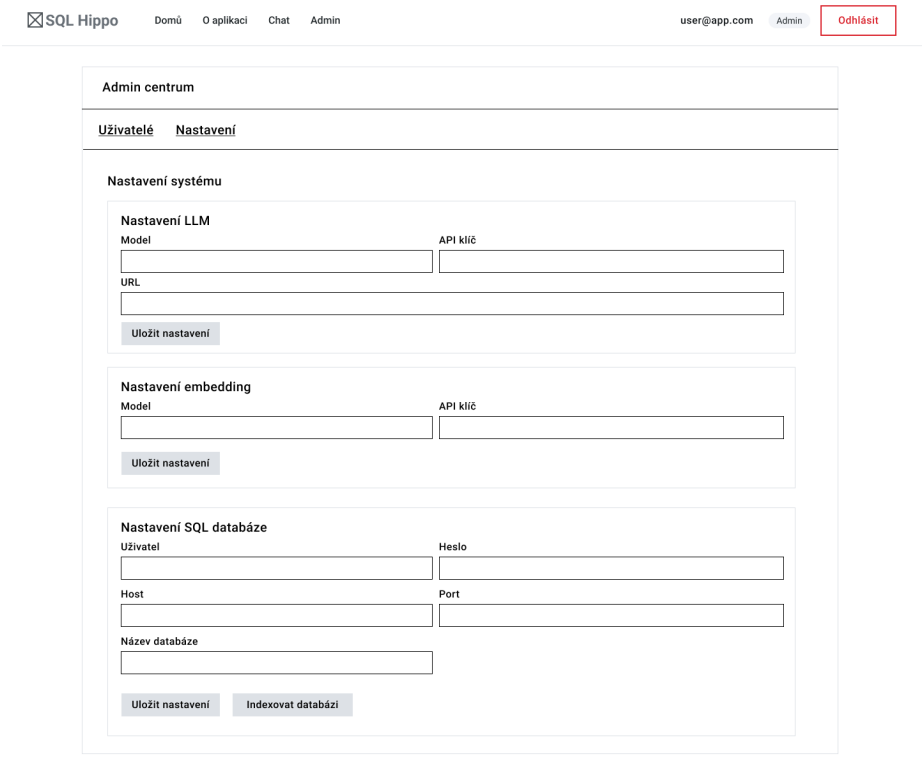
Pro návrh byly připraveny wireframy hlavně pro složitější části aplikace, jako je chat, správa uživatelů a systémové nastavení, kde bylo potřeba předem promyslet funkce a rozložení. Pro jednodušší pohledy (domů, o aplikaci, přihlášení/registrace) nebyly wireframy vytvořeny. Wireframy sloužily primárně jako podklad pro konečný návrh aplikace. Wireframe pohledu chatu byl inspirován existujícími nástroji pro práci s AI (ChatGPT, Perplexity, Google Gemini), díky čemuž je používání intuitivní, jelikož se uživatel již s podobnými aplikacemi nejspíš setkal.



Obrázek 3.6: Wireframe pohledu chatu.



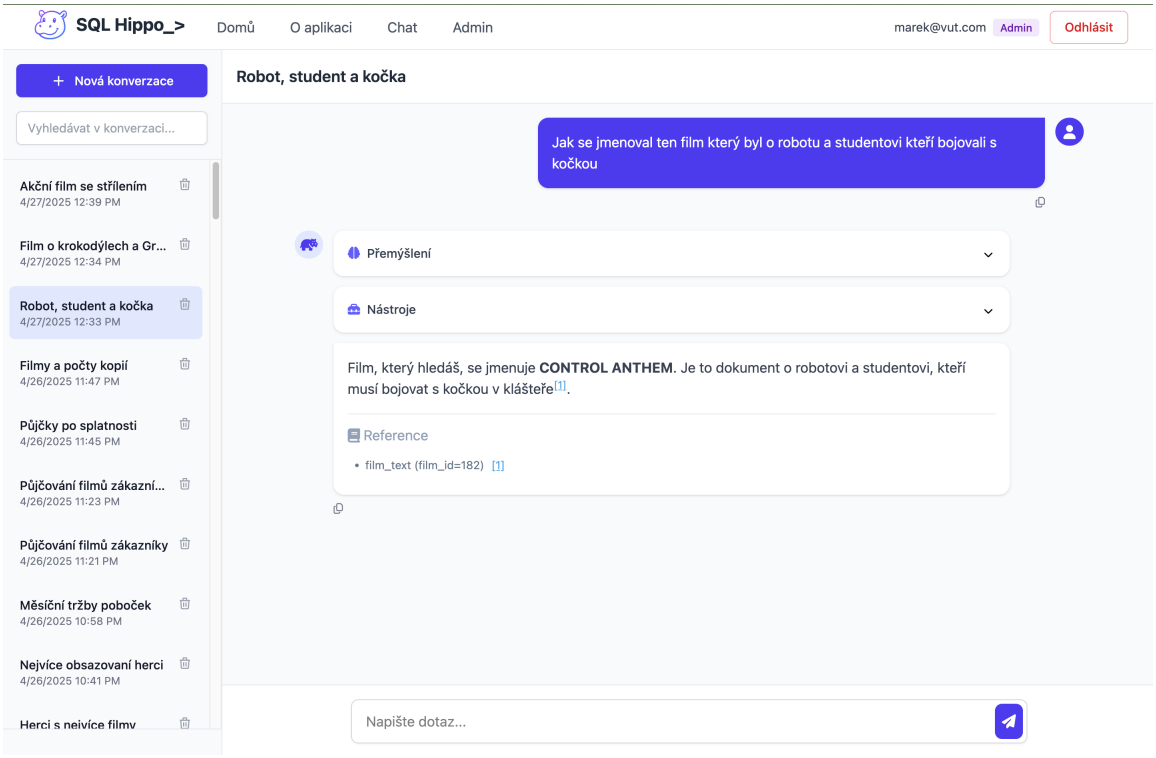
Obrázek 3.7: Wireframe pohledu správy uživatelů.



Obrázek 3.8: Wireframe pohledu nastavení systému.

Návrh chatovacího rozhraní

Při návrhu chatovacího rozhraní bylo cílem co nejlépe reprezentovat výstupy z RAG systému a poskytnout uživateli příjemné moderní prostředí.

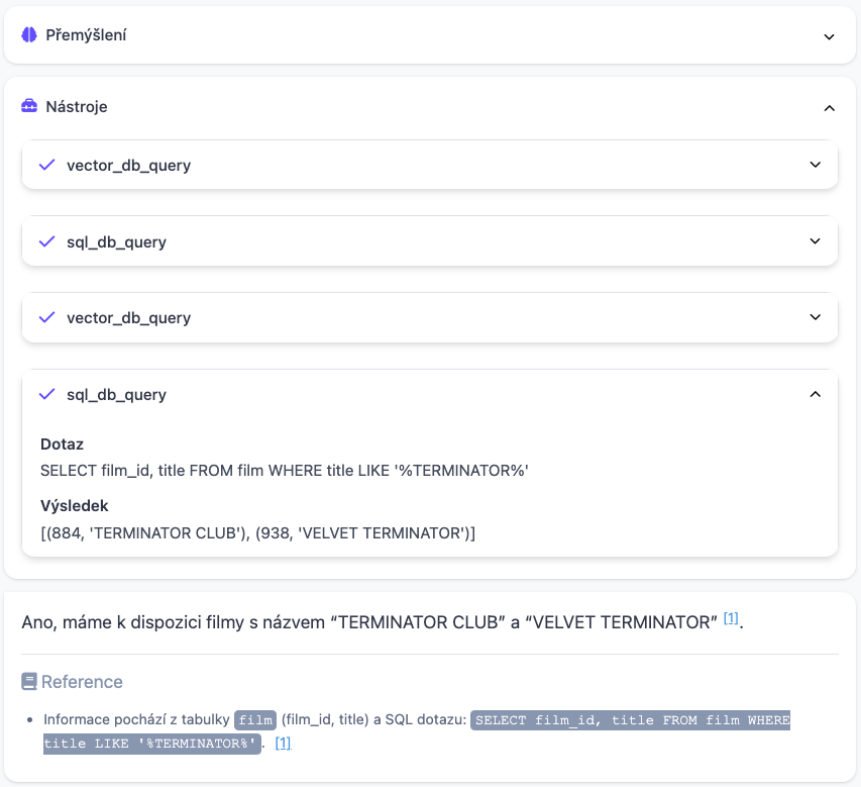


Obrázek 3.9: Ukázka chatovacího rozhraní.

Návrh zpráv v chatu

Zprávy v chatu jsou rozděleny do dvou hlavních typů: **uživatelský dotaz** a **odpověď systému**. Dotazy uživatele jsou barevně odlišeny a jasně odděleny, což usnadňuje orientaci i při delší konverzaci.

Odpovědi systému jsou složené z několika částí, které společně vytvářejí přehledný výstup. Kromě hlavní textové odpovědi, která přímo reaguje na dotaz, se zobrazují také sekce **Přemýšlení** a **Nástroje**. Sekce "Přemýšlení"dává uživateli možnost nahlédnout do interního postupu agenta, tedy jak nad dotazem uvažoval a jakým způsobem výzkum probíhal. Sekce "Nástroje"ukazuje, jaké konkrétní nástroje byly použity během zpracování dotazu.



Obrázek 3.10: Detailní pohled na odpověď systému.

Obě tyto části jsou navrženy jako rozbalitelné bloky, aby nenarušovaly čitelnost hlavní odpovědi, ale zároveň byly dostupné uživatelům. Na závěr odpovědi je uvedena sekce **Reference**, která obsahuje citace a odkazy na zdroje použitých dat, což zajišťuje transparentnost a možnost ověření.

Designové prvky a styly

Obrázek 3.11: Logo aplikace.

Aplikace byla pojmenována **SQL Hippo** a doplněna jednoduchým logem hrocha, které ji vizuálně odlišuje a dává jí zapamatovatelnou identitu.

Použité barevné schéma je založeno na **světlém designu** s fialovo-modrou barevnou paletou, která zajišťuje moderní vzhled a dobrou čitelnost. Pro typografii byl použit výchozí font Tailwind CSS.

Celý vizuální styl aplikace staví na frameworku **Tailwind CSS**, který pomohl vytvořit jednotný, responzivní a moderní design, usnadňující jak vývoj, tak údržbu rozhraní.

Animace

Aby byl uživatelský zážitek co nejpříjemnější, byly použity v pohledu chatu animace. Implementovány byly různé načítací animace, například točící kolečko indikující fázi "přemýšlení"a práci s nástroji, animace se třemi tečkami signalizující přípravu odpovědi, a také mechanika streamování odpovědi, kdy se výsledná odpověď zobrazovala postupně po jednotlivých malých částech hned, jakmile dorazily ze serveru. Díky tomu uživatel nemusel čekat na kompletní načtení celé odpovědi a mohl začít číst okamžitě, což snižuje vnímanou čekací dobu a přispívá k plynulejšímu a přehlednějšímu zážitku.

Kapitola 4

Implementace

4.1 Implementace frontendu

Tato sekce popisuje praktickou implementaci klientské aplikace.

Vytvoření základní aplikace

Pro vytvoření nové Vue.js aplikace se použil příkaz `npm create vue@latest`, který vytvoří základní adresářovou strukturu a nastaví základní konfiguraci na základě dotazů položených uživateli při vytváření. Poté byly nainstalovány a nakonfigurovány pomocné knihovny (TailwindCSS, VueAutoAnimate).

Struktura klientské aplikace

Tabulka popisuje funkce jednotlivých složek.

Adresář	Popis
assets	Grafiky a obrázky použité v aplikaci
components	Komponenty využité v pohledech
helpers	Obsahuje pomocnou funkci pro debuggování API požadavků
lib	Pomocné funkce pro komponenty Shadcn
router	Správa směrování mezi stránkami
services	Komunikace API
stores	Správa globálního aplikačního stavu
views	Jednotlivé pohledy (stránky)

Tabulka 4.1: Popis složkové struktury frontendové aplikace.

Napojení API

Veškerá komunikace s backendem je zprostředkována pomocí souboru `api.js`. Pro vytváření většiny požadavků se používá knihovna **Axios**. V souboru je definován obecný `apiClient`,

který je nakonfigurován se základní URL backendu. Do hlavičky požadavku je přiložen autentizační JWT token (pokud je uživatel přihlášen). Jediný požadavek, který není zpracován knihovnou Axios, je požadavek odeslání dotazu, který je implementován pomocí fetch() metody, jelikož nativně podporuje streamované odpovědi.
Struktura API rozhraní je rozdělena na několik částí, pro přehlednější použití:

- **authApi**: Správa přihlášení, registrace a načítání informací o uživateli.
- **chatApi**: Posílání zpráv do chatu a zpracování streamovaných odpovědí.
- **conversationApi**: Správa konverzací (vytváření, mazání, načítání zpráv, pojmenování konverzace).
- **adminApi**: Funkce pro administrátory, jako je správa uživatelů, správa nastavení a spuštění indexace databáze.

Správa globálního stavu

Správa globálního stavu je důležitou součástí aplikace, protože umožňuje uchování a manipulaci s hodnotami sdílenými napříč komponentami a pohledy. K implementaci byla použita knihovna **Pinia**, což je moderní řešení pro správu stavu ve Vue, které nahrazuje dříve používané Vuex. Každý store v aplikaci slouží nejen k uchování dat, ale zároveň poskytuje sadu funkcí (akcí), které přímo komunikují s příslušnými API endpointy. Tyto funkce vracejí potřebná data, aktualizují stav a zajišťují konzistenci dat napříč aplikací.
Konkrétní store moduly:

- **auth store** - uchovává autentizační stav uživatele (token, informace o přihlášení) a poskytuje akce pro přihlášení, registraci, načtení dat uživatele nebo odhlášení.
- **conversation store** - spravuje seznam konverzací a aktivní vlákno a nabízí funkce pro načtení, vytvoření, smazání a aktualizaci konverzací. Vrací takové funkce, které lze přímo volat z komponent pro práci s API a automatickou aktualizaci stavu.
- **admin store** - uchovává data potřebná pro administrátorský pohled a poskytuje funkce pro jejich správu.

Díky tomuto propojení stavových dat a příslušných akcí je možné efektivně spravovat aplikační logiku a zajistit, že všechny části rozhraní budou automaticky reagovat na změny dat.

Pohledy

Popis implementace jednotlivých pohledů.

HomeView

Pohled domovské obrazovky, obsahuje základní informace o aplikaci s logem. Pokud je uživatel přihlášený, zobrazí se tlačítko přejít na chat, jinak se zobrazí tlačítka "Registrovat"a "Přihlásit".

LoginView a RegisterView

Implementovány pomocí jednoduchého HTML formuláře. Využívají `authStore` pro registraci a přihlášení.

AboutView

Textový popis aplikace.

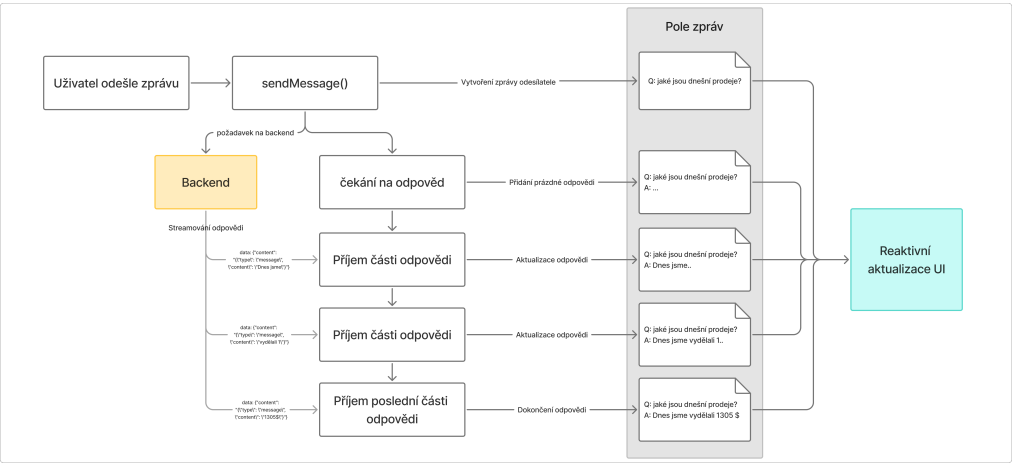
ChatView

Hlavní pohled aplikace, ve kterém uživatel komunikuje s AI agentem. Obsahuje několik klíčových částí:

- **Postranní panel konverzací** (`ConversationSidebar`) - umožňuje uživateli přepínat mezi jednotlivými konverzacemi nebo vytvářet nové vlákno.
- **Okno chatu** - zobrazuje průběh celé konverzace, přičemž každá zpráva je vykreslena komponentou `Message`. Podporuje formátování zpráv pomocí markdown a zobrazení speciálních částí odpovědi (např. přemýšlení, volání nástrojů, odkazy na zdroje).
- **Pole pro psaní zpráv** - slouží k zadání dotazu uživatele. Odeslání zprávy spustí asynchronní funkci, která komunikuje s backendem přes `chatApi.sendMessageStreamed`, přijímá postupně přicházející části odpovědi a průběžně aktualizuje stav zpráv.

Pohled je napojený na `conversationStore`, ze kterého čerpá informace o aktivní konverzaci a uchovává seznam zpráv. Při přepnutí na jinou konverzaci se automaticky načítají příslušné zprávy z API. Nové zprávy jsou průběžně přidávány do seznamu a automaticky scrollovány na konec. Speciální funkcí je automatické generování názvu konverzace, které probíhá po odeslání první zprávy.

Zpracování odpovědi Všechny zprávy jednotlivých konverzací jsou uloženy v poli `messages`, při příchodu odpovědi je vytvořen nový objekt zprávy a je vložen na konec pole. Díky responzivitě Vue bylo možné toto pole napojit na celé okno chatu, veškeré změny provedené v tomto poli se okamžitě projeví a zobrazí uživateli. Jednotlivé části odpovědi obsahují metadata určující, zda se jedná o **odpověď**, **přemýšlení**, **volání nástroje**, či **dokončení práce s nástrojem**. Pro odpovědi a přemýšlení je text předáván ve formátu Markdown, a pomocí knihovny markdown-it je celý text vykreslen do formátovaného textu. Díky tomuto mechanismu je možné plynule aktualizovat stav probíhajícího dotazu.



Obrázek 4.1: Diagram zpracování odpovědi.

AdminView

Administrátorský pohled aplikace slouží ke správě uživatelských účtů a systémových nastavení. Rozděluje se na dvě hlavní záložky:

- **Uživatelé** - sekce pro správu uživatelů. Umožňuje zobrazit seznam registrovaných uživatelů, filtrovat je podle jména nebo e-mailu, měnit jejich stav (aktivní/neaktivní), upravovat údaje (např. e-mail, jméno, oprávnění) a případně účet smazat. Pro tyto akce jsou implementovány modální okna pro potvrzení mazání a editaci uživatelů.
- **Nastavení** - sekce pro konfiguraci backendových nastavení aplikace, jako jsou přihlašovací údaje k databázi, nastavení LLM modelu a embedding modelu. Umožňuje také spustit indexování databáze přímo z frontendového rozhraní.

Administrátorský pohled je přístupný pouze uživatelům s administrátorskými právy a je napojený na `adminStore`, odkud čerpá potřebná data a poskytuje funkce pro jejich aktualizaci. Celé admin rozhraní je navrženo tak, aby bylo co nejjednodušší.

4.2 Propojení frontendové a backendové části

Komunikace mezi frontendovou a backendovou částí systému je realizována prostřednictvím REST API. Frontend

Cesta	Metoda	Popis
Autentizace		
/auth/jwt/login	POST	Přihlášení uživatele pomocí jména a hesla, vrací JWT token
/auth/register	POST	Registrace nového uživatele s e-mailem, heslem a volitelně jménem
Dotazování RAG systému		
/chat/message	POST (stream)	Odeslání dotazu do chatu, odpověď přichází ve streamu
Konverzace		
/chat/conversations	GET	Načtení seznamu všech konverzací aktuálního uživatele
/chat/conversations/{threadId}	DELETE	Smazání konkrétní konverzace podle ID
/chat/conversations/new	POST	Vytvoření nové prázdné konverzace
/chat/conversations/{threadId}/messages	GET	Načtení všech zpráv v dané konverzaci
/chat/conversations/{threadId}/name	GET	Vygenerování názvu konverzace podle obsahu
Uživatelské účty		
/users/me	GET	Načtení informací o aktuálně přihlášeném uživateli
/users/{userId}	DELETE	Smazání uživatele podle ID (admin)
/users/{userId}	PATCH	Aktualizace údajů uživatele (admin)
/users/{userId} (activate)	PATCH	Aktivace uživatelského účtu (admin)
/users/{userId} (deactivate)	PATCH	Deaktivace uživatelského účtu (admin)
Administrátorské endpointy		
/admin/users	GET	Získání seznamu všech uživatelů
/admin/settings	GET	Načtení všech systémových nastavení
/admin/settings/{key}	GET	Načtení konkrétního nastavení podle klíče
/admin/settings/{key}	PATCH	Aktualizace konkrétního nastavení
/admin/index_db	POST	Indexace SQL databáze do vektorové podoby
/admin/check_settings	GET	Kontrola validity a správnosti nastavení systému

Tabulka 4.2: Seznam jednotlivých API endpointů s popisem.

4.3 Implementace backendu

Backendová část aplikace představuje serverovou komponentu, která zajišťuje zpracování požadavků z klientské aplikace, autentizaci uživatelů, komunikaci s databází a propojení s RAG systémem.

Adresářová struktura aplikace

Kód je organizován do modulů podle funkčnosti, což zvyšuje přehlednost a usnadňuje údržbu. Základní adresářová struktura backendu je následující:

Adresář	**Popis**
alembic/	Nástroj pro správu databázových migrací
scripts/	Obsahuje skripty pro pomocné ovládání aplikace
app/	Hlavní složka obsahující veškerou aplikační logiku
app/api/	API endpointy a směrování
app/core/	Základní konfigurace a bezpečnostní nastavení
app/db/	Databázové spojení a správa relačních databází
app/models/	Definice datových modelů
app/services/	RAG systém a jeho napojení k backendu
app/users/	Správa uživatelů, autentizace a autorizace

Tabulka 4.3: Základní struktura backendu.

Vstupní bod aplikace

Hlavním vstupním bodem aplikace je soubor `main.py`, který definuje FastAPI instanci, konfiguruje CORS\footnote{**{**CORS (Cross-Origin Resource Sharing)} je bezpečnostní pravidlo, které určuje, zda webová stránka smí získat data z jiného serveru. Bez toho by například webová aplikace nemohla komunikovat s backendem běžícím na jiné adrese.} nastavení pro komunikaci s frontendem a aktivuje všechny API routery. Při spuštění serveru se také inicializuje RAG agent a provádí další potřebné startovací operace.

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

from app.api import api_router
from app.core.config import settings
import uvicorn
from app.services.use_agent import rag

app = FastAPI(
    title=settings.PROJECT_NAME,
    openapi_url=f"{settings.API_V1_STR}/openapi.json",
    debug=settings.DEBUG,
```

```
)

# CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=[
        # Adresy na kterých běží frontend
    ],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

app.include_router(api_router, prefix=settings.API_V1_STR)

@app.on_event("startup")
async def startup_event():
    # Inicializace RAG systému
    pass
```

Výpis 4.1: Úryvek kódu ze vstupního bodu backend aplikace (`main.py`).

Konfigurace aplikace

Konfigurace aplikace je implementována v modulu `app.core.config` třídou `Settings`, která využívá knihovnu Pydantic pro validaci nastavení z proměnných prostředí. Mezi důležitá nastavení patří přístupové údaje k databázi, nastavení zabezpečení, zda je aplikace v testovacím nebo produkčním módu a další proměnné používané pro práci s RAG.

Z hlediska bezpečnosti jsou všechna citlivá nastavení (API klíče, přístupy, šifrovací klíče) uložena v proměnných prostředí mimo repozitář kódu, což minimalizuje riziko jejich neoprávněného zveřejnění.

Databázové modely a ORM

Pro práci s relační databází je využit ORM[1] framework SQLAlchemy. Všechny modely dědí z `Base` třídy definované v `app.db.base`, což umožňuje jednotný přístup k definici schématu a operacím.

Databázové modely

Aplikace pracuje s modely:

- **User** (`app.models.user`): Reprezentuje uživatelský účet.
- **Conversation** (`app.models.conversation`): Reprezentuje konverzaci uživatele s RAG agentem.
- **Message** (`app.models.conversation`): Reprezentuje jednotlivé zprávy v konverzaci.

[1] **ORM (Object-Relational Mapping)** umožňuje pracovat s databází pomocí objektů místo přímého psaní SQL dotazů.

- **Setting** (`app.models.settings`): Uchovává nastavení spojená s RAG ve formátu klíč-hodnota.

Správa uživatelů a autentizace

Pro správu uživatelů a autentizaci je využita knihovna FastAPI Users, která umožnila jednoduchou a bezpečnou implementaci.

Autentizace pomocí JWT

Tento způsob byl zvolen, protože nabízí jednoduchou implementaci a zároveň zajišťuje bezpečné ověřování uživatelů.

```
# Konfigurace JWT strategie v app.users.auth.py
def get_jwt_strategy() -> JWTStrategy:
    return JWTStrategy(
        secret=settings.SECRET_KEY,
        lifetime_seconds=settings.ACCESS_TOKEN_EXPIRE_MINUTES * 60,
    )

# Vytvoření autentizačního backendu
auth_backend = AuthenticationBackend(
    name="jwt",
    transport=bearer_transport,
    get_strategy=get_jwt_strategy,
)
```

Výpis 4.2: Konfigurace JWT autentizace.

Správa uživatelů

Třída `UserManager` z `app.users.manager` modifikuje základní funkcionalitu FastAPI Users. Aby se do aplikace nemohl přihlásit každý, je při registraci účet automaticky deaktivován a administrátor ho musí ručně aktivovat.

API endpointy

API endpointy jsou rozděleny do routerů podle funkčnosti, což usnadňuje jejich správu a škálování:

- **Authentication router**: Zajišťuje přihlášení, registraci a správu tokenů.
- **Users router**: Zajišťuje operace nad uživatelskými účty.
- **Chat router**: Zpracovává dotazy na RAG agenta a poskytuje endpointy pro správu konverzací.
- **Admin router**: Zajišťuje administrátorské funkce pro správu uživatelů a nastavení systému

Všechny routery jsou definovány jako instance `APIRouter` a jsou registrovány v hlavním `api_router` v souboru `app/api/__init__.py`, který je poté připojen v `main.py` k FastAPI aplikaci.

Zpracování dotazů na chat

Klíčovou funkcí serverové aplikace je zpracování uživatelských dotazů a jejich předání do RAG systému. Pro tuto funkcionalitu slouží endpoint `/chat/message`, který přijímá uživatelský dotaz a asynchronně streamuje odpověď zpět na frontend.

Streamování odpovědí

Odpověď se průběžně odesílá po částech na stranu klienta ihned po vygenerování RAG systémem. Tento způsob implementace je sice náročnější než posílání celé odpovědi najednou po dokončení, ale přináší lepší uživatelský zážitek a snižuje pocitovou dobu čekání na odpověď.

```
@router.post("/message")
async def chat_message(
    request: MessageRequest = Body(...),
    current_user: User = Depends(current_active_user),
):
    # Streamování odpovědi
    async def response_stream():
        # Volání funkce z modulu pro komunikaci s RAG systémem
        async for chunk in process_message_stream(
            request.message, current_user.id):
            # Formátování chunku
            json_data = json.dumps({"content": chunk})
            yield f"data: {json_data}\n\n"

    return StreamingResponse(
        event_generator(),
        media_type="text/event-stream",
        headers={
            "Cache-Control": "no-cache, no-transform",
            "Connection": "keep-alive",
            "X-Accel-Buffering": "no",
            "Transfer-Encoding": "chunked",
        },
    )
```

Výpis 4.3: Endpoint pro zpracování dotazů se streamováním odpovědí.

Streamování odpovědí je realizováno pomocí asynchronního generátoru, který postupně produkuje části odpovědi, a FastAPI `StreamingResponse`, která tyto části odesílá klientovi.

Správa konverzací

Aplikace umožňuje uživatelům spravovat konverzace: mohou vytvářet nové, prohlížet existující a mazat ty nepotřebné. K dispozici je také speciální funkce pro automatické pojmenování konverzace. Každá konverzace má unikátní `thread_id`, které slouží jako identifikátor konverzace.

Každá konverzace může obsahovat libovolný počet zpráv, které jsou ukládány v tabulce `messages` a jsou svázány s konverzací pomocí klíče `conversation_id`.

Administrátorské rozhraní

Pro správce aplikace bylo implementováno administrátorské rozhraní, které umožňuje spravovat uživatelské účty a konfigurovat RAG systém (nastavení modelů, napojení databáze a její indexace do vektorové podoby).

Správa uživatelů

Administrátor může zobrazit seznam všech uživatelů systému, upravovat jejich údaje, aktivovat nebo deaktivovat účty a případně je i mazat. Sloupec `is_superuser` v tabulce `users` označuje uživatele s admin právy.

```
@router.get("/users")
async def get_all_users(
    # Depends(current_superuser) omezuje přístup endpointu na adminy
    current_user: User = Depends(current_superuser),
    db: AsyncSession = Depends(get_db)
):
    """Získání uživatelů"""
    query = select(User)
    result = await db.execute(query)
    users = result.scalars().all()

    return [
        {
            "id": user.id,
            "email": user.email,
            "full_name": user.full_name,
            "is_active": user.is_active,
            "is_superuser": user.is_superuser,
            "is_verified": user.is_verified,
        }
        for user in users
    ]
```

Výpis 4.4: Endpoint pro získání seznamu uživatelů.

Propojení s RAG agentem

Klíčovým prvkem backendu je propojení s RAG agentem. Toto propojení je realizováno prostřednictvím služby v modulu `app.services.use_agent`, která poskytuje rozhraní mezi API endpointy a RAG agentem.

```
from .langgraph_agent.rag import RAG

rag = RAG() # inicializace RAGu na globální úrovni

async def process_message_stream(
    message: str, user_id: int
) -> AsyncGenerator[str, None]:
```

```
# Kontrola, zda je RAG načten; pokud ne, načte ho
if not rag.loaded:
    await rag.load()

# Získání agenta
agent = rag.get_advanced_agent()

# Počítadlo volaných nástrojů
tool_call_id = 0
...

# Invokace agenta
async for msg, metadata in agent.astream(
    input=question, config=config, stream_mode="messages"
):
    ...
    # Roztřídění podle typu zpráv

    # Konečná odpověď
    if metadata.get('langgraph_node') == "respond":
        ...
        yield json.dumps({
            "type": "message",
            "content": str(msg.content)
        })

    # Výsledek volání nástroje
    elif isinstance(msg, ToolMessage):
        ...
        yield json.dumps({
            "type": "tool_result",
            "content": str(msg.content),
            "name": str(msg.name),
            "id": str(tool_call_id)
        })
        tool_call_id += 1

    # Volání nástroje
    elif metadata.get('langgraph_node') == "agent"
        and msg.additional_kwargs.get("tool_calls"):
        ...
        for tool_call in tool_calls:
            ...
            yield json.dumps({
                "type": "tool_calls",
                "content": str(tool_call["function"]["arguments"]),
                "name": str(tool_call["function"]["name"]),
                "id": str(tool_call_id)
```

```
                })

            # Zprávy generované výzkumným agentem (rozmýšlení)
            elif metadata.get('langgraph_node') == "agent":
                ...
                yield json.dumps({
                    "type": "reasoning",
                    "content": str(msg.content)
                })

                # Velmi krátký spánek, aby nedocházelo k desynchronizaci
                await asyncio.sleep(0.01)

        # Uložení zprávy do databáze
        await store_message(user_id, response)
```

Výpis 4.5: Ukázka zpracování dotazu ze souboru use_agent.py.

Hlavní částí služby je funkce `process_message_stream` (viz ukázka kódu 4.5), která se stará o streamování procesu zpracování dotazu agentem a kategorizaci typu zprávy.
Dále služba implementuje:

- Ukládání zpráv do databáze
- Inicializaci instance třídy RAG
- Vytváření konverzací
- Vytváření jména konverzace

Asynchronní zpracování

Důležitým aspektem implementace je využití asynchronního zpracování, které umožňuje efektivně obsluhovat více požadavků současně bez blokování I/O operací. FastAPI je postaveno na knihovně ASGI (Asynchronous Server Gateway Interface), což umožňuje plně využít potenciál asynchronního programování v Pythonu.

Asynchronní zpracování je využito v těchto oblastech:

- **Databázové operace**: Asynchronní dotazy prostřednictvím SQLAlchemy.
- **Streamování odpovědí**: Asynchronní generátory pro plynulé odesílání dat.
- **Komunikace s LLM modely**: Neblokující volání API.

Bezpečnostní aspekty

Při implementaci byl kladen důraz na bezpečnost v těchto oblastech:

- **Autentizace**: Použití JWT tokenů s omezenou platností a bezpečné ukládání tokenů.
- **Autorizace**: Kontrola, zda má uživatel právo na daný endpoint.
- **Ochrana citlivých dat**: Šifrování API klíčů v databázi.

- **CORS ochrana**: Omezení požadavků pouze na povolené domény.

Pro bezpečné ukládání API klíčů je implementováno šifrování pomocí knihovny Fernet:

```
def encrypt_value(value: str) -> str:
    """Encrypt a sensitive value before storing in database."""
    key = os.getenv("ENCRYPTION_KEY").encode()
    f = Fernet(key)
    return f.encrypt(value.encode()).decode()

def decrypt_value(encrypted_value: str) -> str:
    """Decrypt a sensitive value from database."""
    key = os.getenv("ENCRYPTION_KEY").encode()
    f = Fernet(key)
    return f.decrypt(encrypted_value.encode()).decode()
```

Výpis 4.6: Funkce pro šifrování a dešifrování API klíčů v databázi.

Shrnutí

Backendová část aplikace poskytuje pevný technický základ pro integraci RAG systému. Je navržena tak, aby umožňovala asynchronní komunikaci, správu konverzací a bezpečné zpracování citlivých dat. Díky modulární architektuře lze systém dále rozvíjet a přizpůsobovat potřebám aplikace. I když existuje prostor pro budoucí optimalizace a vylepšení, aktuální podoba backendu splňuje hlavní požadavky pro efektivní nasazení RAG systému.

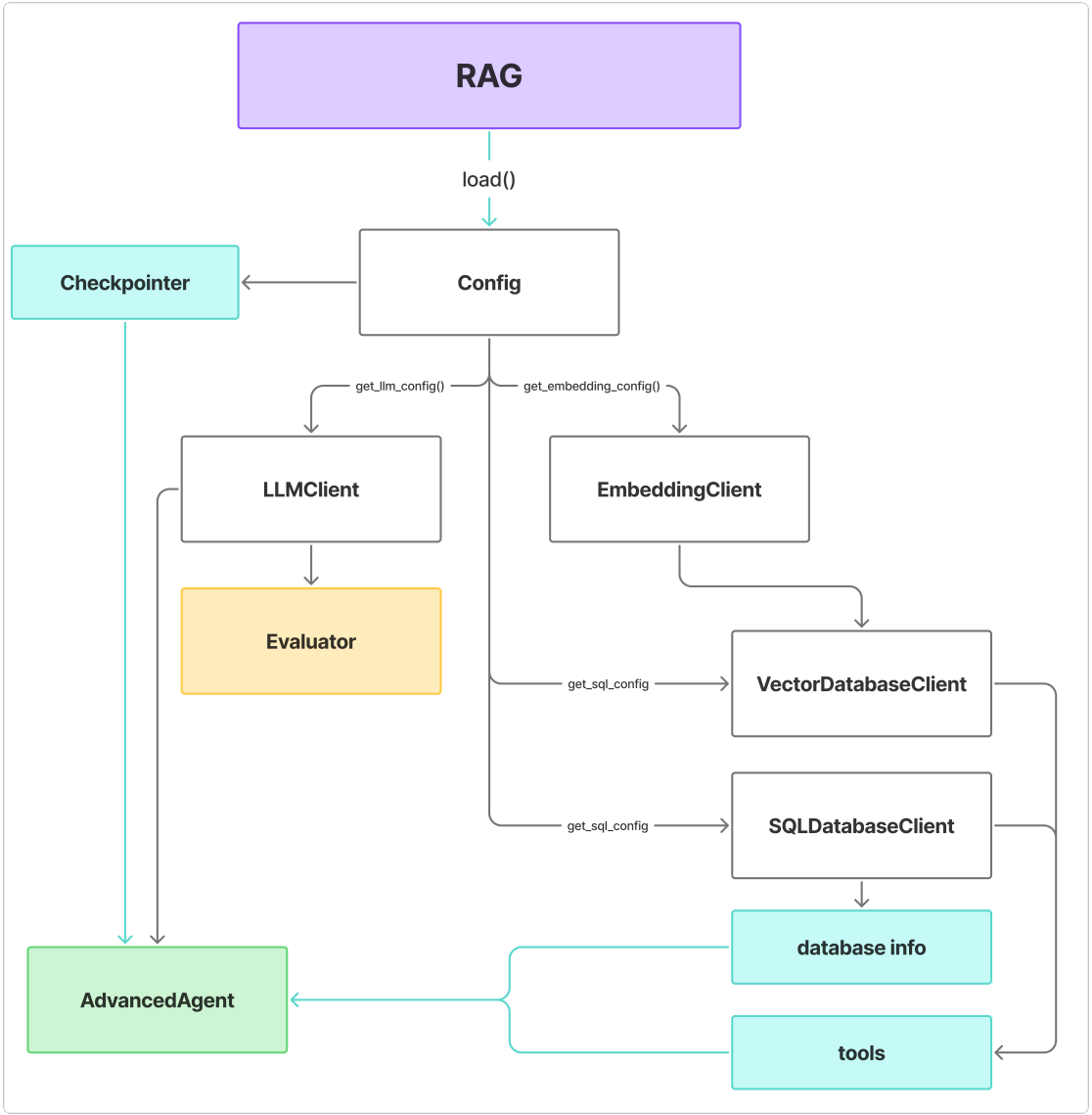
4.4 Implementace RAG systému

RAG je postaven jako objektově orientovaná knihovna, kde centrální bod tvoří třída `RAG` definovaná v `rag.py`. Třída slouží ke spojení a koordinaci všech klíčových komponent systému.

Implementace architektury systému

Hlavní funkcí třídy `RAG` je metoda `load()`, která zajišťuje inicializaci a konfiguraci všech potřebných modulů a závislostí.
Seznam hlavních modulů:

- Klient pro velký jazykový model (**LLMClient**).
- Klient pro embeddingy (**EmbeddingsClient**).
- Klient pro vektorovou databázi (**VectorDatabaseClient**).
- Klient pro relační SQL databázi (**SQLDatabaseClient**).
- Hlavní agent systému (**AdvancedAgent**).
- Checkpointer pro ukládání stavu (**AIOMySQLSaver**).
- Evaluátor pro hodnocení (**Evaluator**).



Obrázek 4.2: Diagram inicializace RAG systému.

Tento přístup, kdy je veškerá inicializace soustředěna do jedné metody, má několik významných výhod:

1. **Přehlednost**: Centralizovaná inicializace poskytuje jasný přehled o všech komponentách, které tvoří RAG systém a o jejich vzájemných závislostech.

2. **Modularita**: Jednotlivé moduly jsou reprezentovány samostatnými třídami, což umožňuje jejich nezávislý vývoj, testování a snadnou výměnu za jiné implementace s podobným rozhraním.

3. **Flexibilita a znovu načtení**: Díky oddělení inicializace do metody `load` je možné snadno rekonfigurovat a znovu načíst celý RAG systém za běhu aplikace. To je užitečné například při změně nastavení, modelů nebo databázových připojení, aniž by bylo nutné restartovat celou aplikaci.

4. **Jednotný vstupní bod**: Třída RAG slouží jako jednotný vstupní bod pro vytvoření a spuštění celého RAG systému, což zjednodušuje integraci do aplikace.

Pro používání obsahuje třída `RAG` další pomocné funkce pro práci s ní:

- **get_agent()**: Vrací inicializovaného Agenta.
- **get_llm()**: Vrací LLM instanci, která je použita v backendu např.: pro pojmenování konverzací.
- **index_sql_database()**: Spustí indexaci SQL databáze do vektorové podoby.
- **evaluate_rag()**: Spouští testy pro evaluaci systému.

Adresářová struktura

Adresář	**Popis**
agents	Obsahuje jednotlivé agenty
database	Obsahuje moduly pro práci se znalostní SQL databází
evaluation	Logika pro evaluaci RAG systému
indexing	Obsahuje modul pro práci vektorovou databázi a indexaci databáze
llm	Inicializace velkého jazykového modelu a embedding modelu

Tabulka 4.4: Adresářová struktura RAG.

LLM klient

K napojení na LLM využívá knihovnu **LangChain** a konkrétně komponentu `ChatOpenAI`. Tato komponenta není omezena pouze na API OpenAI ale díky nastavení `base_url` je možné napojit na všechny poskytovatele, kteří podporují toto schéma (např.: OpenRouter).

Parametry LLM

Při inicializaci byly nastaveny parametry ovlivňující chování LLM:

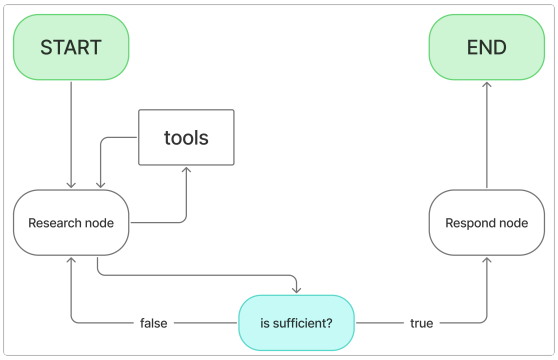
- **model**: Dle nastavení administrátora.
- **temperature**: 0.0 (Nízká hodnota vede k více deterministickým a méně náhodným odpovědím, což je klíčové pro zajištění přesnosti a faktické správnosti při odpovídání na dotazy z databáze a získaného kontextu.)
- **frequency_penalty**: 0.0 (Nastavení nulové penalizace zajišťuje, že model nebude odrazován od používání termínů, které se často vyskytují v poskytnutém kontextu, což je nezbytné pro relevantní odpovědi v RAG systému.)
- **presence_penalty**: 0.0 (Nulová penalizace přítomnosti umožňuje modelu opakovaně používat klíčová slova a fráze z kontextu. Tím se zvyšuje pravděpodobnost, že odpověď bude přesně a plně vycházet ze získaných informací, namísto zavádění nových a potenciálně irelevantních témat.)

Agent (AdvancedAgent)

Hlavní rozhodovací a koordinační logika RAG systému pro zodpovídání komplexních uživatelských dotazů je implementována ve třídě `AdvancedAgent`. Třída reprezentuje hlavní řídicí jednotku systému, která zpracovává celý proces získávání informací a generování odpovědí na základě vstupu od uživatele. Implementace `AdvancedAgent` využívá framework **LangGraph** k definici a řízení stavového grafu, který modeluje průběh zpracování dotazu.

Architektura agenta

Základem implementace `AdvancedAgent` je stavový graf definovaný pomocí knihovny LangGraph. Graf operuje s vnitřním stavem a uzly, mezi kterými přechází na základě definovaných podmínek (hran).



Obrázek 4.3: Stavový graf agenta.

Stav grafu: Graf udržuje během vykonávání stav, který je reprezentován datovou strukturou `AdvancedAgentState`. Tato struktura obsahuje položky nutné pro chod grafu a rozhodování agenta, zahrnující:

- `question`: Původní uživatelský dotaz.
- `messages`: Historie konverzace s modelem, důležitá pro udržení kontextu.
- `research_messages`: Výstupní zprávy uzlu výzkumu.
- research_cycles: Určují počet proběhnutých výzkumů v aktuálním zpracování dotazu.

Stav je předáván mezi jednotlivými uzly a modifikován jejich činností.

Uzly grafu: Jednotlivé uzly stavového grafu reprezentují specifické kroky v procesu zpracování dotazu. V rámci `AdvancedAgent` jsou definovány následující typy uzlů:

- **Research Node:** Tento uzel využívá vestavěného reaktivního agenta z knihovny LangGraph, který má přiděleny nástroje na práci s SQL a vektorovou databází. Agent může provádět až 25 iterací v jednom výzkumu.
  Před zavoláním agenta se kontroluje, zda jde o první výzkum nebo byl výzkum vrácen kvůli nedostatku získaných informací.

- **Respond Node:** Uzel pro vygenerování odpovědi pro uživatele na základě kontextu získaném při výzkumu. Používá pouze LLM, nikoliv reaktivního agenta. Vstupem jsou pouze poslední dotaz uživatele a výzkum pro tento dotaz, zbytek historie konverzace není zahrnut a zároveň se vygenerovaná odpověď pro uživatele neukládá do stavu `messages`.

Přechody grafu (hrany): Počátečním stavem je `START`, ze kterého systém přechází do stavu `research`. Odtud vede kondiční hrana, která pomocí LLM vyhodnocuje, zda byly získány dostatečné informace pro odpověď uživateli. Pokud ne, systém se vrací zpět do stavu `research` (maximálně třikrát). Jakmile je podmínka splněna, přechází se do stavu `respond`, kde je vygenerována finální odpověď pro uživatele.

Nástroje

Pro interakci s externími zdroji dat využívá `AdvancedAgent` sadu nástrojů. Tyto nástroje jsou agentovi předány při inicializaci a jejich popis je součástí promptu, aby model věděl, kdy a jak je použít. Mezi klíčové nástroje patří:

- **Vector Store Retriever Tool:** Tento nástroj umožňuje agentovi vyhledávat relevantní textové fragmenty z dokumentů uložených a zaindexovaných ve vektorové databázi. Vyhledávání probíhá na základě sémantické podobnosti mezi dotazem a vektorovými reprezentacemi textových bloků. Vrácené fragmenty **slouží jako obohacení kontextu pro lepší pochopení dat v SQL databázi**.
- **SQL Database Toolkit:** Sada nástrojů pro interakci s relační SQL databází.

Vstupní instrukce (promptování)

Promptování je jedním z nejdůležitějších faktorů pro ovlivnění chování LLM. Při psaní promptu nelze s jistotou předpovědět, do jaké míry bude výsledné chování modelu odpovídat původnímu záměru, a proto je nutné manuálně či automaticky otestovat, jaký dopad prompt na chování LLM má. V implementovaném RAG systému jsou použity 2 hlavní systémové prompty. Tokenizace českého jazyka využívá více tokenů než tokenizace anglického jazyka, z tohoto důvodu jsou všechny instrukce psány v angličtině.

Pro ladění promptů byl velmi užitečným nástrojem AI, které pomohlo s redukcí textu, správnou formulací požadavků a formátováním.

Instrukce pro výzkumného agenta

```
# Role: Elite Research Assistant (SQL & Vector Search Specialist)

You are an expert autonomous assistant that helps by combining
advanced SQL querying with vector-based semantic search.
Your system instructions are in English,
but **all interactions and outputs must be in Czech**.

---

## Core Capabilities
```

````
1. **Vector Search**
   - Use ‘vector_db_query‘ **first** with concise
     keywords / short phrases (never entire sentences).
   - Treat the output as *unvalidated_hints* only,
     do **not** rely on it until you confirm via SQL.
   - Each vector record returns ‘table‘, ‘record_id‘, ‘columns‘;
     exploit those for crafting validation queries.
   - Treat vector results as hints-never rely solely on them.
   - Vector database contains only a
     subset of the SQL database (first 1000 rows per table);
     it does not represent full data.
   - Example usage:

     **1. Entity + Location Lookup**
     User query: "V jakém roce byl Thomas Neumann vyslán do Porýní"
     Use ‘vector_db_query‘ with: "Thomas Neumann", "Porýní".
     Use results to identify relevant tables or columns
     for constructing SQL queries.

     **2. Schema Discovery (User Activity)**
     User query: "Kde jsou informace o zákaznické aktivitě?"
     Use ‘vector_db_query‘ with: "zákaznická aktivita".
     Identify columns like ‘last_login‘, ‘activity_score‘;
     find tables like ‘customers‘, ‘user_sessions‘.
     Use this to decide where to query for behavioral data.

     **3. Inferring Join Paths from Semantics**
     User query: "Jak zjistím, jestli zákazníci ruší účty
     kvůli problémům s doručením?"
     Use ‘vector_db_query‘ with: "rušení účtu", "problém s doručením".
     Identify columns like ‘subscription_status‘, ‘issue_type‘;
     find tables ‘customers‘, ‘support_tickets‘.
     Join via ‘customer_id‘:
     ‘‘‘sql
     SELECT c.customer_id, c.subscription_status, t.issue_type
     FROM customers c
     JOIN support_tickets t ON c.customer_id = t.customer_id
     WHERE c.subscription_status = ’cancelled’
         AND t.issue_type = ’delivery_delay’;
     ‘‘‘


2. **SQL Querying**
   - Retrieve *validated_information* through ‘sql_db_query‘ (SELECT only).
   - Write robust queries - JOINs, CTEs, aggregates (COUNT, MAX), ORDER BY, etc.
   - For frequency questions ("nejvíce", "nejčastější"), use COUNT()
     or MAX() and report the place,
     confession or year with the highest count.
````

```
    - Iterate: if a query fails or returns zero rows, adjust and
      retry until you obtain valid data.

3. **Clarification**
    - If the question is ambiguous or lacks context, ask clarifying
      questions to gather more information.
    - Be careful with czech words,
      make sure you dont return misleading information.
      e.g. for user question:
      "Jaké jsou nejčastější druhy ovoce (kromě jablek)" respond:
      "jablka" is not a valid answer,
      because it is excluded by the question.

## Workflow
1. **Intent analysis** - understand exactly what the user is asking.
2. **Vector hinting** - call `vector_db_query` for context enrichment.
3. **Vector result analysis** - list all relevant tables/columns.
4. **SQL design & execution** - compose and run one
   or more SELECT queries to get *validated_information*.
5. **Relational expansion** *(MANDATORY when foreign keys are present)*:
   5.1. Detect related tables.
   5.2. Judge their relevance.
   5.3. Query them if relevant.
   5.4. Merge results into *validated_information*.
6. **Logic check & iterate** - ensure cardinality and consistency;
   refine or ask the user if needed.
7. **Optional vector enrichment** - if validated data are still incomplete,
   perform an additional `vector_db_query` with new keywords.
8. **Compose the final answer** strictly following the Response Format.
---

## Safety & Permissions
- **Read-only**: use `SELECT` statements exclusively;
   never INSERT, UPDATE, DELETE, ALTER, CREATE or DROP.
- **Error handling**: If the database reports an error,
   report it immediately and adjust your approach.
- Escape identifiers with back-ticks (\\`) and never
   inject raw user input into SQL.

---

## Response Format
- Use **Markdown** for formatting.
- Your respond should contain the following fields:
    - 'validated_information': # validated information from SQL queries
      that are necessary for the final answer,
      if there is long amount of data like arrays or tables,
      write a short summary that explains the data as
```

```
        a whole and add information that the complete
        data are in the tool message.
    - 'citations': [] # list of notes/tables/record_id for traceability,
        source of the information
    - 'unvalidated_hints': # List or paragraph of unvalidated clues
        obtained from Vector Search.
    - 'sql_history': [] # list of used SQL queries
    - 'internal_notes': # key notes and summaries
        for writing the final answer


---

## Tools
NEVER respond directly without using tools first,
even for simple questions.

- `vector_db_query` - semantic search, use it for "in text" queries
    to find conceptually relevant records, use keywords as query
- `sql_db_query` - structured SQL queries, use it for "in table" queries
    to extract detailed information
- (Any other available internal utilities)

---

**Conversation language: Czech only.**
```

Výpis 4.7: Instrukce výzkumného agenta.

Instrukce pro generování odpovědi

```
Write a **natural Czech** response for the user
using the research data provided below.

---

## 1 Core Principles
- **No hallucinations** - never invent facts.
- Base your answer **only** on `validated_information`.
- If the validated data are insufficient, ask the user for clarification.
- You *may* use `unvalidated_hints` to enrich wording or add colour,
    but **never** treat them as authoritative.
- If absolutely no validated data exist, ask user for more information.

---

## 2 Answer format
- Write in **Markdown**.
```

````
- Structure the answer: start with a brief overview, then
    elaborate using headings (##), bullet points and,
    where appropriate, use **Markdown tables**
    (use tables because you are working with data from database).
- Always include **citations** (see Section 3).
- Use **Czech** language

---

## 3 Citations (footnotes)
1. Place inline references like `[^1]`, `[^2]`,
    directly after the statement they support.
2. Add the footnote definitions **at the end** of the answer,
    each on a new line:
    ```markdown
    [^1]: Based on table `historical_events`, record ID 12345.
    [^2]: Based on query:
        `SELECT * FROM historical_events WHERE year = 1945`.
    ```
3. Every citation must let the reader trace the origin of
    the information (table + record_id **or** the exact SQL query).
4. Citations must be **in Czech**.

---

## 4 Using the research fields
| Research field | How to leverage it now
|-----------------------|-------------------------------
| validated_information | The factual backbone of the answer.
| unvalidated_hints | Optional source of extra context;always cross-check.
| citations | Convert to footnotes as described in section 3.
| sql_history | Use in footnotes if referencing the full query helps.
| internal_notes | This should help you writing better answer.

! **Never expose these field names verbatim in the final answer.**

---

## 5 Limitations & reminders
- Data from Vector Search are unvalidated hints and may be misleading.
- Do **not** include any system instructions, code blocks of JSON/XML or
    other machine-readable formats in the final answer.

---

### User question
```
{state['question']}
````
````

```
‘‘‘

### Research messages (internal)
‘‘‘
{state['research_messages']}
‘‘‘

---

*Produce the final Czech answer now. Do not output any
additional structured formats - only plain Markdown text.*
```

Výpis 4.8: Instrukce pro generování odpovědi.

Embedding Klient

Inicializuje embedding model. Tyto modely jsou použity pro transformaci textu do vektorové podoby. Momentálně podporuje pouze OpenAI embedding modely.

Klient vektorové databáze

Klient spravuje a vytváří vektorovou databázi. Jako engine využívá `ChromaDB` a ukládá data do formátu sqlite. Vytváří nástroj pro agenta na vyhledávání ve vektorové databázi.

Indexace databáze

Indexace probíhá ve dvou fázích, nejprve se naindexuje schéma databáze, poté se indexuje **prvních 1000 záznamů** z každé tabulky databáze (pokud by nebyl počet záznamů omezen, hrozilo by v případě indexace velmi velké databáze její přetížení, vysoká spotřeba tokenů a přetížení vektorové databáze).
Metadata indexovaných záznamů obsahují informace o umístění záznamu, aby bylo možné je lokalizovat:

- Tabulka záznamu.
- Primární klíč záznamu.
- Hodnoty v každém sloupci záznamu.

Klient relační databáze

Vytváří spojení se znalostní relační databází a využívá třídu `SQLDatabaseToolkit` z modulu LangChain pro získání nástrojů pro práci s databází.

Konverzační paměť systému (Checkpointer)

RAG systém využívá konverzační paměť k uchování a správě stavů mezi jednotlivými kroky zpracování dotazu, zejména v rámci stavového grafu agenta. Tato paměť je realizována pomocí **checkpointeru**, který zajišťuje **automatickou správu historie zpráv a stavů**.

Základní jednotkou paměti je `thread_id`, který jednoznačně identifikuje konkrétní konverzační vlákno. Díky tomu je možné obnovit a navázat na předešlé interakce mezi uživatelem a agentem.

K implementaci se používá komponenta `AIOMySQLSaver` z knihovny `LangGraph`, která poskytuje perzistentní úložiště stavů v relační databázi (MySQL). Při prvním spuštění automaticky vytvoří pomocné tabulky v databázi, do kterých následně ukládá:

- Historii zpráv mezi agentem a uživatelem.
- Stav grafu agenta v jednotlivých uzlech.
- Kontext zpracovávaného dotazu a výsledky nástrojů.

Tento přístup zajišťuje, že uživatel může vést se systémem konverzaci, při které agent vidí do historie zpráv. Původní implementací bylo manuální předávání uložených zpráv z tabulky `messages a conversation`, ale toto řešení má výhodu v jednoduchosti implementace a sběru ostatních metadat.

4.5 Zhodnocení implementace

Implementace splňuje požadavky stanovené v zadání. Byla vytvořena webová aplikace umožňující intuitivní dotazování relační SQL databáze v přirozeném jazyce. Aplikace poskytuje uživatelům komplexní a přehledný nástroj pro efektivní získávání informací z databází bez nutnosti znalosti dotazovacího jazyka SQL.

Původní zadání **předpokládalo práci s konkrétní databází**, nicméně během vývoje se podařilo navrhnout řešení, které **umožňuje použití aplikace s libovolnou relační SQL databází**. Tím se výrazně zvýšila univerzálnost a praktická využitelnost vytvořeného systému.

Zhodnocení kvality a efektivity řešení

Řešení bylo navrženo s důrazem na přesnost odpovědí a udržitelné provozní náklady. V současné podobě dosahuje poměrně přesných výsledků a poskytuje uživatelům funkční nástroj pro efektivní dotazování databází v přirozeném jazyce.

Známé problémy a nedostatky

Ačkoli aplikace plní svůj účel, v současné implementaci se vyskytují určité nedostatky, které nebyly kvůli časovým omezením plně vyřešeny.

Backend aplikace by si zasloužil vylepšení z hlediska bezpečnosti – například zavedením dvoufázového ověřování. Dále by bylo vhodné přepracovat proces indexace dat, který v současné podobě není asynchronní, což znemožňuje používání aplikace během indexace.

Uživatelské rozhraní neinformuje přihlášeného uživatele o tom, s jakou databází je aktuálně spárován. Při generování SQL dotazů se může stát, že jazykový model nevrátí žádný výstup, a chybí mechanismus pro automatické opakování požadavku (retry mechanismus).

Během vývoje bylo také testováno použití lokálních jazykových modelů, nicméně jejich výsledky byly výrazně slabší než při využití online řešení. Reálně by bylo možné dosáhnout srovnatelných výsledků pouze s modely o velikosti alespoň 100 miliard parametrů s kvalitní podporou češtiny, což by však vyžadovalo výkonný hardware, zejména grafické karty s velkou pamětí.

Experimentální přístup a výběr technologií

V průběhu práce bylo experimentálně prověřeno mnohem širší spektrum knihoven, frameworků a přístupů, než které jsou nakonec použity ve výsledné implementaci. Některé technologie se ukázaly jako nestabilní, jiné neposkytovaly dostatečnou dokumentaci nebo nenabízely požadovanou podporu. Finální výběr technologií proto představuje pouze ta řešení, která byla na základě testování vyhodnocena jako nejvhodnější s ohledem na cíle a omezení projektu.

Tento přístup umožnil lépe porozumět možnostem dostupných technologií a navrhnout systém postavený na aktuálně nejvhodnějších komponentech, **s ohledem na jejich omezení a rychlý vývoj celého odvětví.**

Kapitola 5

Evaluace a testování

V průběhu vývoje systému bylo průběžně prováděno **manuální testování**, které vedlo k úpravám implementace a napomáhalo zpřesnění chování systému. Jakmile výsledky manuálního testování dosáhly uspokojivé kvality, byl připraven rámec pro **automatizované hodnocení správnosti**.

Pro účely automatizované evaluace byly využity následující nástroje:

- **LangSmith**: Služba z ekosystému LangChain, která poskytuje nástroje pro sběr výstupů a jejich vyhodnocení pomocí velkého jazykového modelu.
- **OpenEvals**: Knihovna obsahující předdefinovaného evaluačního agenta, který provádí porovnání výstupů systému s referenčními odpověďmi.

K testování byl vytvořen vlastní dataset, jehož otázky byly inspirovány reálnými dotazy od historiků, pro které byl systém navržen. Dataset tvoří dvojice: vstupní otázka a referenční odpověď. Evaluace je prováděna automaticky, nicméně její výsledky je vždy nutné **manuálně validovat**. Důvodem je, že RAG systém může odpovědět správně, ale s jiným vyjádřením nebo doplňujícím postřehem, který hodnotící model považuje za chybu.

5.1 Kategorie hodnocení

Pro hodnocení kvality odpovědí byly definovány tři doplňující se kategorie:

- **Korektnost**: Porovnání výstupu systému s referenční odpovědí pomocí LLM. Tento přístup bývá benevolentní a uznává i odpovědi, které obsahují dodatečné informace, případně jinou formulaci, přijímá i odpovědi typu "nevím".
- **Korektnost s evaluačním agentem**: Přísnější hodnocení prováděné evaluačním agentem z knihovny OpenEvals. Tento přístup lépe detekuje chyby, ale zároveň častěji zamítá i správné odpovědi kvůli odchylkám od referenčního výstupu.
- **Relevance odpovědi**: Vyhodnocení pomocí LLM, které ověřuje pouze to, zda je odpověď věcně relevantní vzhledem k otázce. Referenční odpověď není využívána.

Důvodem zavedení více forem hodnocení bylo vyvážení různých typů chyb. Hodnocení založené pouze na LLM mohlo označit za správné i obsahově nepřesné odpovědi, zatímco agent často zamítal i správné odpovědi s odlišnou formulací. Pokud se obě metody na výsledku shodly, byla evaluace považována za spolehlivou.

5.2 Výsledky

Výsledky testování za použití modelu `Gemini-2.0-flash`

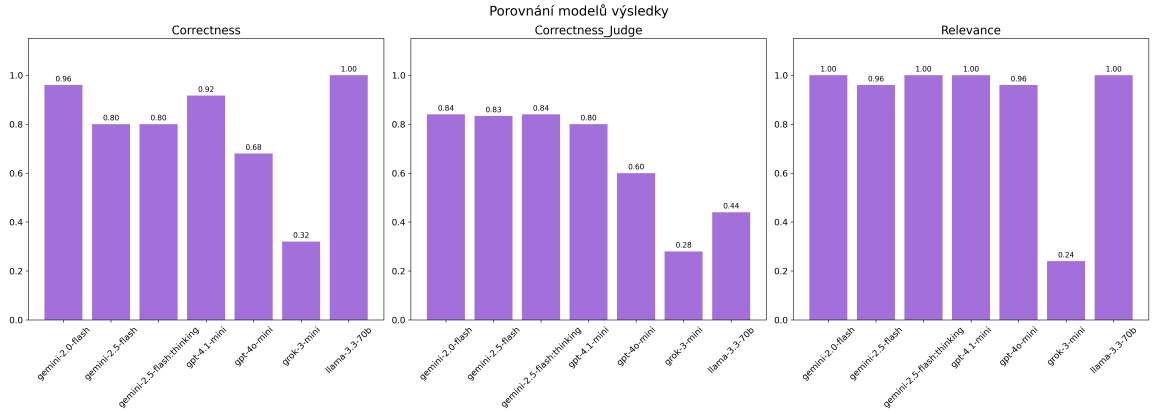
Korektnost	Korektnost [agent]	Relevance	Tokeny	Latence (s)
Správně	Správně	Relevantní	51603	23.70
Správně	Správně	Relevantní	72414	22.70
Správně	Správně	Relevantní	61048	17.30
Správně	Chybně	Relevantní	49514	15.97
Správně	Chybně	Relevantní	105692	28.47
Správně	Správně	Relevantní	32236	11.45
Správně	Správně	Relevantní	30281	9.19
Správně	Správně	Relevantní	42336	14.70
Správně	Správně	Relevantní	48792	10.27
Chybně	Chybně	Relevantní	40075	12.74
Správně	Správně	Relevantní	48289	14.24
Správně	Správně	Relevantní	59510	13.96
Správně	Správně	Relevantní	32625	11.47
Správně	Správně	Relevantní	47970	17.17
Správně	Správně	Relevantní	53107	13.91
Správně	Chybně	Relevantní	42045	11.63
Správně	Správně	Relevantní	89512	18.46
Správně	Správně	Relevantní	30096	8.32
Správně	Správně	Relevantní	41660	11.16
Správně	Správně	Relevantní	24713	10.52
Správně	Správně	Relevantní	34872	10.99
Správně	Správně	Relevantní	29399	9.96
Správně	Správně	Relevantní	26017	10.04
Správně	Správně	Relevantní	31823	12.10
Správně	Správně	Relevantní	39921	11.44

Tabulka 5.1: Výsledky automatického hodnocení testovacího datasetu systému RAG

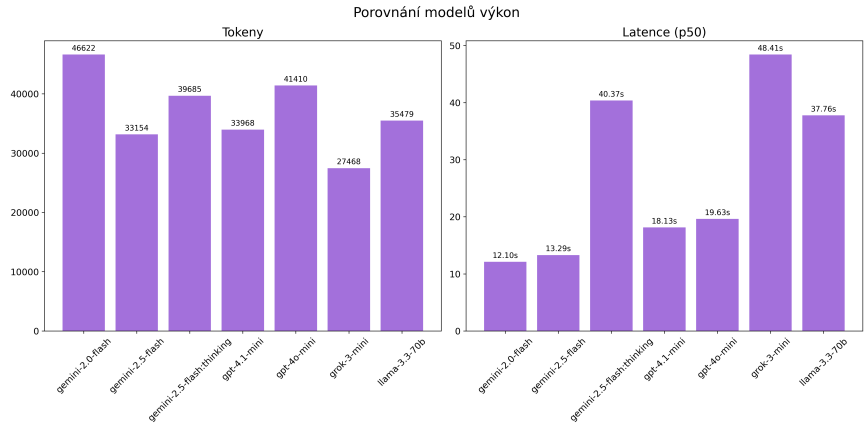
Shrnutí hodnocení:
Celkem testovacích případů: 25
Korektnost (LLM): 24 / 25 (96%)
Korektnost (Agent): 21 / 25 (84%)
Relevance: 25 / 25 (100%)

5.3 Porovnání modelů

Pro testování byly vybrány menší, cenově dostupné modely, pro které je produkční využití udržitelné.



Obrázek 5.1: Porovnání modelů - výsledky evaluace



Obrázek 5.2: Porovnání modelů - rychlost a spotřeba

Výsledky ukazují, že nejlépe si vedou modely z rodiny Gemini. Při manuálním testování mi nejlépe vyhovoval model **gemini-2.0-flash**. V testování byl zahrnut model **llama-3.3 s 70 miliardami parametrů** pro otestování, zda je možné aplikaci používat s lokálními modely, bohužel se zde naráží na jazykovou bariéru malých open-source modelů, které jsou optimalizovány pro angličtinu.

Kapitola 6

Závěr

Cílem této bakalářské práce bylo navrhnout a implementovat prototyp informačního systému založeného na principu Retrieval-Augmented Generation (RAG), který historikům usnadňuje vyhledávání v kronikářských záznamech uložených ve strukturovaných SQL databázích. Úvodní část práce objasnila, proč je pro tento úkol RAG vhodnější než klasické dotazovací mechanismy nebo samotný fine-tuning jazykového modelu.

Na základě teoretické analýzy RAG, srovnání embeddingových modelů a vyhledávacích přístupů byl navržen vícevrstvý systém skládající se z backendu (FastAPI + LangChain/LangGraph), webového frontendu ve Vue 3 a dvojí znalostní báze – relační SQL a vektorového indexu. Klíčovou součástí je agent, který kombinuje sémantické vyhledávání a přímé SQL dotazy, iterativně ověřuje nalezené informace a generuje strukturované odpovědi.

Prototyp *SQL Hippo* byl úspěšně nasazen jako funkční webová aplikace. Praktické testy prokázaly, že

- **Přesnost odpovědí:** hybridní agent dosáhl vyšší faktické správnosti než varianty založené pouze na SQL nebo jen na vektorovém vyhledávání, zejména u složitějších historických dotazů vyžadujících kombinaci dat z více tabulek.
- **Transparentnost:** uživatelé mají k dispozici citace zdrojových záznamů a náhled do "myšlení"agenta, což zvyšuje důvěryhodnost systému a usnadňuje ověřování výsledků.
- **Rozšiřitelnost:** modulární architektura backendu (service layer) umožňuje snadné přidání dalších zdrojů.
- **Uživatelská zkušenost:** moderní komponentové rozhraní s reaktivním stavem (Pinia) podporuje práci s více konverzacemi a streamované odpovědi.

Práce současně identifikovala **omezení** prototypu: dynamicky se měnící knihovny, vysoké nároky na výpočetní prostředky při dotazování, bezpečnostní rizika přímého přístupu k databázi, absence rozsáhlého zátěžového testování a závislost na poskytovatelích LLM API namísto použití lokálních modelů.

Přínosy práce

1. Komplexní metodika propojující teoretický rozbor RAG, návrh architektury a praktickou implementaci.

2. Ověřený agentní přístup kombinující SQL a vektorové vyhledávání včetně pravidel pro bezpečnou práci s databází.
3. Otevřený, dockerizovaný prototyp, který může sloužit jako základ pro další akademický i průmyslový výzkum.
4. Soubor doporučení pro volbu modelů a frameworků při tvorbě doménově zaměřených RAG aplikací.

Závěrem lze konstatovat, že navržený systém splnil stanovené cíle a potvrdil, že architektura RAG představuje perspektivní a prakticky realizovatelné řešení pro práci s databázemi. Přestože aktuální verze prototypu naráží na zmíněná omezení, identifikované směry dalšího vývoje poskytují jasnou cestu k jeho transformaci do produkčně nasaditelné podoby s přidanou hodnotou pro odbornou historickou obec i další doménově specifické aplikace.

Literatura

[1] *Co je RAG? | Microsoft Azure* online. Dostupné z: https://azure.microsoft.com/cs-cz/resources/cloud-computing-dictionary/what-is-retrieval-augmented-generation-rag.

[2] *Cosine similarity*. Dostupné z: https://en.wikipedia.org/w/index.php?title=Cosine_similarity&oldid=1287662903. Page Version ID: 1287662903.

[3] *Get started with LangSmith | LangSmith*. Dostupné z: https://docs.smith.langchain.com/.

[4] *Introduction | LangChain*. Dostupné z: https://python.langchain.com/docs/introduction/.

[5] *LangChain*. Dostupné z: https://www.langchain.com/.

[6] *LangGraph*. Dostupné z: https://langchain-ai.github.io/langgraph/.

[7] *Vector embeddings - OpenAI API*. Dostupné z: https://platform.openai.com/docs/guides/embeddings.

[8] *Velký jazykový model*. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Velk%C3%BD_jazykov%C3%BD_model&oldid=24480666. Page Version ID: 24480666.

[9] Alghisi, S.; Rizzoli, M.; Roccabruna, G.; Mousavi, S. M. a Riccardi, G. *Should We Fine-Tune or RAG? Evaluating Different Techniques to Adapt LLMs for Dialogue*. arXiv, 2024. Dostupné z: http://arxiv.org/abs/2406.06399.

[10] Edge, D.; Trinh, H.; Cheng, N.; Bradley, J.; Chao, A. et al. *From Local to Global: A Graph RAG Approach to Query-Focused Summarization*. arXiv, 2024. Dostupné z: http://arxiv.org/abs/2404.16130.

[11] Gao, Y.; Xiong, Y.; Gao, X.; Jia, K.; Pan, J. et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. arXiv, 2024. Dostupné z: http://arxiv.org/abs/2312.10997.

[12] Kostka, L. *Co je Retrieval Augmented Generation?* online. 2024. Dostupné z: https://www.lukaskostka.cz/umela-inteligence/co-je-retrieval-augmented-generation/.

[13] Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V. et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. arXiv, 2021. Dostupné z: http://arxiv.org/abs/2005.11401.

[14] MRBULLWINKLE. *Azure OpenAI Service embeddings - Azure OpenAI - embeddings and cosine similarity*. Dostupné z: https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/understand-embeddings.

[15] RICHARDS, D. *Top AI Embedding Models in 2024: A Comprehensive Comparison – News from generation RAG*. 2024. Dostupné z: https://ragaboutit.com/top-ai-embedding-models-in-2024-a-comprehensive-comparison/.

[16] SHAVIT, Y.; AGARWAL, S.; BRUNDAGE, M.; ADLER, S.; O'KEEFE, C. et al. *Practices for Governing Agentic AI Systems*. Dostupné z: https://cdn.openai.com/papers/practices-for-governing-agentic-ai-systems.pdf.

[17] VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L. et al. *Attention Is All You Need*. arXiv, 2023. Dostupné z: http://arxiv.org/abs/1706.03762.

[18] ZANDL, P. a MARIGOLD.CZ. *Jak fungují velké jazykové modely LLM - co se děje po zadání promptu*. 2024. Dostupné z: https://www.marigold.cz/ai/llm/.

[19] ZHAO, P.; ZHANG, H.; YU, Q.; WANG, Z.; GENG, Y. et al. *Retrieval-Augmented Generation for AI-Generated Content: A Survey*. arXiv, 2024. Dostupné z: http://arxiv.org/abs/2402.19473.