

Yannick Lansink, 900102137

Beroepsproduct

Software Developer, Hogeschool NOVI

18 Augustus, 2024

Afkortingenlijst

Afkortingen	Definitie
IV	Informatievoorziening
AVG	Algemene Verordening Gegevensbescherming
LLM	Large Language Model
IBS	Integratie Business Services
NLP	Natural Language Processing
RAG	Retrieval Augmented Generation
AI	Artificial Intelligence
ASP.NET	Active Server Pages .NET (een Microsoft framework)
Scaled Agile Framework	SAFe

Disclaimer

“Wegens de vertrouwelijke aard van de informatie in het werkstuk/ de afstudeerscriptie en de bijbehorende bijlagen, mag de inhoud noch geheel noch gedeeltelijk op enigerlei wijze worden aangepast, gewijzigd of verveelvoudigd zonder schriftelijke toestemming van de auteur en het betrokken bedrijf. Zowel de docent/ scriptiebegeleider, de examinerator alsmede de medewerkers van NOVI Hogeschool worden gehouden de inhoud van het werkstuk/ de afstudeerscriptie als strikt vertrouwelijk te behandelen. NOVI Hogeschool bewaart het document in een afgesloten databank. Gedurende de bewaarperiode kunnen studentdossiers worden ingezien door medewerkers van NOVI, de Examencommissie van NOVI alsmede de Inspectie van het Onderwijs en een visitatiecommissie van de Nederlands Vlaams Accreditatieorganisatie (NVAO, bij (her)accreditaties).”

Inhoud

Inleiding.....	5
1. Functioneel ontwerp.....	6
1.1 Praktijk- en literatuuronderzoek voor kernfunctionaliteit	6
1.2 Functionele Requirements	8
1.3 Gebruikersscenario: chatapplicatie in de praktijk.....	9
1.4 Use case diagrammen	10
1.5 Wireframe	12
1.6 Technologiystack	14
1.7 Architectuur	14
1.8 Third-Party Integraties	15
1.9 API Specificaties RAG-service	16
1.10 Database Structuur	16
1.11 Azure Function	17
1.12 Monitoring en Logging.....	17
1.13 Samenvatting.....	18
2. Technisch ontwerp	19
2.1 Terminologie en architectuurcontext.....	19
2.2 Technische requirements	20
2.3 Technische infrastructuur.....	23
2.4 Risicoanalyse	31
3. Werkende software.....	33
3.1 Doelstelling en functionaliteit	33
3.2 Gebruikersgroepen en toepassingsgebied.....	33
3.3 Systeemvoordelen.....	33
3.4 Ontwikkelingsstandaarden.....	34
3.5 Technische implementatie	38
3.6 Systeemarchitectuur	42
3.7 Beveiligingsmaatregelen	43
4. Installatiehandleiding.....	46
4.1 Voorbereiding.....	46
4.2 Configureer de omgevingsvariabelen	47
4.3 Start de backend (RAG-service).....	47
4.4 Start de frontend (Blazor WebAssembly).....	48
4.5 Optioneel: Azure Blob Storage configuratie.....	49
5. Belangrijke functionaliteiten	51

5.1 Gedetailleerde Analyse van Kernfunctionaliteiten	51
Bronvermelding.....	53

Inleiding

In een dynamische en complexe organisatie zoals de Belastingdienst is de toegang tot accurate en actuele informatie essentieel voor het effectief uitvoeren van taken. Integratie Business Services (IBS) Toeslagen, onderdeel van de Belastingdienst, heeft te maken met versnipperde informatiebronnen zoals repositories, wiki's en chatkanalen. Dit belemmert niet alleen de efficiëntie van medewerkers, maar ook de kwaliteit van kennisdeling binnen en tussen teams. In reactie hierop is een Retrieval-Augmented Generation (RAG) systeem ontwikkeld dat als geavanceerde chatapplicatie fungeert.

Deze applicatie biedt een innovatieve oplossing door gebruik te maken van semantische zoektechnologieën, vector-gebaseerde opslag en automatische documentanalyse. De inzet van geavanceerde AI-technologieën, zoals een Large Language Model (LLM), stelt medewerkers in staat om snel relevante informatie te vinden en directe antwoorden te verkrijgen op specifieke vragen. Dit bevordert niet alleen de productiviteit, maar ook de tevredenheid van gebruikers.

De technische architectuur van de applicatie omvat een combinatie van moderne technologieën, waaronder FastAPI, Pinecone, Azure Blob Storage en Blazor WebAssembly. De nadruk ligt op schaalbaarheid, beveiliging en gebruikersgemak. Daarnaast biedt het systeem een robuuste infrastructuur met real-time queryverwerking en automatische documentanalyse, wat resulteert in een efficiëntere en intuïtieve werkomgeving.

Dit rapport beschrijft het functionele en technische ontwerp van de applicatie, de implementatie van kernfunctionaliteiten en de voordelen die de applicatie biedt voor de organisatie. Hiermee draagt het bij aan de optimalisatie van werknemerstevredenheid en efficiëntie binnen IBS Toeslagen.

Voor de actuele software raadpleeg source control: <https://github.com/yannicklansink/central-search-engine-toeslagen>

Voor een beeld van de chat app heeft de auteur een video gemaakt waarbij hij door de technische implicaties loopt van de app: <https://www.youtube.com/watch?v=Gk66Wk0aq-c>

1. Functioneel ontwerp

In een dynamische en complexe organisatie zoals de Belastingdienst is de toegang tot accurate en actuele informatie essentieel voor het effectief uitvoeren van taken. Integratie Business Services (IBS) Toeslagen, onderdeel van de Belastingdienst, kampte met versnipperde informatiebronnen (zoals verschillende repositories, wiki's en chatkanalen) waarin gegevens verspreid lagen.

Deze applicatie biedt een innovatieve oplossing door gebruik te maken van een chatgerichte gebruikersinterface bovenop semantische zoektechnologie, vector-gebaseerde opslag en automatische documentanalyse. Medewerkers kunnen hun vragen in gewone taal aan de chat stellen, waarna de applicatie relevante informatie ophaalt en direct beknopte antwoorden genereert. De inzet van geavanceerde AI-technologieën, zoals een Large Language Model (LLM), stelt medewerkers in staat om sneller relevante informatie te vinden en direct antwoord te krijgen op specifieke vragen. Dit bevordert niet alleen de productiviteit, maar verhoogt ook de tevredenheid van gebruikers doordat het de werkdruk vermindert en frustratie tegengaat. Traditioneel kostte het vinden van de juiste beleidsinformatie veel zoekwerk; met de chatinterface wordt die zoektijd drastisch gereduceerd. Uit recent onderzoek blijkt bijvoorbeeld dat AI-assistenten de productiviteit van werknemers aanzienlijk kunnen verhogen (Golden, 2023), en meer dan de helft van de organisaties verwacht dat generatieve AI-zoeksystemen de efficiëntie en productiviteit verder zullen verbeteren (NV & StockTitan, 2024).

De functionele eisen van de chatapplicatie zijn direct herleidbaar tot de behoeften van de gebruikers en onderbouwd met best practices uit literatuur- en praktijk onderzoek. Uit gesprekken en interviews met teamleden is gebleken dat zij behoefte hebben aan een intuïtieve, vraag-gebaseerde interface waarmee informatie snel gevonden kan worden. Daarom is gekozen voor een chatgerichte gebruikersinterface. Onderzoek toont aan dat een conversatiegerichte benadering het zoeken naar informatie efficiënter en gebruiksvriendelijker maakt; medewerkers kunnen in natuurlijke taal vragen stellen en krijgen precieze antwoorden zonder zelf te hoeven speuren in documentatie (Moesker, 2025).

1.1 Praktijk- en literatuuronderzoek voor kernfunctionaliteit

Een case study bij een onderzoeksorganisatie liet zien dat een AI-gedreven zoekoplossing met zo'n conversatie-interface de productiviteit significant verhoogde en ervoor zorgde dat waardevolle kennis niet langer onvindbaar bleef binnen de organisatie (Moesker, 2025). Dit bevestigt dat onze gekozen functionaliteit – een chat waarin de gebruiker gewoon vragen kan typen – aansluit op een reële gebruikersbehoefte en aantoonbaar tijd bespaart.

Daarnaast is er praktijkonderzoek gedaan binnen IBS Toeslagen. Door middel van gestructureerde en ongestructureerde interviews, gebruikerstests en demonstratiesessies met het development team (team Vos) zijn belangrijke inzichten verkregen. De feedback heeft geleid tot gerichte aanpassingen en verbeteringen in de applicatie, wat resulteerde in een hogere gebruiksvriendelijkheid en effectiviteit. Hieronder per kernfunctionaliteit hoe deze tot stand is gekomen.

1.1.1 Document Upload Functionaliteit – Kernfunctionaliteit 1

Uit interviews met gebruikers bleek dat zij vaak informatie uit eigen documenten willen gebruiken bij het beantwoorden van vragen. De mogelijkheid om documenten te uploaden (FR1.1) werd hierdoor als essentieel beschouwd. Dit stelt gebruikers in staat om hun eigen kennisbronnen toe te voegen, wat de relevantie en nauwkeurigheid van de gegenereerde antwoorden verhoogt.

Daarnaast kwam uit feedback naar voren dat gebruikers behoefte hadden aan ondersteuning voor verschillende bestandsformaten, zoals .txt, .pdf en .html (FR1.2). Dit werd geïmplementeerd om de flexibiliteit en bruikbaarheid van de applicatie te verbeteren. Een ander aandachtspunt was het verstrekken van visuele feedback na het uploaden van documenten, aangezien gebruikers onzeker waren over de status van hun uploads. Dit werd opgelost door een duidelijke statusindicator toe te voegen (FR1.3). Deze verbeteringen zorgen ervoor dat de document upload functionaliteit naadloos in het werkproces past: gebruikers kunnen nu eigen documenten eenvoudig integreren in de zoekomgeving, met onmiddellijke bevestiging, wat de gebruiksvriendelijkheid ten goede komt.

1.1.2 Realtime Vraag-Antwoord API – Kernfunctionaliteit 2

Uit tests en demonstratiesessies bleek dat gebruikers sterk waarde hechtten aan snelheid en nauwkeurigheid bij het verkrijgen van antwoorden op hun vragen. De real-time streaming van antwoorden (FR2.2) werd als bijzonder nuttig ervaren, omdat het de interactie met de applicatie verbeterde en gebruikers direct inzicht gaf in de voortgang van de verwerking.

Gebruikers benadrukten tevens de noodzaak van een hoge nauwkeurigheid van de antwoorden (FR2.3). Dit werd bereikt door geavanceerde AI-modellen en een verbeterde zoekfunctionaliteit toe te passen, wat gebruikers hielp om snel relevante informatie te vinden.

1.1.3 Slimme Document Zoekfunctie – Kernfunctionaliteit 3

Tijdens het evaluatieproces gaven teamleden aan dat zij vaak zoeken naar specifieke informatie zonder exacte trefwoorden te kennen. Daarom werd de semantische zoekfunctionaliteit (FR3.1) als een cruciale verbetering gezien. Hierdoor kunnen gebruikers informatie vinden op basis van conceptuele overeenkomsten in plaats van exacte zoektermen, wat de nauwkeurigheid en relevantie van de resultaten verhoogt (FR3.2).

1.1.4 Automatische Documentanalyse met Embeddings – Kernfunctionaliteit 4

De mogelijkheid om documenten automatisch te analyseren en te verwerken werd toegevoegd op basis van gebruikersbehoeften die uit de tests naar voren kwamen. Documenten worden automatisch geanalyseerd bij het uploaden (FR4.1) en embeddings worden gegenereerd (FR4.2), waardoor de informatie efficiënter kan worden verwerkt en doorzocht. Dit verbeterde de prestaties van de applicatie aanzienlijk en zorgde voor een meer gestroomlijnde gebruikerservaring.

Gebruikers merkten op dat de opslag van deze embeddings efficiënt moest gebeuren om de snelheid van de applicatie te waarborgen. Daarom werd ervoor gekozen om embeddings in goed georganiseerde chunks op te slaan (FR4.3), wat zowel de prestaties als de schaalbaarheid van de applicatie ten goede kwam.

1.1.5 Persoonlijke Chatervaring met Geschiedenis – Kernfunctionaliteit 5

Tijdens het evaluatieproces werd duidelijk dat gebruikers het prettig vinden om eerder gevoerde gesprekken terug te kunnen lezen. Daarom werd een functie toegevoegd waarmee chatgeschiedenis lokaal wordt opgeslagen (FR5.2). Gebruikers waardeerden deze functie omdat het hen in staat stelde om eenvoudig terug te grijpen op eerdere antwoorden en vervolgvragen te stellen zonder informatie opnieuw te hoeven invoeren.

Privacy en beveiliging werden hierbij als belangrijke randvoorwaarden meegenomen. De chatgeschiedenis wordt lokaal en veilig opgeslagen (FR5.3), zodat gevoelige informatie niet wordt

blootgesteld aan onbeheerde externe diensten. Dit sluit aan bij de richtlijnen die door de security specialist werden opgesteld tijdens de evaluatiefase.

Gedurende de verschillende fasen van het ontwikkelingsproces werd gebruikersfeedback actief verwerkt. Technische en functionele verbeteringen, zoals de optimalisatie van de uploadfunctionaliteit en de implementatie van geavanceerde zoekalgoritmes, droegen bij aan een betere prestaties en een vloeiendere interactie met het systeem.

De ontvangen feedback bevestigt dat de functionele keuzes aansluiten op reële gebruikersbehoeften en praktijkervaringen. De evaluaties onderstrepen dat de applicatie niet alleen technische vooruitgang biedt, maar ook daadwerkelijk bijdraagt aan een efficiëntere en gebruiksvriendelijkere werkomgeving binnen IBS Toeslagen. Dit vormt een solide basis voor verdere doorontwikkeling en optimalisatie van het systeem in toekomstige iteraties.

1.2 Functionele Requirements

Document Upload Functionaliteit

- **FR1.1:** Gebruikers moeten in staat zijn om documenten te uploaden via de frontend interface. Dit stelt gebruikers in staat om relevante informatie beschikbaar te maken voor verdere verwerking en analyse.
- **FR1.2:** De applicatie moet verschillende bestandsformaten ondersteunen, zoals .txt, .pdf en .html. Dit zorgt voor flexibiliteit en gebruiksgemak voor de gebruikers.
- **FR1.3:** Na het uploaden moet de gebruiker een bevestiging ontvangen dat het document succesvol is geüpload. Dit biedt duidelijkheid en zekerheid aan de gebruiker.

Realtime Vraag-Antwoord API

- **FR2.1:** Gebruikers moeten vragen kunnen stellen via de frontend interface. Dit maakt interactie met de applicatie mogelijk en verbetert de gebruikerservaring.
- **FR2.2:** De applicatie moet real-time antwoorden kunnen geven op basis van de geüploade documenten. Dit zorgt voor snelle en relevante informatieverstrekking.
- **FR2.3:** De antwoorden moeten relevant en nauwkeurig zijn. Dit verhoogt de betrouwbaarheid en bruikbaarheid van de applicatie.

Slimme Document Zoekfunctie

- **FR3.1:** De zoekfunctionaliteit moet semantische overeenkomsten kunnen vinden, zelfs als trefwoorden niet exact overeenkomen. Dit verbetert de nauwkeurigheid en relevantie van de zoekresultaten.
- **FR3.2:** De zoekresultaten moeten relevant en nauwkeurig zijn. Dit verhoogt de tevredenheid en efficiëntie van de gebruikers.

Automatische Documentanalyse met Embeddings

- **FR4.1:** Bij het uploaden moeten documenten automatisch worden geanalyseerd. Dit zorgt voor een efficiënte en effectieve verwerking van documenten.

- **FR4.2:** De applicatie moet embeddings genereren met behulp van AI (OpenAI). Dit verbetert de mogelijkheden voor verdere analyse en zoekfunctionaliteit.
- **FR4.3:** De gegenereerde embeddings moeten effectief worden opgeslagen in chunks. Dit zorgt voor een efficiënte opslag en verwerking van embeddings.

Persoonlijke Chatervaring met Geschiedenis

- **FR5.1:** Gebruikers moeten chatgesprekken kunnen voeren via de frontend interface. Dit verbetert de interactie en gebruikerservaring.
- **FR5.2:** De chatgeschiedenis moet lokaal worden opgeslagen zodat gebruikers deze kunnen terugkijken. Dit verhoogt de bruikbaarheid en functionaliteit van de applicatie.
- **FR5.3:** De chatgeschiedenis moet veilig en privé worden opgeslagen. Dit zorgt voor de privacy en veiligheid van de gebruikersgegevens.

1.3 Gebruikersscenario: chatapplicatie in de praktijk

Onderstaand scenario illustreert hoe een medewerker van IBS Toeslagen de chatapplicatie gebruikt in haar dagelijkse werk. We volgen een medewerker terwijl zij een complexe vraag moet beantwoorden en laten zien hoe de applicatie haar probleem oplost. Hierbij komen de kernfunctionaliteiten samen: de medewerker vindt direct relevante beleidsinformatie zonder handmatig documenten te hoeven doorzoeken, wat haar werkdruk verlaagt en frustratie voorkomt.

Scenario:

Thomas is een nieuwe developer bij IBS Toeslagen en maakt deel uit van team Vos. Zijn eerste opdracht is om een wijziging aan te brengen in een bestaande microservice die gerelateerd is aan de verwerking van toeslagaanvragen. Hoewel hij ervaring heeft met microservices en API's, is de architectuur van IBS Toeslagen complex, met tientallen services verspreid over verschillende teams. Om te begrijpen hoe zijn wijziging impact heeft op het systeem, moet Thomas eerst nagaan welke componenten betrokken zijn en wie de verantwoordelijke ontwikkelaars zijn. Normaal gesproken zou hij een senior developer moeten vragen of door honderden pagina's aan documentatie en Confluence-wiki's moeten zoeken, een tijdrovend en inefficiënt proces.

Vandaag besluit Thomas de chatapplicatie te gebruiken. Hij navigeert naar de Chat-interface en stelt direct zijn vraag: *"Welke service is verantwoordelijk voor het valideren van toeslagaanvragen?"* De chatapplicatie doorzoekt razendsnel de interne architectuurdocumentatie en API-specificaties die eerder door het team zijn geüpload. Dankzij de semantische zoekfunctie herkent de AI dat Thomas eigenlijk op zoek is naar de "Toeslag Validatie Service", een microservice die validatieregels toepast voordat een aanvraag wordt goedgekeurd. Binnen enkele seconden verschijnt het antwoord op zijn scherm:

"De Toeslag Validatie Service is verantwoordelijk voor de validatie van toeslagaanvragen. De service controleert of alle vereiste gegevens aanwezig zijn en of de aanvraag voldoet aan de wetgeving. Dit wordt beheerd door team Hippo."

Naast deze uitleg toont de applicatie ook direct de relevante architectuurdocumentatie en een link naar de API-specificaties, zodat Thomas verder kan lezen als hij meer details nodig heeft.

Omdat Thomas niet alleen wil weten wat de service doet, maar ook wie er verantwoordelijk voor is, stelt hij een vervolgvraag: *"Wie is de contactpersoon voor deze service?"* De chatapplicatie herkent

dat hij op zoek is naar de verantwoordelijke developers en toont direct een overzicht met de huidige eigenaar en het team:

“De Toeslag Validatie Service wordt beheerd door team Hippo. De huidige contactpersoon is Jeroen van Dijk (Tech Lead).”

Hierdoor hoeft Thomas geen senior developer te storen met basisvragen over teamverantwoordelijkheden; hij weet direct bij wie hij moet zijn.

Vervolgens wil Thomas weten in welk groter systeem deze service past. Hij stelt een nieuwe vraag: *“Welke andere services werken samen met de Toeslag Validatie Service?”* Dankzij de integratie met de systeemdokumentatie toont de chatapplicatie een overzicht van de gekoppelde services, inclusief een grafische weergave van de interacties tussen componenten:

“De Toeslag Validatie Service communiceert met de Toeslag Berekening Service en de Burgergegevens Service. Dit betekent dat wijzigingen in de validatieregels mogelijk invloed hebben op de verwerking en berekening van toeslagen.”

Deze informatie bespaart Thomas uren aan zoekwerk in interne documentatie en voorkomt dat hij fouten maakt bij het aanpassen van de code.

Dankzij de chatapplicatie heeft Thomas binnen enkele minuten het antwoord op drie essentiële vragen over de service waarmee hij moet werken. Hij hoeft geen senior developer lastig te vallen met basisvragen en kan zelfstandig de informatie vinden die hij nodig heeft. Hierdoor begrijpt hij sneller hoe het IBS Toeslagen IT-landschap in elkaar zit en kan hij eerder productief bijdragen aan het team.

Wat voorheen uren aan zoekwerk of meerdere gesprekken met collega's zou kosten, is nu binnen minuten opgelost. De chatapplicatie versnelt het leerproces, vermindert de werkdruk van ervaren collega's en helpt nieuwe developers sneller effectief te worden in hun werk.

1.4 Use case diagrammen

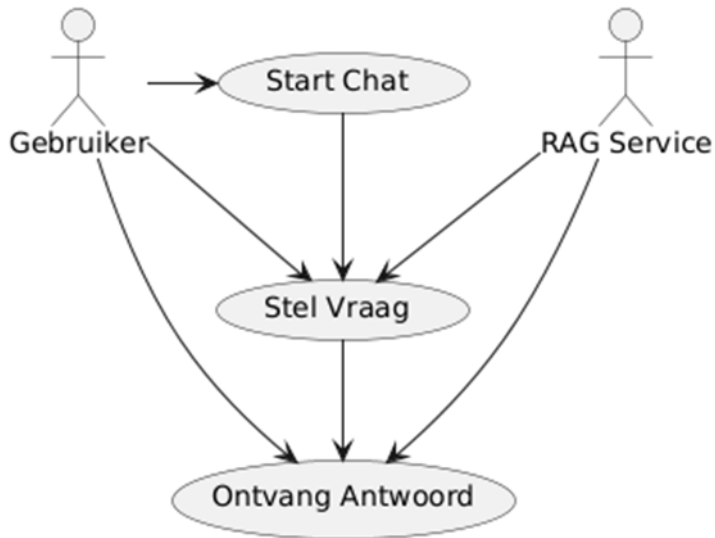
In dit hoofdstuk worden de belangrijkste interacties binnen de chatapplicatie geanalyseerd aan de hand van use case diagrammen. De diagrammen geven een visuele representatie van de kernfunctionaliteiten en de betrokken actoren. Deze beschrijving helpt bij het inzichtelijk maken van de wijze waarop gebruikers en systemen interageren om de gewenste functionaliteit te realiseren.

1.4.1 Chat interactie

De chatinteractie is een van de primaire functionaliteiten van de applicatie en stelt gebruikers in staat om vragen te stellen aan de RAG service en hierop relevante antwoorden te ontvangen. De betrokken actoren binnen deze interactie zijn:

- **Gebruikers IBS Toeslagen:** Dit zijn de eindgebruikers die via de frontend vragen stellen en antwoorden ontvangen. Ze initiëren de interactie door de chat te starten en sturen vragen in natuurlijke taal.
- **RAG Service:** Dit is de backend-component van de applicatie die verantwoordelijk is voor het verwerken van de vragen en het genereren van relevante antwoorden. De service maakt gebruik van AI-modellen om informatie uit documenten en andere kennisbronnen te halen.

Het proces begint wanneer een gebruiker een chatsessie start en een vraag stelt. De vraag wordt doorgestuurd naar de RAG-service, die vervolgens een antwoord genereert op basis van de beschikbare data. Zodra de verwerking is voltooid, ontvangt de gebruiker een antwoord in de chatinterface. Dit proces is in real-time en is ontworpen om snel en accuraat relevante informatie te verstrekken.



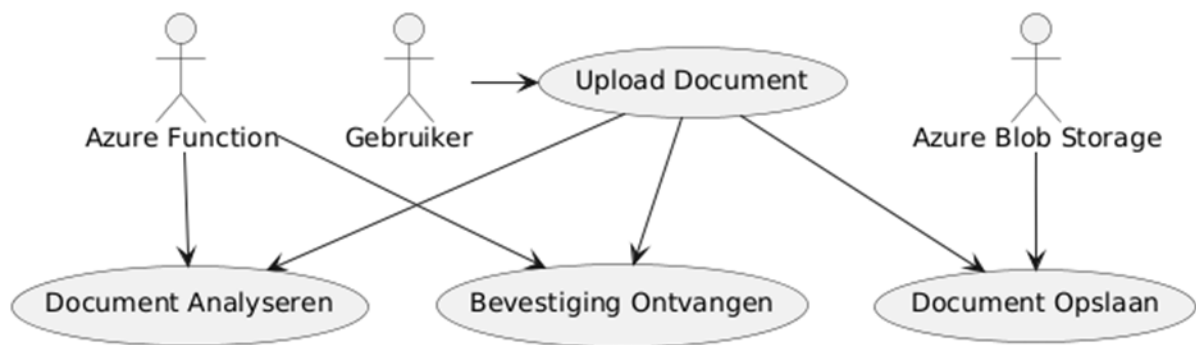
Afbeelding 1.4.1 use case diagram chat interactie

1.4.2 Upload interactie

De uploadfunctionaliteit is essentieel voor het verrijken van de kennisbasis van de applicatie, doordat gebruikers documenten kunnen uploaden die vervolgens worden geanalyseerd en opgeslagen. De actoren binnen deze use case zijn:

- **Gebruikers IBS Toeslagen:** De gebruikers uploaden documenten via de frontend-interface, wat essentieel is voor het toevoegen van nieuwe informatie aan de applicatie.
- **Azure Function:** Dit is een serverless component dat verantwoordelijk is voor het verwerken en analyseren van geüploade documenten. Het kan geavanceerde bewerkingen uitvoeren, zoals het extraheren van tekst en het genereren van embeddings.
- **Azure Blob Storage:** Dit systeem slaat de geüploade documenten op en zorgt ervoor dat ze beschikbaar blijven voor verdere verwerking en raadpleging door de RAG-service.

Het proces begint wanneer een gebruiker een document uploadt. Dit document wordt vervolgens doorgestuurd naar Azure Function voor analyse. Tegelijkertijd wordt het bestand opgeslagen in Azure Blob Storage. Na de analyse ontvangt de gebruiker een bevestiging van een succesvolle upload en verwerking. De gegenereerde metadata en embeddings worden vervolgens beschikbaar gesteld aan de zoekfunctionaliteit en de RAG-service, waardoor de accuraatheid en volledigheid van de antwoorden verbetert.



Afbeelding 1.4.2 use case diagram upload interactie

1.5 Wireframe

De visuele representatie van de chatapplicatie is een essentieel onderdeel van het functioneel ontwerp en draagt bij aan een intuïtieve gebruikerservaring. In deze sectie worden de wireframes van de verschillende schermen beschreven, waarbij de nadruk ligt op de gebruikersinteractie en functionaliteit.

1.5.1 Schets van het home scherm

De homepage fungeert als de introductie van de applicatie en biedt gebruikers een overzicht van de mogelijkheden. Dit scherm verwelkomt de gebruiker met een korte beschrijving van de applicatie en een embedded demonstratievideo. De navigatie bevindt zich aan de linkerkant van het scherm, waar gebruikers kunnen schakelen tussen de homepage, de chatinterface en de uploadpagina. De opzet is minimalistisch, met een duidelijke focus op gebruiksvriendelijkheid.

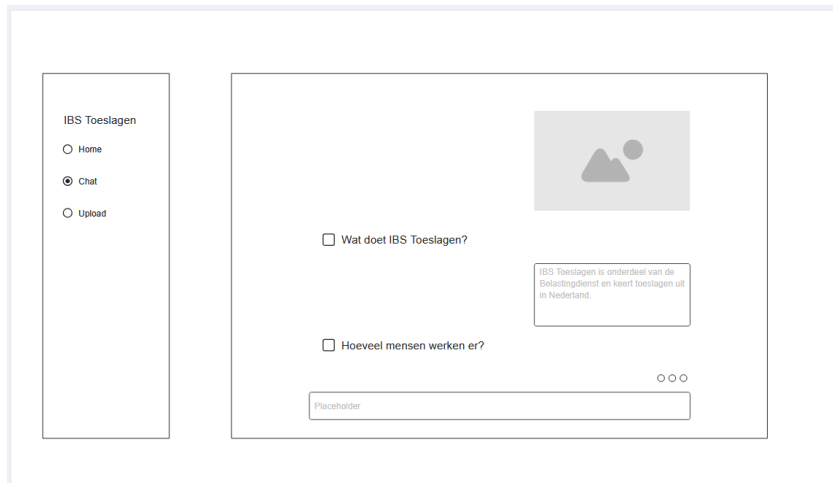


Afbeelding 1.5.1 Home scherm wireframe

1.5.2 Schets van de chatinterface

De chatinterface vormt het kernonderdeel van de applicatie en stelt gebruikers in staat om vragen te stellen en direct antwoorden te ontvangen. De interface is ontworpen met een gestructureerde layout, waarin de vragen en antwoorden overzichtelijk worden weergegeven. Gebruikers kunnen hun vragen invoeren in een tekstveld, waarna de applicatie het antwoord genereert. Bovenaan de

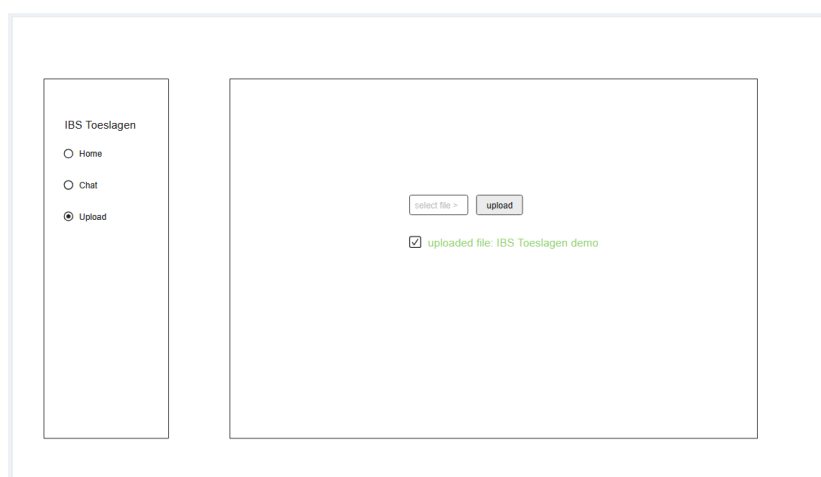
chatinterface is ruimte voor relevante suggesties, die gebruikers kunnen aanvinken om snel informatie op te vragen. Daarnaast wordt er een visuele weergave van de RAG-service gepresenteerd, zodat gebruikers inzicht krijgen in de herkomst van de antwoorden.



Afbeelding 1.5.2 Chat functionaliteit wireframe

1.5.3 Schets van de uploadinterface

De uploadpagina biedt gebruikers de mogelijkheid om documenten toe te voegen aan de applicatie, waardoor de antwoorden die de chatinterface genereert relevanter en specifieker worden. De interface is eenvoudig en functioneel: gebruikers selecteren een bestand, uploaden dit naar het systeem en ontvangen een bevestiging zodra de upload succesvol is voltooid. Daarnaast wordt de status van de upload duidelijk weergegeven, zodat gebruikers weten of het document correct is verwerkt. De integratie met Azure Blob Storage zorgt ervoor dat de documenten veilig worden opgeslagen en later kunnen worden gebruikt door de RAG-service.



Afbeelding 1.5.3 Upload functionaliteit wireframe

1.6 Technologiystack

De gekozen technologiystack voor dit project is zorgvuldig samengesteld om de gewenste functionaliteiten te ondersteunen en naadloos op elkaar aan te laten sluiten. Hieronder de belangrijkste componenten en tools met uitleg:

- Programmeertaal: Python.
- Web Framework: FastAPI – voor het ontwikkelen van de REST API backend.
- Frontend Framework: Blazor WebAssembly – voor het bouwen van de interactieve webapplicatie (single-page webapplicatie) die in de browser draait.
- Vector Database: Pinecone – specialistische database voor het opslaan en snel doorzoeken van vectorrepresentaties (embeddings) van documenten.
- Embeddings Service: OpenAI API – gebruikt voor het genereren van embeddings van tekst en het uitvoeren van de vraag-antwoordsuggesties via een LLM.
- Cloud Storage: Azure Blob Storage – voor het opslaan van geüploade documenten (binaries) en eventuele tussenresultaten.
- Dependency Management: Poetry (en requirements.txt) – voor het beheren van Python-dependenties en pakketten in het project.

Deze stack is gekozen om een betrouwbare applicatie te kunnen bouwen. Zo biedt FastAPI snelle API-responses en eenvoudige integratie met Python's AI-libraries, terwijl Blazor WebAssembly een responsieve gebruikerservaring in de browser mogelijk maakt zonder pagina-refreshes. Pinecone en OpenAI vormen samen het hart van de AI-functionaliteit: Pinecone is in staat om miljoenen tekstfragmenten als vector te indexeren en binnen milliseconden semantisch vergelijkbare content te vinden, en OpenAI's LLM zorgt voor hoogkwalitatieve interpretatie van vragen en formulering van antwoorden. Door Azure Blob Storage en andere Azure-componenten in te zetten, wordt bovendien ingespeeld op de enterprise-omgeving van IBS Toeslagen, waar schaalbaarheid, integratie met bestaande infrastructuur en veiligheid voorop staan.

1.7 Architectuur

De architectuur van de applicatie bestaat uit verschillende belangrijke componenten en hun interacties. Hieronder een overzicht van de kernonderdelen en hun rol binnen het systeem:

FastAPI: Voor het opzetten van de API-server en het afhandelen van binnenkomende verzoeken. Dit is de backend die de brug vormt tussen de frontend (chatinterface) en de AI-logica. Bij elke vraag van de gebruiker zorgt FastAPI ervoor dat deze correct wordt doorgegeven aan de verwerkingslogica en dat het antwoord teruggestuurd wordt.

Pinecone: Voor het opslaan en doorzoeken van vectorrepresentaties van documenten (de embeddings). Pinecone fungeert als de vector database. Dankzij geavanceerde zoekalgoritmen en indexstructuren kan Pinecone semantische overeenkomsten vinden tussen een vraag en documentfragmenten, zelfs in een zeer grote dataset aan documenten. Dit betekent dat ongeacht de omvang van de kennisbasis, relevante informatie razendsnel gevonden wordt op basis van betekenis in plaats van letterlijke trefwoorden.

OpenAI: Voor het genereren van embeddings én het beantwoorden van vragen. OpenAI's modellen worden gebruikt om van iedere tekst (document of vraag) een numerieke vector (embedding) te maken die de betekenis representeert.

Azure Blob Storage: Voor het opslaan van de geüploade documenten. Elke keer wanneer een gebruiker een document uploadt via de applicatie, wordt dit bestand veilig opgeslagen in Blob Storage (in een afgeschermd container). Deze cloud-opslag is schaalbaar en betrouwbaar, wat betekent dat zelfs grote of meerdere documenten tegelijk kunnen worden verwerkt.

LangChain: Voor het opzetten van de chain of thought en de sequentie van bewerkingen die nodig zijn om een antwoord te genereren. LangChain is een bibliotheek die helpt bij het orchestreren van de verschillende stappen in het RAG-proces: van het ontvangen van de vraag, het ophalen van relevante embeddings uit Pinecone, tot het samenstellen van een prompt voor het LLM en het post-processen van het antwoord. Door LangChain te gebruiken worden deze stappen modulair en beheersbaar opgezet, wat ontwikkeling en onderhoud vereenvoudigt.

Het samenspel van deze componenten zorgt ervoor dat de applicatie aan de functionele eisen voldoet: FastAPI + Blazor vormen de UI/API laag voor communicatie, Pinecone + OpenAI + LangChain vormen de intelligente kern voor zoeken en genereren, en Azure Blob Storage + Functions verzorgen de documentopslag en -verwerking.

1.8 Third-Party Integraties

Naast de eigen ontwikkelde componenten maakt de applicatie gebruik van een aantal externe diensten (third-party) die geïntegreerd zijn voor specifieke functionaliteit:

Azure Cloud Services: met name Azure Blob Storage voor het betrouwbaar opslaan van bestanden (documentuploads) in de cloud, en Azure Functions voor serverless processing. De keuze voor Azure sluit aan bij de bestaande infrastructuur van de Belastingdienst en biedt schaalbaarheid en security op enterprise-niveau.

Pinecone: als externe gehoste vector database service. Door Pinecone te integreren hoeft niet zelf een complexe vectorindex van scratch gebouwd te worden; in plaats daarvan wordt een bewezen dienst gebruikt die hoge prestaties levert voor semantisch zoeken in grote hoeveelheden data. Pinecone draait in de cloud en wordt benaderd via een API-key, maar is zo ingesteld dat alle data encrypted en geïsoleerd is, conform de veiligheidsvereisten.

OpenAI API: de AI-dienstverlener voor taalmodellen en embeddings. Via de OpenAI API wordt toegang verkregen tot krachtige pretrained modellen (zoals GPT-3.5/4) zonder dat de organisatie zelf zware AI-infrastructuur hoeft te hosten. Integratie vindt plaats middels beveiligde API-aanroepen. OpenAI's service is gekozen vanwege de goede prestaties op gebied van natuurlijke taal. Dit is cruciaal om gebruikersvragen goed te begrijpen en contextuele antwoorden te genereren.

Elke integratie is zorgvuldig afgewogen op basis van betrouwbaarheid, snelheid en beveiliging. Door gebruik te maken van deze third-party services de auteur zich focussen op de businesslogica en gebruikersinterface.

1.9 API Specificaties RAG-service

De applicatie stelt een API ter beschikking die door de frontend wordt gebruikt om vragen te versturen en documentuploads te verwerken. Voor de functionele werking zijn met name de volgende endpoints van belang:

- GET / – Redirect naar de API-documentatie.

Dit endpoint dient enkel voor om ontwikkelaar naar de gedetailleerde API documentatie te leiden. Het is geen inhoudelijk onderdeel van de functionaliteit, maar een hulp voor integratie en debugging. Wanneer dit endpoint in de browser wordt geopend, wordt de gebruiker doorgestuurd naar een pagina waar alle endpoints beschreven staan. (In een productiecontext zal de eindgebruiker dit niet direct benutten; het is voornamelijk voor ontwikkelaars.)

- POST /RAG/stream – Verwerkt een vraag en streamt het antwoord terug naar de client.

Dit is het kern-endpoint van de backend. Wanneer de gebruiker in de frontend een vraag stelt, wordt deze via een POST-verzoek naar dit endpoint gestuurd (in JSON-formaat, met de vraagtekst en eventueel een conversation ID of context). De server (FastAPI backend) handelt dit verzoek als volgt af:

1. De vraagtekst wordt ontvangen en via de integratie met Pinecone wordt een semantische search uitgevoerd op de vector database om de meest relevante documenten/passages (embeddings) te vinden die passen bij de vraag.
2. Met die contextuele informatie wordt een prompt opgebouwd en naar het OpenAI LLM gestuurd, dat een antwoord genereert.
3. Het antwoord wordt in stukken (streaming) teruggestuurd naar de client zodra de tokens binnenkomen, zodat de gebruiker niet hoeft te wachten tot het hele antwoord gereed is.

Dit endpoint zorgt er dus voor dat de gehele vraag-antwoord-keten georkestreerd wordt. Het streaming-mechanisme is belangrijk om de frontend ervaring vloeiend te houden.

Het is goed om te weten dat de upload functionaliteit niet via de RAG-service verloopt. Dit betreft namelijk een upload vanaf de frontend naar Azure Blob storage direct.

1.10 Database Structuur

De gegevens die de applicatie gebruikt en produceert, worden op verschillende plaatsen opgeslagen, afhankelijk van het type data en de vereisten qua toegangssnelheid en veiligheid. De database-architectuur bestaat uit:

Documenten: Geüploade documenten (bijv. PDF's of teksten) worden in Azure Blob Storage bewaard, en wel voor een tijdelijke duur. Deze tijdelijke opslag is nodig zodat de Azure Function ze kan verwerken. Na succesvolle verwerking (d.w.z. extraheren van tekst en aanmaken van embeddings) kan het document uit de blob storage worden verwijderd om opslagruimte te besparen en te voldoen aan eventuele compliance-eisen. Het document blijft maximaal 30 minuten liggen voordat het verwijderd wordt.

Embeddings: De gegenereerde vectorrepresentaties van documenten (de tekstinhoud in numerieke vorm) worden opgeslagen in Pinecone (de vector database). Elke chunk tekst uit een document correspondeert met één vector. Naast de vector zelf wordt metadata bijgehouden (zoals document-ID, paginanummer, bronverwijzing) zodat later nog duidelijk is uit welk document een gevonden

fragment afkomstig is. Pinecone vormt hiermee de feitelijke kennisbank van de applicatie: het is geoptimaliseerd om zeer snel deze vectors te doorzoeken op semantische overeenkomsten met een gebruikersvraag. Dankzij deze structuur kan het systeem binnen fracties van een seconde door honderden of duizenden pagina's aan tekst zoeken, puur op inhoudsgelijkenis.

Chatgeschiedenis: De conversatiehistorie (vragen en antwoorden van de gebruiker) wordt in de browser's localStorage van de gebruiker opgeslagen (dus client-side). Dit betekent dat de geschiedenis aan de clientkant bewaard blijft: telkens als de gebruiker de applicatie heropent in dezelfde browser, kan de applicatie de vorige gesprekken uit localStorage laden en tonen. Deze ontwerpkeuze is bewust gemaakt om privacyredenen – er wordt geen centrale opslag van chatcontent bijgehouden op de server. Het voordeel is dat gevoelige informatie uit chats (bijvoorbeeld casus-specifieke details die de gebruiker noemt) niet op de server of in de cloud wordt bewaard, waardoor risico op datalekken via de applicatie nihil is.

1.11 Azure Function

Azure Functions worden gebruikt om serverless computing mogelijk te maken, wat schaalbaarheid en kostenbesparing biedt (Ggailey, 2023). In deze applicatie is een Azure Function ingezet voor specifieke achtergrondtaken, met name het verwerken van geüploade bestanden. Het idee is dat er automatisch een stukje code draait zodra een document geüpload wordt, zonder dat een permanente server daarvoor hoeft te draaien.

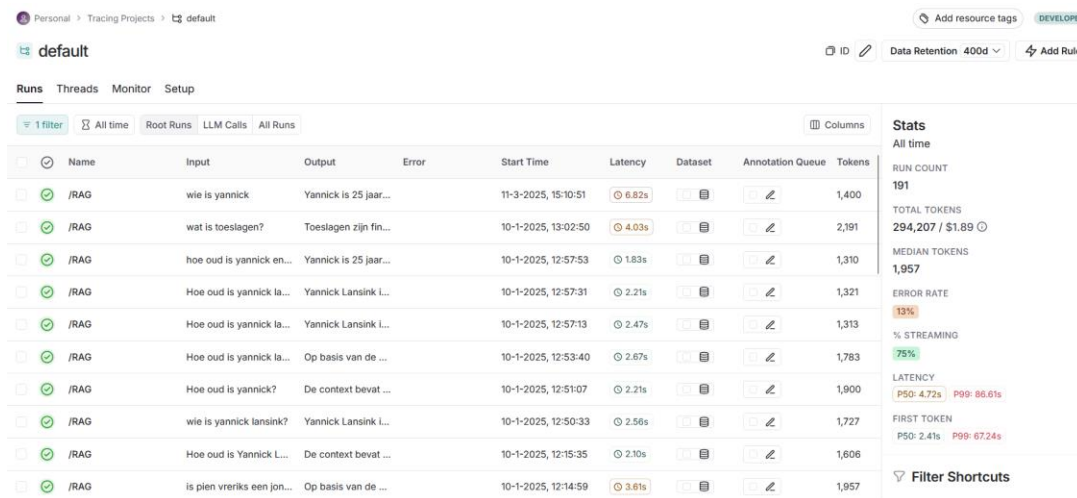
Werking upload-verwerking: De Azure Function wordt geactiveerd door een trigger uit Azure Blob Storage. Wanneer een bestand door de gebruiker naar een specifieke container in Blob Storage wordt geüpload (bijvoorbeeld de container "pending-documents"), detecteert Azure dit evenement en roept het de vooraf geregistreerde Function aan. Deze function voert vervolgens de nodige stappen uit: het haalt het bestand op uit Blob Storage, extraheert de tekst en stuurt deze tekst door naar de component die embeddings genereert (OpenAI). De resulterende embeddings worden daarna rechtstreeks via de Pinecone API in de vector database opgeslagen, inclusief relevantie metadata. Tenslotte kan de Azure Function, na succesvolle opslag in Pinecone, het originele documentbestand uit de blob storage verwijderen of verplaatsen naar een archief-locatie.

Door deze aanpak verloopt het uploadproces volledig automatisch en asynchroon: de gebruiker krijgt snel een uploadbevestiging en hoeft niet te wachten op alle analyse. De zware verwerking gebeurt losgekoppeld, en dankzij de serverless schaalbaarheid kan Azure Function meerdere uploads gelijktijdig aan als dat nodig is, zonder dat daarvoor een aparte server ingesteld hoeft te worden. Bovendien betaalt IBS Toeslagen alleen voor de rekentijd die daadwerkelijk gebruikt wordt tijdens zo'n verwerking (kosten efficiënt). Dit is een van de business doelstellingen van het project.

1.12 Monitoring en Logging

Een robuust en schaalbaar AI-gedreven systeem vereist continue monitoring en logging om de betrouwbaarheid, prestaties en foutdiagnose te waarborgen. In het kader van de ontwikkelde Retrieval-Augmented Generation (RAG)-service wordt gebruikgemaakt van LangSmith, een gespecialiseerde tool voor het loggen en monitoren van interacties met een Large Language Model (LLM). LangSmith faciliteert een diepgaande analyse van API-aanroepen, tracking van prestatie-indicatoren en foutdetectie, waardoor zowel reactief als preventief onderhoud mogelijk wordt gemaakt.

Door een geautomatiseerde monitoringstrategie te implementeren, wordt de stabiliteit van de applicatie gewaarborgd, de responstijd geoptimaliseerd en kunnen afwijkingen vroegtijdig worden opgespoord en opgelost. Dit draagt direct bij aan de gebruikservaring van medewerkers binnen IBS Toeslagen, die afhankelijk zijn van snelle en accurate zoekresultaten.



Name	Input	Output	Error	Start Time	Latency	Dataset	Annotation Queue	Tokens
/RAG	wie is yannick	Yannick is 25 jaar...		11-3-2025, 15:10:51	6.82s			1,400
/RAG	wat is toeslagen?	Toeslagen zijn fin...		10-1-2025, 13:02:50	4.03s			2,191
/RAG	hoe oud is yannick en...	Yannick is 25 jaar...		10-1-2025, 12:57:53	1.83s			1,310
/RAG	Hoe oud is yannick la...	Yannick Lansink L...		10-1-2025, 12:57:31	2.21s			1,321
/RAG	Hoe oud is yannick la...	Yannick Lansink L...		10-1-2025, 12:57:13	2.47s			1,313
/RAG	Hoe oud is yannick la...	Op basis van de ...		10-1-2025, 12:53:40	2.67s			1,783
/RAG	Hoe oud is yannick?	De context bevat ...		10-1-2025, 12:51:07	2.21s			1,900
/RAG	wie is yannick lansink?	Yannick Lansink L...		10-1-2025, 12:50:33	2.56s			1,727
/RAG	Hoe oud is Yannick L...	De context bevat ...		10-1-2025, 12:15:35	2.10s			1,606
/RAG	is pien vrekis een jon...	Op basis van de ...		10-1-2025, 12:14:59	3.61s			1,957

Afbeelding 1.11 Logging in langsmith

LangSmith is een tool die wordt gebruikt voor het loggen en monitoren van API-aanroepen naar een LLM. Het helpt bij het bijhouden van de keten van bewerkingen die worden uitgevoerd om een antwoord te genereren. Hierdoor kunnen ontwikkelaars beter inzicht krijgen in de prestaties en betrouwbaarheid van hun LLM-integraties.

Voorbeeld: Het loggen van verzoeken, antwoorden, en fouten tijdens de verwerking wordt gevisualiseerd en geanalyseerd.

1.13 Samenvatting

In dit functioneel ontwerp is een volledig overzicht gegeven van de doelstellingen, functionaliteiten, technologieën, architectuur, API-specificaties, databasestructuur, en de inzet van Azure Functions en monitoring voor de RAG-chatapplicatie. De beschreven componenten en processen vormen samen een robuuste en schaalbare oplossing die voldoet aan de behoeften van IBS Toeslagen.

Belangrijker nog, dit ontwerp laat zien waarom deze oplossing effectief is: door versnipperde informatie te centraliseren in een slimme chatinterface, kunnen medewerkers van IBS Toeslagen aanzienlijk sneller de juiste informatie vinden. De semantische zoektechnologie en realtime AI-antwoorden zorgen ervoor dat gebruikers niet langer zelf uren hoeven te zoeken of bladeren; in plaats daarvan krijgen ze in seconden de kennis die ze nodig hebben. Dit verlaagt de zoekdruk en vermindert de werkdruk en frustratie bij het uitvoeren van hun taken, terwijl de algehele efficiëntie en kwaliteit van het werk toeneemt. Daarmee draagt de chatapplicatie direct bij aan een productievere, meer tevreden en beter geïnformeerde workforce binnen IBS Toeslagen – een investering die zich terugbetaalt in consistente en snelle dienstverlening.

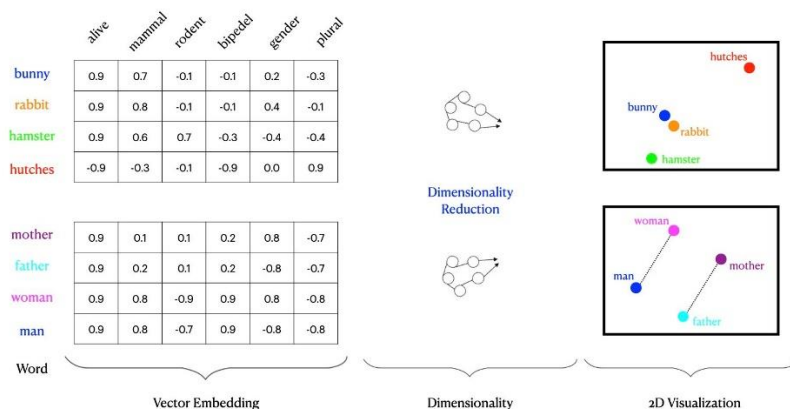
2. Technisch ontwerp

De gekozen tech stack is ontworpen om een efficiënte, schaalbare en veilige chatapplicatie te ondersteunen die medewerkers van Integratie Business Services (IBS) Toeslagen helpt bij het vinden van informatie. De tech stack omvat zowel frontend- als backend-technologieën, hostingoplossingen en beveiligingsmaatregelen om een robuuste en gebruiksvriendelijke applicatie te leveren.

2.1 Terminologie en architectuurcontext

In het technisch ontwerp document worden vaktermen nu duidelijk gedefinieerd *voordat* ze gebruikt worden, zodat de lezers de technische details beter kunnen volgen. Zo wordt gesproken over chunks en embeddings in het kader van de vector database. Chunks zijn de kleinere tekstfragmenten waarin documenten worden opgedeeld. Voordat een document in de kennisbank (vector database) wordt opgeslagen, wordt het document eerst opgesplitst in hapklare brokken tekst – deze fragmenten noemen we chunks. Het gebruik van chunks zorgt ervoor dat ook specifieke paragrafen of zinnen uit een document teruggevonden kunnen worden zonder het hele document in één keer te hoeven doorzoeken. (*What Are Vector Embeddings? Applications, Use Cases & More* | DataStax, 2025)

Vervolgens wordt elke chunk omgezet in een embedding: dit is een wiskundige representatie (vector) van de inhoud van zo'n tekstfragment. Een embedding vangt de semantische betekenis van tekst in numerieke vorm. Met andere woorden, elke chunk wordt door een AI-model vertaald naar een reeks getallen (een punt in een meerdimensionale ruimte) die de essentie en context van dat tekstfragment vertegenwoordigt. Chunks met vergelijkbare inhoud krijgen embeddings die dicht bij elkaar liggen in die vectorruimte. Door documenten te “chunken” en elk deel als embedding in de vector database op te slaan, kan de applicatie bij een vraag van de gebruiker razendsnel semantisch relevante stukken informatie vinden – zelfs als er niet letterlijk hetzelfde trefwoord in staat. Hieronder een afbeelding hoe een embedding bestaat uit een reeks getallen en hoe dat dan visueel eruit ziet in 2D.



Afbeelding 2.1.1 (McLane, 2023)

Denk aan de vector database als een bibliotheek die op betekenis sorteert: elke embedding is als het ware een label dat aangeeft waar het fragment over gaat. Wanneer de gebruiker een vraag stelt,

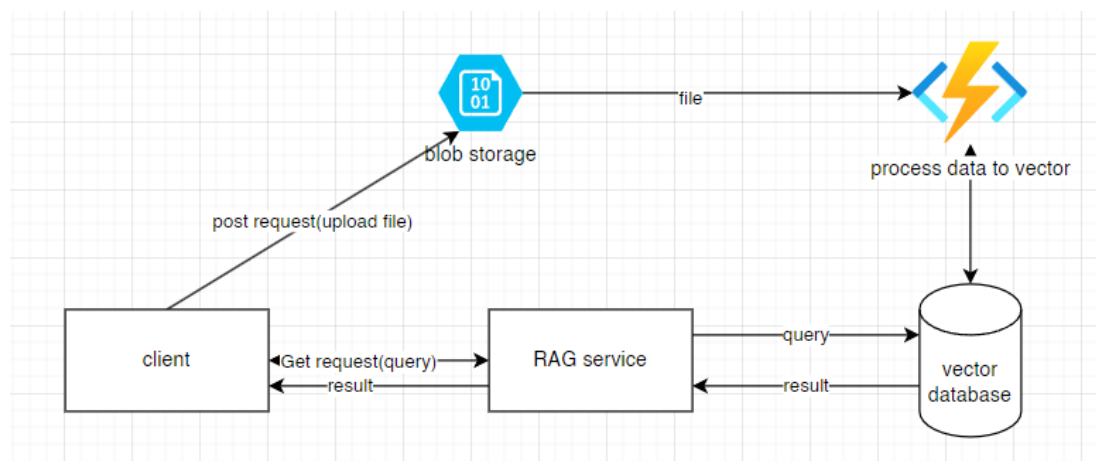
wordt ook die vraag omgezet in een embedding, en de vector database zoekt naar fragmenten (embeddings) die hier dicht bij in de buurt liggen qua betekenis.

Doordat we deze termen nu helder hebben gedefinieerd, is het duidelijk hoe de techniek werkt: chunks maken gericht zoeken mogelijk, en embeddings zorgen dat dit zoeken gebaseerd is op betekenis in plaats van puur trefwoorden. Dit draagt bij aan een beter begrip van de verdere technische uitwerking.

Om het technische overzicht te verbeteren, begint het technisch ontwerp nu met een contextdiagram (zie afbeelding 2.1.2 hieronder) dat de applicatie in haar omgeving plaatst. In dit diagram is te zien hoe de componenten onderling en met elkaar samenwerken. Centraal in het contextdiagram staat de *RAG-service* (Retrieval Augmented Generation -service) met daaromheen de belangrijkste interacties: de client (gebruiker) die via hier met de RAG-service communiceert, de koppeling van de RAG-service met de Vector Database (extern beheerde database voor embeddings), en de aanroep van de Azure Function voor documentverwerking vanuit de blob storage.

Het contextdiagram laat bijvoorbeeld zien dat de gebruiker via de web-frontend een vraag stelt aan de RAG-service; de RAG-service zoekt vervolgens data in de vector DB en vraagt het LLM om een antwoord. Bij het uploaden van een document wordt er een trigger aangetrapt vanuit Azure function om de blob storage uit te lezen. Deze data wordt vervolgens door de function als embedding opgeslagen in de vector DB.

Door dit overzichtelijke contextdiagram aan het begin te plaatsen, is meteen duidelijk hoe de chatapplicatie als geheel in elkaar steekt en welke externe factoren een rol spelen. Stakeholders en ontwikkelaars krijgen hiermee een hoog over beeld van de architectuur voordat in de hier opvolgbare paragrafen de details van elke component worden besproken.



Afbeelding 2.1.2 contextdiagram chatapplicatie

2.2 Technische requirements

2.2.1 Functionele technische requirements

Documentbeheer en slimme opslag

- **TR1.1:** Het systeem moet gebruikers in staat stellen om bestanden te uploaden via de frontend, waarbij ondersteuning wordt geboden voor meerdere bestandsformaten en robuuste foutafhandeling.

- **TR1.2:** Geüploade bestanden moeten efficiënt en veilig worden opgeslagen in een cloud-gebaseerde opslagomgeving.
- **TR1.3:** Het systeem moet een maximale bestandsgrootte van 10 MB afdwingen en duidelijke foutmeldingen tonen bij overschrijding.
- **TR1.4:** Documenten moeten versleuteld worden opgeslagen en alleen toegankelijk zijn voor geautoriseerde gebruikers.
- **TR1.5:** Geüploade bestanden moeten tijdelijk beschikbaar blijven en automatisch worden verwijderd na 1 uur. De opslag moet voldoen aan industriestandaarden voor encryptie en beveiliging tijdens transport en opslag.

Realtime vraag-antwoord API

- **TR2.1:** Het systeem moet API-endpoints bevatten die vragen van gebruikers kunnen verwerken en een antwoord kunnen genereren.
- **TR2.2:** De API moet efficiënt kunnen omgaan met grote hoeveelheden data en complexe queries, met een focus op prestaties en schaalbaarheid.
- **TR2.3:** Het systeem moet ondersteuning bieden voor het doorzoeken van documentinhoud op basis van semantische overeenkomsten.
- **TR2.4:** Antwoorden moeten in real-time naar de gebruiker worden teruggestuurd, met een lage latentie en hoge betrouwbaarheid.

Slimme document zoekfunctie

- **TR3.1:** Gebruikers moeten documenten kunnen doorzoeken via een zoekfunctie in de frontend met snelle responstijden en een intuïtieve gebruikersinterface.
- **TR3.2:** De zoekfunctie moet semantisch relevante en nauwkeurige resultaten opleveren, ongeacht de exacte formulering van de zoekopdracht.
- **TR3.3:** Het systeem moet geavanceerde zoekopdrachten ondersteunen op basis van documentinhoud en context.

Automatische documentanalyse met embeddings

- **TR4.1:** Na het uploaden van een document moet het systeem automatisch een analyse uitvoeren en relevante metadata genereren.
- **TR4.2:** De documentanalyse moet efficiënt en accuraat verlopen, ongeacht het bestandsformaat.
- **TR4.3:** De gegenereerde documentrepresentaties moeten kunnen worden opgeslagen en snel kunnen worden opgehaald voor zoekopdrachten.
- **TR4.4:** De kwaliteit en nauwkeurigheid van gegenereerde documentrepresentaties moeten gewaarborgd worden door geavanceerde AI-modellen.
- **TR4.5:** Documenten moeten zo worden verwerkt dat zowel opslag als zoekopdrachten efficiënt en schaalbaar blijven.

- **TR4.6:** Het systeem moet mechanismen bevatten om bestandsnamen te controleren en duplicaten correct af te handelen, zodat de meest recente versie van een bestand behouden blijft.

Persoonlijke Chatervaring met geschiedenis

- **TR5.1:** Gebruikers moeten via een chatinterface vragen kunnen stellen en antwoorden ontvangen.
- **TR5.2:** Het systeem moet de chatgeschiedenis lokaal kunnen opslaan en herstellen bij herladen van de applicatie.
- **TR5.3:** De chatgeschiedenis moet correct worden beheerd, inclusief synchronisatie en foutafhandeling.
- **TR5.4:** Opslag en ophalen van de chatgeschiedenis moeten efficiënt en veilig plaatsvinden.

2.2.2 Niet-functionele technische requirements

NFR1. Beveiliging & Privacy: De applicatie moet bedrijfsgevoelige informatie veilig verwerken en opslaan. Er mag geen data onbeveiligd of onnodig naar derde partijen worden gestuurd. Dit betekent bijvoorbeeld dat documenten en gegenereerde embeddings versleuteld worden opgeslagen en dat communicatie met de LLM-service en vector database via encryptie plaatsvindt. Ook dient de oplossing te voldoen aan relevante wet- en regelgeving (zoals de AVG) voor dataverwerking.

NFR2. Prestatie: De applicatie moet snel reageren. Gebruikersvragen dienen binnen enkele seconden beantwoord te worden. Concreet wordt gestreefd naar een responstijd van <5 seconden voor een gemiddeld zoekproces, zodat medewerkers vlot antwoord krijgen en niet afhaken. Dit stelt eisen aan de efficiëntie van de zoekalgoritmen in de vector database en de snelheid van de LLM-respons.

NFR3. Schaalbaarheid: De architectuur moet schaalbaar zijn zodat een groei in het aantal gebruikers of in de hoeveelheid data niet leidt tot significante performance degradatie. De keuze voor cloud-componenten zoals Azure Functions ondersteunt dit: Azure Functions kunnen automatisch opschalen bij hogere belasting, waardoor de applicatie bij toenemend gebruik stabiel en responsief blijft (Ggailey, 2024). Eveneens moet de vector database schaalbaar zijn (horizontaal uitbreidbaar) om grote volumes aan documenten en embeddings te kunnen blijven doorzoeken met lage latentie.

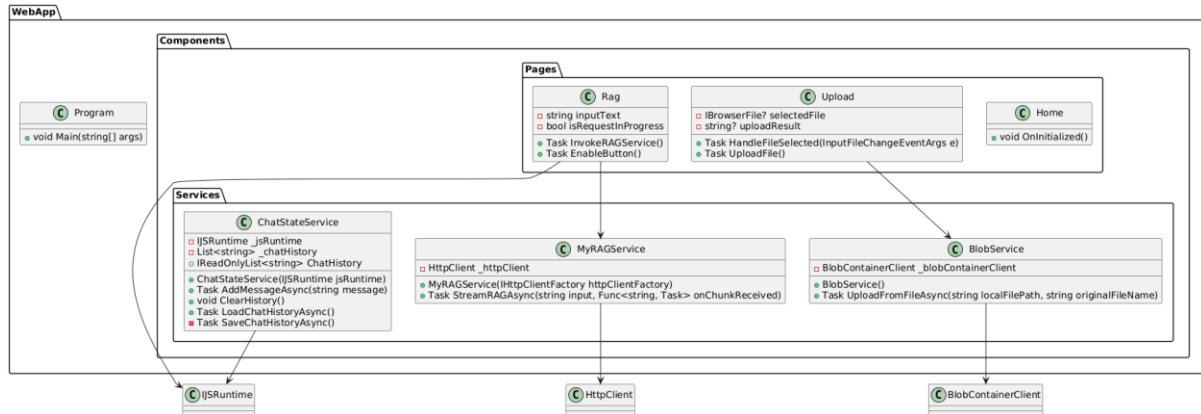
NFR4. Integratie & Compatibiliteit: De oplossing moet naadloos integreren in de bestaande IBS Toeslagen infrastructuur. Dat houdt in dat de applicatie binnen het interne netwerk draait of via de geautoriseerde cloud-omgeving, zodat authenticatie (bijv. Single Sign-On) en autorisatie van gebruikers overeenkomt met de bedrijfsstandaarden. Externe API-sleutels en endpoints (voor LLM en vector DB) moeten configurabel zijn, zodat de applicatie ook met andere maar vergelijkbare services zou kunnen werken indien nodig.

NFR5. Onderhoudbaarheid & Codekwaliteit: De codebase moet goed onderhoudbaar zijn. Dit wordt bereikt door een modulair ontwerp en het toepassen van codeerstandaarden. Concreet betekent dit dat logica gescheiden is per component (bijv. aparte modules/klassen voor communicatie met de vector DB, voor aansturing van het LLM, enzovoort), en dat er duidelijke documentatie en comments aanwezig zijn. Ook moet het eenvoudig zijn om onderdelen te vervangen of te upgraden (bijvoorbeeld een ander taalmodel integreren) zonder het hele systeem te herschrijven. Versiebeheer (Git) en testautomatisering dienen te worden toegepast om de kwaliteit bij wijzigingen te bewaken.

2.3 Technische infrastructuur

2.3.1 Frontend

UML Class diagram van de frontend



Afbeelding 2.3.1 class diagram frontend

2.3.1.1 Uitleg van klassen

Het klassendiagram (afbeelding 2.3.1) maakt niet alleen de individuele klassen inzichtelijk, maar ook hun onderlinge relaties en hoe deze samenwerken om de frontend-functionaliteit te realiseren. In de frontend-architectuur zijn de relaties opgezet volgens het principe van scheiding der verantwoordelijkheden (Wikipedia contributors, 2025b): UI-componenten (pagina's) maken gebruik van service-klassen voor specifieke taken, zonder dat ze die taken zelf hoeven te implementeren.

Een duidelijke relatie is bijvoorbeeld te zien tussen de pagina-klassen en de service-klassen. De RAG (chatpagina) component werkt nauw samen met zowel `MyRAGService` als `ChatStateService`. Wanneer een gebruiker een vraag stelt via de RAG-interface, roept RAG intern de methode `InvokeRAGService()` aan, die gebruikmaakt van de `MyRAGService` om de vraag door te sturen naar de backend en om het antwoord stapsgewijs op te halen. Stapsgewijs houdt in via streaming om de gebruiker direct een antwoord te geven. Gedurende dit proces ontvangt de RAG-klasse via de callback van `MyRAGService` elk deel van het antwoord en voegt dit (samen met de oorspronkelijke vraag) toe aan de chatgeschiedenis door `ChatStateService.AddMessageAsync(...)` aan te roepen. Op deze manier composeert de RAG-pagina de functionaliteit van twee services: `MyRAGService` voor externe communicatie en `ChatStateService` voor lokale opslag van het gesprek. Het resultaat is dat de gebruiker vloeiend een vraag-antwoord dialoog kan voeren, waarbij de interface realtime wordt bijgewerkt en de geschiedenis bewaard blijft.

Een vergelijkbare samenwerking bestaat tussen de `Upload`-pagina en de `BlobService`. De `Upload`-klasse vertrouwt op `BlobService` om de technische details van het bestand-uploaden af te handelen. Wanneer de gebruiker op de uploadknop klikt (waarmee `UploadFile()` wordt gestart), zal `Upload` de methode `BlobService.UploadFromFileAsync(...)` aanroepen en het geselecteerde bestand en bestandsnaam doorgeven.

Hierdoor besteedt de UI-component de feitelijke opslagoperatie uit aan de `BlobService`, die weet hoe om te gaan met de Azure Blob Storage API. De relatie tussen `Upload` en `BlobService` is typisch een associatie waarin `Upload` een instantie van `BlobService` gebruikt (via dependency injection). Dankzij

deze relatie hoeft de Upload-pagina zich niet bezig te houden met de implementatiedetails van de cloudopslag en kan hij zich richten op gebruikersinteractie (zoals het ophalen van het te uploaden bestand en het tonen van meldingen).

Naast deze hoofdrelaties is ook de connectie met het hoofdprogramma van belang. De Program-klasse installeert en verstrekt de service-objecten aan de pagina's. In het UML-klassendiagram (afbeelding 2.3.1) zou dit kunnen worden weergegeven als afhankelijkheden: de pagina-klassen (Home, Rag, Upload) zijn afhankelijk van bepaalde services die door Program worden geïnitieerd. Bijvoorbeeld, RAG en Upload krijgen respectievelijk toegang tot MyRAGService/ChatStateService en BlobService. Deze vorm van losse koppeling betekent dat de pagina's niet zelf nieuwe service-objecten aanmaken, maar ze aangereikt krijgen.

Onderling hebben de service-klassen in de frontend typisch weinig directe relaties met elkaar, wat weer past bij hun afzonderlijke verantwoordelijkheden (Wikipedia contributors, 2025b). ChatStateService, BlobService en MyRAGService opereren onafhankelijk: de ene service hoeft niet te weten hoe de andere intern werkt. Zij komen pas samen in de context van de UI componenten die ze gebruiken. Dit is een kenmerk van een goed gelaagde frontend architectuur: de presentatie-laag (UI) orkestreert de samenwerking tussen services, terwijl de services zelf herbruikbaar en onafhankelijk blijven.

Uit het klassendiagram blijkt ook dat de front-end klassen samen een coherent geheel vormen voor de functionaliteit van de chatapplicatie. De Home-pagina staat losser en heeft geen sterke afhankelijkheden, wat logisch is gezien zijn informatieve rol.

2.3.1.2 Technologische basis

Programeertaal

- **C#**

Frameworks:

- Blazor WebAssembly: Voor het bouwen van interactieve webapplicaties met behulp van C# en .NET.

Styling:

- CSS: Voor het stylen van de gebruikersinterface.
- Bootstrap: Voor het gebruik van kant-en-klare stijlen en componenten.

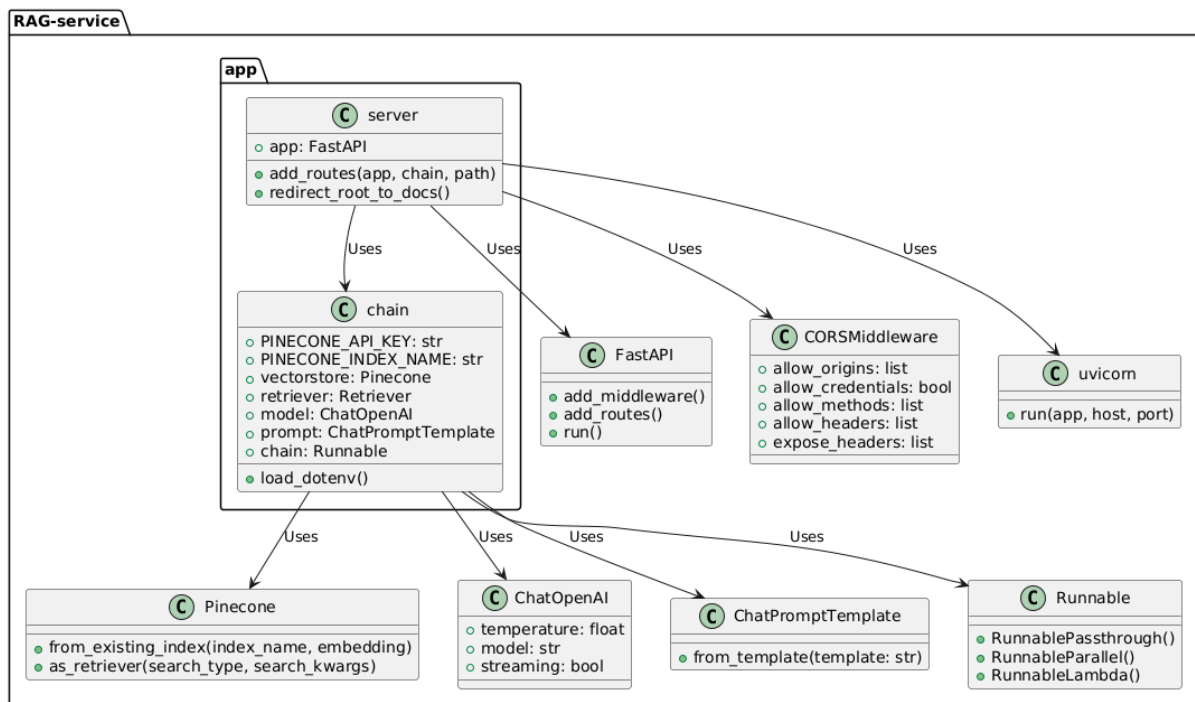
Build Tools:

- .NET SDK: Voor het bouwen en publiceren van de Blazor WebAssembly applicatie.

2.3.2 Backend

Hieronder vind je een klassendiagram van de RAG-service. De RAG-service vormt de kern van de backend van de IBS Toeslagen chatapplicatie en is verantwoordelijk voor de verwerking van gebruikersinvoer en de gegenereerde antwoorden op basis van een Retrieval-Augmented Generation (RAG) architectuur. De implementatie maakt gebruik van een combinatie van neurale netwerken en vectorgebaseerde zoektechnologieën om relevante informatie op te halen en contextbewuste antwoorden te genereren. De belangrijkste architecturale componenten bevinden zich binnen de

RAG-service directory en omvatten klassen en methoden die bijdragen aan de functionaliteit van de applicatie.



Afbeelding 2.3.2 class diagram backend service (rag-service)

2.3.2.1 Technische uitleg backend klassendiagram

De basis van de backend wordt gevormd door de module `app.chain`, waarin de initiële configuratie van de RAG-pipeline wordt vastgelegd. Hier worden omgevingsvariabelen geladen, zoals de `PINEcone_API_KEY` en `PINEcone_INDEX_NAME`, die nodig zijn voor het verbinden met de vector-database Pinecone. De vectorstore wordt geïnstantieerd met behulp van Pinecone, waarbij de retriever wordt geconfigureerd om relevante documenten uit de vectorindex te halen. De klasse maakt verder gebruik van een `ChatOpenAI`-model, dat verantwoordelijk is voor het genereren van responsen op basis van een vooraf gedefinieerde `ChatPromptTemplate`. Deze componenten worden gecombineerd in een `Runnable`-pipeline, die de input van de gebruiker verwerkt, documenten ophaalt en het gegenereerde antwoord retourneert.

Binnen deze architectuur speelt Pinecone een cruciale rol als vectorgebaseerde opslag- en zoektechnologie. Door middel van de methode `from_existing_index` kan een bestaand index worden geladen en wordt een retriever gecreëerd via `as_retriever`, waarbij zoekparameters zoals `search_type` en `search_kwargs` worden gespecificeerd. Dit zorgt ervoor dat alleen de meest relevante documenten uit de vectorruimte worden geselecteerd als input voor de taalmodellen.

De taalmodellering en conversatiegeneratie worden verzorgd door de `ChatOpenAI`-klasse, die een direct koppelpunt vormt met het OpenAI-taalmodel. Dit model wordt geconfigureerd met parameters zoals `temperature`, waarmee de mate van variatie in gegenereerde antwoorden wordt beïnvloed, en `streaming`, waarmee real-time gegenereerde tekst wordt doorgegeven aan de frontend. Het `ChatPromptTemplate`-object fungeert als een dynamisch sjabloon voor de

gegenereerde prompts, wat bijdraagt aan de coherentie en controleerbaarheid van de gegenereerde antwoorden.

De verwerking van de input-output stroom wordt gecoördineerd door de Runnable-klasse. Dit is een abstractie die verschillende uitvoerbare processen beheert. Binnen de implementatie worden verschillende uitvoeringsmodellen gebruikt, zoals RunnablePassthrough, RunnableParallel en RunnableLambda. RunnablePassthrough stuurt de input direct door zonder transformatie, RunnableParallel voert meerdere berekeningen tegelijkertijd uit en RunnableLambda biedt de mogelijkheid om aangepaste functies als uitvoerbare eenheden te definiëren.

De hoofdserverside wordt beheerd door de module `app.server`, waarin de FastAPI-webserver wordt gedefinieerd. FastAPI is verantwoordelijk voor het afhandelen van HTTP-verzoeken en het aanbieden van de RAG-functionaliteiten aan de frontend. De serverapplicatie wordt geïntanceerd als een FastAPI-object en routes worden dynamisch toegevoegd met de methode `add_routes(app, chain, path)`. Dit zorgt ervoor dat alle interacties met de backend via een goed gedefinieerde API-interface verlopen.

De server wordt uiteindelijk gestart door het aanroepen van Uvicorn, een efficiënte server die is geoptimaliseerd voor snelle en asynchrone verwerking van webverzoeken. De methode `run(app, host, port)` activeert de webservice en maakt deze toegankelijk voor client-applicaties.

2.3.2.2 Technologische basis

Programmeertaal

- **Python**

Frameworks

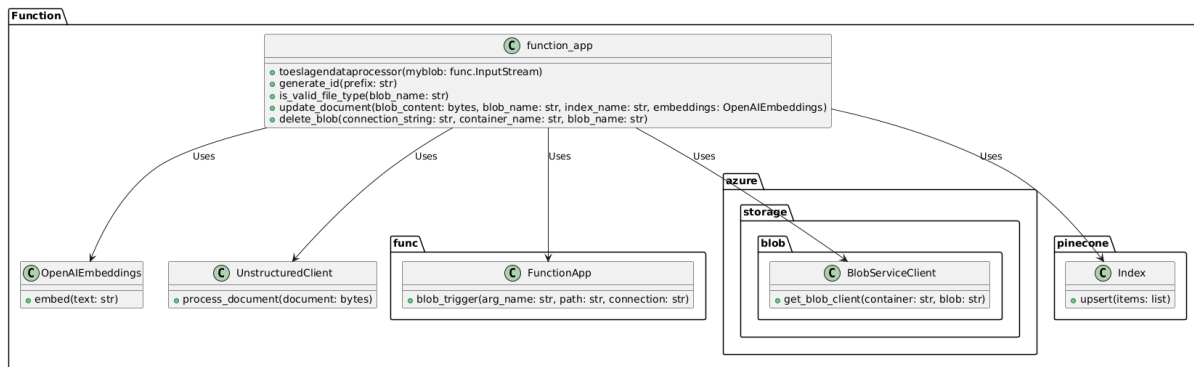
- **FastAPI:** Voor het opzetten van de API-server en het verwerken van verzoeken.
- **LangChain:** Voor het opzetten van de keten van bewerkingen die nodig zijn om een antwoord te genereren.
- **Uvicorn:** ASGI-server voor het draaien van de FastAPI applicatie.

Database

- **Pinecone:** Voor het opslaan en doorzoeken van vectorrepresentaties van documenten.

2.3.3 Azure Functie

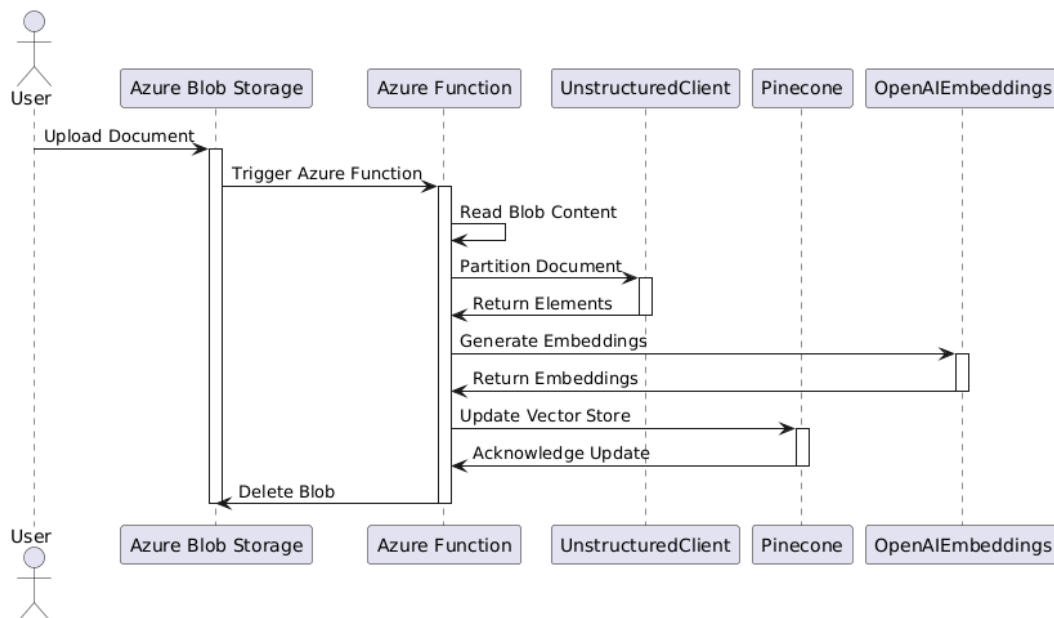
De Azure Function binnen de ontwikkelde chatapplicatie speelt een cruciale rol als onderdeel van de backend-architectuur en is ontworpen om real-time documentverwerking te faciliteren. De primaire functie van dit component is om documenten die door gebruikers via de frontend worden geüpload automatisch te verwerken. Dit draagt direct bij aan de efficiëntie en bruikbaarheid van de Retrieval Augmented Generation (RAG) service, een kernonderdeel van de applicatie, zoals beschreven in het beroepsproduct van het project.



Afbeelding 2.3.3 klassendiagram Azure Function

Centraal in deze architectuur staat de klasse `function_app`. Deze klasse implementeert alle methoden die nodig zijn om documentuploads effectief en efficiënt te verwerken. Bij het uploaden van documenten door gebruikers naar Azure Blob Storage wordt automatisch een trigger aangeroepen door de klasse `func.FunctionApp`. Deze trigger, geïmplementeerd via de methode `blob_trigger(arg_name: str, path: str, connection: str)`, detecteert automatisch wanneer een nieuw bestand is toegevoegd aan een specifieke blob-container in Azure Storage.

De verwerking van een nieuw document begint met het lezen van de inhoud uit Azure Blob Storage. De `azure.storage.blob.BlobServiceClient`-klasse wordt hiervoor aangeroepen om een verbinding te maken met de Blob Storage en de data van het geüploade bestand te verkrijgen. Zodra de inhoud van het bestand beschikbaar is, wordt deze verder verwerkt met behulp van de methode `update_document(blob_content: str)` in de `function_app`-klasse. Dit proces omvat het analyseren en partitioneren van de documenten met behulp van de externe dienst `UnstructuredClient`. Zie afbeelding 2.3.4 hierin staat hoe het verloop is als de function wordt gebruikt.



Afbeelding 2.3.4 sequence diagram azure function

Na deze initiële verwerking worden de documenten door de methode `update_blob` automatisch opgesplitst in kleinere stukken tekst, genaamd chunks, die geschikt zijn voor semantische analyse. Dit proces is essentieel omdat de chatapplicatie gebaseerd is op Retrieval Augmented Generation (RAG), waarbij stukken tekst semantisch gematcht worden met vragen die gebruikers stellen.

De gegenereerde tekstfragmenten worden vervolgens omgezet naar embeddings met behulp van het AI-model van OpenAI via de `OpenAIEmbeddings`-klasse. Deze embeddings zijn numerieke representaties van de semantische inhoud van de tekstfragmenten en worden gegenereerd door de methode `embed` van deze klasse. Deze vectorrepresentaties worden vervolgens opgeslagen in de vector database Pinecone via de `Pinecone Index`-klasse. Dit maakt gebruik van de methode `update_blob` om de embeddings toe te voegen via het zogenaamde "upsert"-proces, waarbij bestaande data wordt bijgewerkt of nieuwe data wordt toegevoegd indien deze nog niet bestaat.

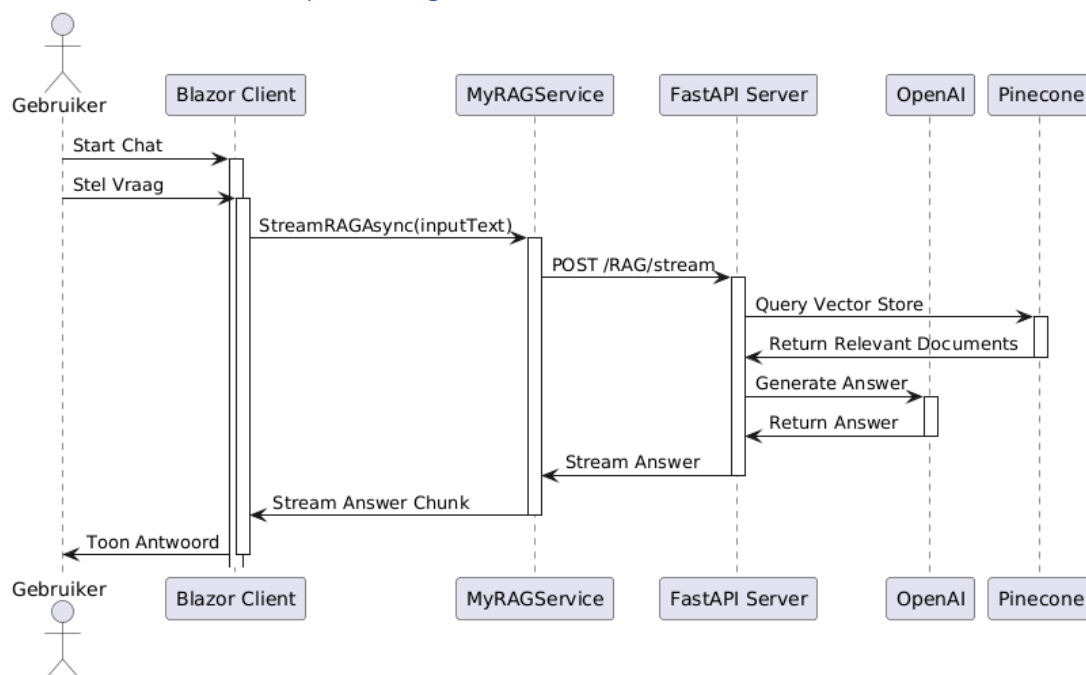
Om gegevensintegriteit en beveiliging te waarborgen, bevat de klasse `function_app` ook methoden zoals `delete_blob`, die zorgt voor het verwijderen van bestanden uit Azure Blob Storage na verwerking. Dit is een belangrijk onderdeel van de applicatie-architectuur, aangezien documenten slechts tijdelijk worden opgeslagen (met een maximale retentie van één uur) om veiligheidsredenen.

Daarnaast omvat het klassendiagram de `FunctionApp`-klasse, die het centrale triggermechanisme implementeert via de methode `blob_trigger`. Dit mechanisme wordt geactiveerd zodra een nieuw bestand in Azure Blob Storage geplaatst wordt, waarna de gehele verwerkingsstroom automatisch begint.

2.3.4 Requirements sequences

Hieronder volgen sequence diagrammen die uitleg geven over hoe de applicatie stroom werkt. Het biedt als input voor de uitwerking naar code. Het omschrijft werkelijke methodes die uitgewerkt dienen te worden en geeft de stroom weer tussen componenten. Er is daarnaast technisch uitgelegd wat elke stap in detail doet.

2.3.4.1 Chat interactie sequence diagram



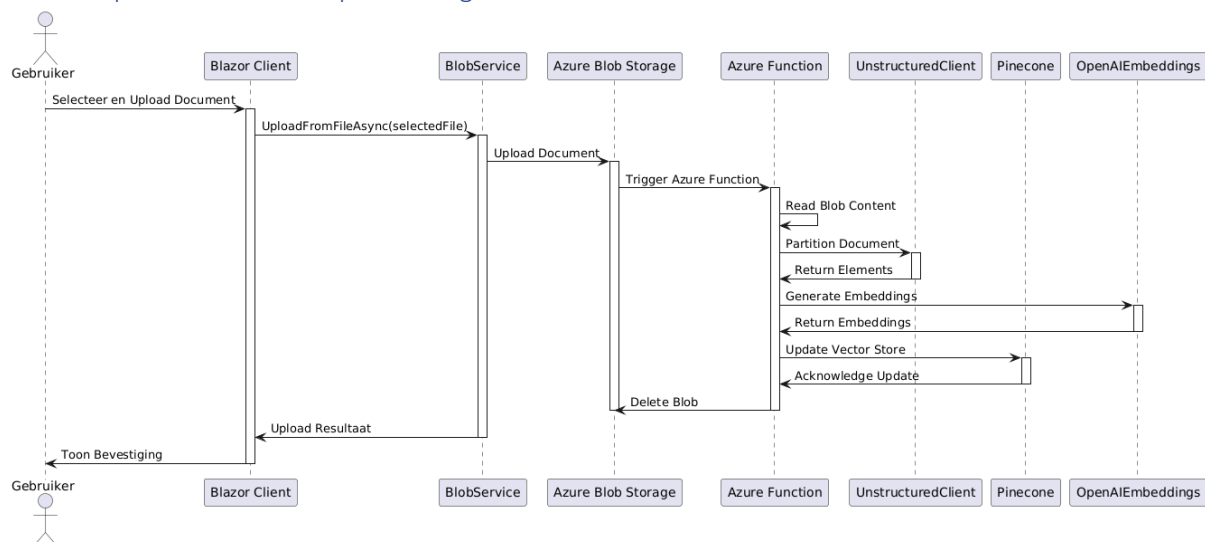
Afbeelding 2.3.4.1 sequence diagram chat interactie

Het sequence diagram (afbeelding 2.3.4.1) van de chatinteractie illustreert de stroom van gegevens en communicatie tussen verschillende componenten binnen de RAG-gebaseerde chatapplicatie. De gebruiker initieert de interactie via de Blazor Client door een chat te starten en een vraag te stellen. Deze invoer wordt via de methode `StreamRAGAsync(inputText)` doorgestuurd naar `MyRAGService`, een service die de communicatie tussen de frontend en backend faciliteert. Vervolgens wordt een HTTP POST-verzoek (`POST /RAG/stream`) gegenereerd en doorgestuurd naar de FastAPI Server, waar de verwerking van de vraag begint.

Binnen de FastAPI Server wordt de vector-gebaseerde opslag bevraagd via Pinecone, een gespecialiseerde database voor semantische zoekopdrachten. Pinecone retourneert de meest relevante documenten op basis van de ingevoerde vraag, waarna deze informatie wordt doorgegeven aan OpenAI. Hier wordt de vraag gecombineerd met de relevante documenten en verwerkt tot een semantisch passend antwoord. Dit antwoord wordt in een gestreamde vorm teruggestuurd naar de FastAPI Server, die de gegevens in segmenten aanbiedt aan `MyRAGService`. Deze service zorgt ervoor dat de antwoordchunks in realtime beschikbaar komen voor de frontend.

De Blazor Client ontvangt de gestreamde antwoordchunks en toont deze direct aan de gebruiker, wat zorgt voor een dynamische en responsieve gebruikerservaring. De keuze voor server-side streaming zorgt ervoor dat de gebruiker onmiddellijk feedback krijgt, waardoor de waargenomen responstijd wordt geminimaliseerd (*Streaming Server-Side Rendering*, z.d.). Dit sequence diagram benadrukt de efficiënte samenwerking tussen de vectorgebaseerde opslag (Pinecone), de taalverwerkingservice (OpenAI) en de backend-infrastructuur (FastAPI).

2.3.4.2 Upload interactie sequence diagram



Het sequence diagram (afbeelding 2.3.4.2) van de uploadinteractie toont de processtroom die wordt doorlopen wanneer een gebruiker een document uploadt naar de chatapplicatie. De interactie begint wederom bij de Blazor Client, waar de gebruiker een document selecteert en een uploadactie initieert. Hierna volgt een aanroep van de methode `UploadFromFileAsync(selectedFile)` binnen de `BlobService`, die verantwoordelijk is voor de communicatie met Azure Blob Storage. Het document wordt vervolgens overgebracht naar de cloudopslag, waarna Azure Blob Storage een uploadbevestiging afgeeft en een Azure Function wordt geactiveerd voor verdere verwerking.

De Azure Function wordt getriggerd zodra een bestand in Azure Blob Storage wordt geplaatst. Deze functie haalt de blob-inhoud op en voert een reeks verwerkingsstappen uit. Het document wordt eerst gechunked in kleinere tekstfragmenten door de `UnstructuredClient`, waarna de resulterende elementen worden geretourneerd. Vervolgens worden embeddings gegenereerd met behulp van de `OpenAIEmbeddings-service`, die de semantische representatie van de tekst omzet in numerieke vectoren. Deze gegenereerde embeddings worden teruggestuurd en opgeslagen in de vectorgebaseerde opslag (Pinecone). Zodra de embeddings correct zijn verwerkt, wordt een bevestiging teruggestuurd naar de Azure Function.

Na succesvolle verwerking wordt het oorspronkelijke document uit Azure Blob Storage verwijderd, conform het retentiebeleid dat de opslagduur van bestanden beperkt. De Blazor Client ontvangt een update van het uploadproces en toont een bevestiging aan de gebruiker, waarmee de uploadprocedure wordt voltooid. Dit sequence diagram illustreert een geoptimaliseerde en geautomatiseerde documentverwerkingsworkflow, waarbij gebruik wordt gemaakt van cloud-gebaseerde opslag, AI-gebaseerde semantische verwerking en vectoropslag.

2.3.5 Beveiliging

Huidige status: Momenteel wordt nog weinig gedaan op het gebied van beveiliging omdat het om een prototype gaat.

Toekomstige maatregelen: In de toekomst worden de volgende maatregelen genomen om ervoor te zorgen dat onze data veilig blijft:

- **Data Encryptie:** Alle data die wordt opgeslagen in Azure Blob Storage en Pinecone wordt versleuteld om ongeautoriseerde toegang te voorkomen.
- **Veilige API-aanroepen:** Gebruik van HTTPS voor alle API-aanroepen om ervoor te zorgen dat de data tijdens het transport wordt versleuteld.
- **Beveiligingsprotocollen:** Implementatie van strikte beveiligingsprotocollen en best practices om ervoor te zorgen dat gevoelige informatie niet toegankelijk is voor onbevoegde partijen.

- **Regelmatige Beveiligingsaudits:** Voeren van regelmatige beveiligingsaudits en penetratietests om potentiële kwetsbaarheden te identificeren en te verhelpen.
- **Gebruik van Secure Credentials:** Gebruik van veilige methoden voor het opslaan en beheren van API-sleutels en andere gevoelige informatie, zoals Azure Key Vault.
- **LLM op eigen servers draaien:** Het draaien van de Language Learning Models (LLM) op eigen servers om data in huis te houden.

2.4 Risicoanalyse

Een risicoanalyse is essentieel om potentiële problemen te identificeren die de voortgang en het succes van het project kunnen beïnvloeden. Hieronder volgt een gedetailleerde risicoanalyse voor de ontwikkeling en implementatie van de RAG-service.

Tabel 2.4 Risicoanalyse

Risico	Impact	Kans van optreden	Beheersmaatregel
Integratieproblemen met Pinecone en OpenAI	Hoog	Middel	Monitor en log de prestaties van de integraties met behulp van Langsmith om problemen vroegtijdig te detecteren en op te lossen.
Schaalbaarheidsproblemen	Hoog	Middel	Optimaliseer de code en gebruik schaalbare cloud-oplossingen zoals Azure Functions en Blob Storage.
Beveiligingslekken	Zeer hoog	Middel	Gebruik voor het prototype geen gevoelige data en implementeer basisbeveiligingsmaatregelen om de impact van eventuele lekken te minimaliseren.
Onvoldoende training van gebruikers	Middel	Hoog	Organiseer een demo bij IBS Toeslagen, waarbij gebruikers Toeslagen breed een voorproefje krijgen van de chatapplicatie. Dit stelt hen in staat om de functionaliteiten te verkennen en vragen te stellen.
Systeem downtime	Hoog	Laag	Redundante systemen, disaster recovery planning

Data kwaliteit	Hoog	Middel	Implementatie van data validatie processen
Weerstand tegen verandering	Middel	Hoog	Change management strategie, stakeholder engagement
Afhankelijkheid van externe vendors	Hoog	Middel	Vendor diversificatie, fall-back scenarios
Budget overschrijding	Hoog	Middel	Strikte projectmonitoring, fase-gewijze implementatie

De risicoanalyse identificeert verschillende potentiële risico's die de ontwikkeling en implementatie van de RAG-service kunnen beïnvloeden. Door deze risico's te identificeren en passende mitigatiestrategieën te implementeren, kan ik de impact van deze risico's minimaliseren en de kans op een succesvol project vergroten. Het is belangrijk om regelmatig de risico's te evalueren en de mitigatiestrategieën bij te werken om in te spelen op nieuwe uitdagingen en veranderingen in de projectomgeving.

3. Werkende software

In dit hoofdstuk wordt de werkende software van de ontwikkelde chatapplicatie uitgebreid besproken, specifiek toegespitst op IBS Toeslagen. De applicatie is ontworpen als een geavanceerd chatapplicatie dat medewerkers in staat stelt om snel en eenvoudig relevante informatie te vinden binnen een complexe kennisomgeving.

Voor een praktische illustratie van de werking en opbouw van deze applicatie is een uitgebreide demo-video gemaakt. In deze video wordt niet alleen gedemonstreerd hoe gebruikers op een natuurlijke manier vragen stellen en antwoorden ontvangen, maar wordt ook diepgaand besproken hoe de applicatie technisch is opgebouwd en hoe verschillende componenten zoals de vector database, AI-model en cloudinfrastructuur samenwerken. De video is te bekijken via

<https://www.youtube.com/watch?v=Gk66Wk0aq-c>

3.1 Doelstelling en functionaliteit

De ontwikkelde applicatie is een geavanceerd systeem dat functioneert als een chatapplicatie, specifiek ontworpen voor IBS Toeslagen. Het primaire doel van deze applicatie is om medewerkers te ondersteunen bij het snel en efficiënt vinden van relevante informatie, waardoor de algehele productiviteit wordt verhoogd en medewerkers tevredener zijn met de informatievoorziening. Door gebruik te maken van moderne AI-technieken in een gebruiksvriendelijke chatinterface kunnen gebruikers in natuurlijke taal vragen stellen, waarna het systeem relevante antwoorden genereert op basis van zowel de vraag als de beschikbare documentatie. Hiermee wordt een centrale, gemakkelijk toegankelijke kennisbron gecreëerd die past binnen de complexe informatieomgeving van IBS Toeslagen.

3.2 Gebruikersgroepen en toepassingsgebied

De primaire gebruikers van de applicatie zijn IBS Toeslagen medewerkers (circa 150 IT-professionals) en nieuwe medewerkers die het inwerktraject doorlopen. Binnen het toepassingsgebied – een Scaled Agile Framework (SAFe) omgeving met 15 gedistribueerde teams – wordt de chatapplicatie ingezet om de uitdagingen rond informatieversnippering te adresseren. In de huidige situatie is informatie verspreid over diverse bronnen zoals code repositories, wiki's en chatkanalen. Dit maakt het lastig voor teamleden om snel de juiste informatie te vinden (Jansen, 2023). De chatapplicatie centraliseert de toegang tot kennis in deze SAFe-context, zodat alle teams kunnen profiteren van een gedeelde “single source of truth” voor documentatie en eerdere Q&A's. Het systeem is daarmee relevant voor zowel ervaren medewerkers die tijd willen besparen bij het zoeken, als voor nieuwe medewerkers die sneller hun weg willen vinden in de beschikbare informatie.

3.3 Systeemvoordelen

Uiteindelijke operationele voordelen waar we op doelen met dit project:

1. Reductie van Zoektijd voor Informatie

Waarom dit een voordeel is: In een complexe organisatie zoals IBS Toeslagen, waar informatie verspreid is over verschillende bronnen zoals repositories, wiki's, en chatkanalen, kan het vinden van de juiste informatie tijdrovend zijn. Medewerkers besteden vaak veel tijd aan het zoeken naar antwoorden op vragen, wat de productiviteit vermindert.

Hoe de chatapplicatie dit oplost: De chatapplicatie maakt gebruik van een geavanceerde vector database en een Large Language Model (LLM) om snel en nauwkeurig antwoorden te genereren op basis van de context van de vraag. Door gebruik te maken van semantische zoekfunctionaliteit, kunnen medewerkers relevante informatie vinden zonder exacte trefwoorden te kennen. Dit vermindert de zoektijd aanzienlijk en verhoogt de efficiëntie.

2. Verhoogde Medewerkerstevredenheid door Efficiëntere Opzoek Functionaliteit

Waarom dit een voordeel is: Medewerkers ervaren vaak frustratie wanneer ze niet snel de benodigde informatie kunnen vinden. Dit kan leiden tot een negatieve werkervaring en verminderde tevredenheid.

Hoe de chatapplicatie dit oplost: De chatapplicatie biedt een intuïtieve en gebruiksvriendelijke interface waarmee medewerkers snel en eenvoudig vragen kunnen stellen en antwoorden kunnen ontvangen. De real-time vraag-antwoord API zorgt ervoor dat medewerkers direct relevante informatie krijgen, wat hun efficiëntie verbetert en hun tevredenheid verhoogt. 2 van de belangrijkste business doelstellingen van dit project (Lansink, 2023).

3. Verbeterde Onboarding-Ervaring voor Nieuwe Medewerkers

Waarom dit een voordeel is: Nieuwe medewerkers hebben vaak moeite om hun weg te vinden in een complexe informatieomgeving. Dit kan de onboarding-ervaring negatief beïnvloeden en de tijd die nodig is om productief te worden verlengen.

Hoe de chatapplicatie dit oplost: De chatapplicatie biedt nieuwe medewerkers een centrale plek waar ze snel antwoorden kunnen vinden op hun vragen. Door gebruik te maken van de semantische zoekfunctionaliteit en de contextuele antwoorden van de LLM, kunnen nieuwe medewerkers sneller de benodigde informatie vinden en zich sneller aanpassen aan hun nieuwe rol. Dit verbetert de onboarding-ervaring en verkort de tijd die nodig is om productief te worden.

4. Geoptimaliseerde Kennisdeling tussen Teams

Waarom dit een voordeel is: In een organisatie met meerdere teams kan het delen van kennis en informatie een uitdaging zijn. Informatie kan verloren gaan of moeilijk toegankelijk zijn voor andere teams, wat de samenwerking en efficiëntie vermindert.

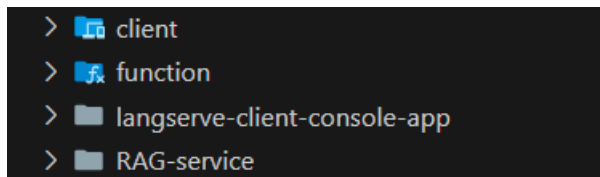
Hoe de chatapplicatie dit oplost: De chatapplicatie fungeert als een centrale hub voor informatie, waar alle teams toegang hebben tot dezelfde bron van waarheid. Door gebruik te maken van de vector database en de LLM, kunnen teams snel en eenvoudig informatie delen en vinden. Dit bevordert de samenwerking en zorgt ervoor dat kennis efficiënt wordt gedeeld tussen teams, wat de algehele productiviteit en effectiviteit verhoogt.

3.4 Ontwikkelingsstandaarden

3.4.1 Codekwaliteit en best practises

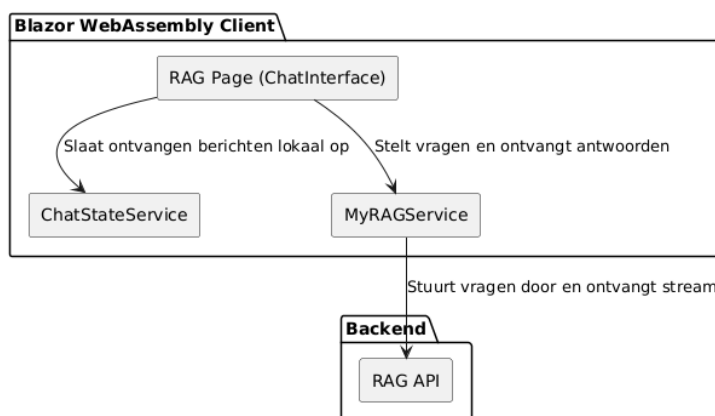
Bij de implementatie van de codebase is veel aandacht besteed aan best practices op het gebied van software-architectuur en -ontwikkeling, om de toepassing onderhoudbaar en uitbreidbaar te

houden. Allereerst is de code structuur opgezet volgens een modulair ontwerp met duidelijke scheiding van verantwoordelijkheden, wat inhoudt dat het opgedeeld is in modules. Elke belangrijke deelfunctionaliteit is ondergebracht in een aparte module of klasse. Zo zijn er afzonderlijke componenten voor bijvoorbeeld de communicatie met de vector database, de aansturing van het LLM, de frontend-logica en de documentopslag. Deze scheiding van logica per component maakt het niet alleen makkelijker om de code te begrijpen, maar faciliteert ook het vervangen of aanpassen van onderdelen zonder ingrijpende wijzigingen in de rest van het systeem. Concreet kan men bijvoorbeeld het taalmodel (LLM) of de vector database in de toekomst verwisselen voor een andere provider, door alleen de betreffende module aan te passen, terwijl de interfaces naar de overige componenten gelijk blijven. Dit draagt bij aan de uitbreidbaarheid van de applicatie.



Afbeelding 3.4 Verdeling van componenten

Tevens is binnen de front-end architectuur het principe van scheiding der verantwoordelijkheden strikt toegepast. In de Blazor WebAssembly client worden functionaliteiten opgesplitst tussen UI-componenten (zoals pagina's) en service-classes. Een pagina-klasse zoals de chatinterface (RAG pagina) doet niet zelf de API-aanroepen, maar maakt gebruik van een service als MyRAGService om vragen door te sturen naar de backend en de gestreamde antwoorden op te halen (zie afbeelding 3.4.1). Tegelijkertijd gebruikt diezelfde pagina een ChatStateService om de ontvangen berichten lokaal op te slaan in de chatgeschiedenis, in plaats van die opslag zelf te beheren. Daarnaast werkt de uploadpagina samen met een BlobService die de technische details van het uploaden naar Azure Blob Storage afhandelt. Door dependency injection toe te passen krijgt de UI daarmee toegang tot deze services zonder ze direct te kennen of te instantiëren, wat zorgt voor losgekoppelde componenten. Dit ontwerppatroon verhoogt de herbruikbaarheid van code, omdat services onafhankelijk kunnen blijven.



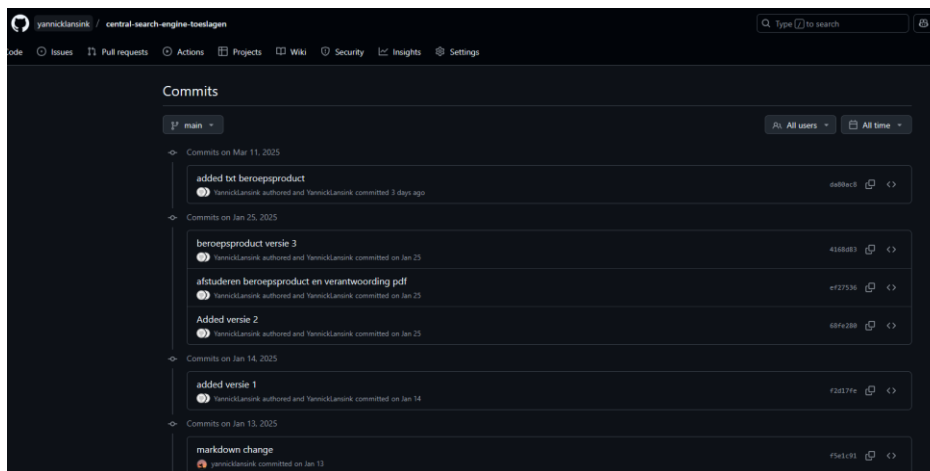
Afbeelding 3.4.1 Verdeling van componenten

Daarnaast zijn gebruik van frameworks en standaarden zorgvuldig overwogen om aan te sluiten bij best practices. Het gebruik van FastAPI voor de backend en Blazor voor de frontend brengt het

voordeel mee van beproefde, community-supported structuren en conventies. Zo dwingt FastAPI bijvoorbeeld typehinting en schema's af voor API-endpoints, wat de betrouwbaarheid verhoogt en later misverstanden voorkomt. Blazor op zijn beurt stimuleert duidelijke herbruikbare componenten. Beide frameworks hebben uitgebreide documentatie en ondersteuning, hetgeen de ontwikkelbaarheid en overdraagbaarheid van de code ten goede komt. Ook is er gekozen voor duidelijke codeerstandaarden en documentatie binnen de code. Belangrijke modules en functies zijn voorzien van toelichtende comments, en namen van variabelen en methodes zijn consistent en betekenisvol gehouden. Dit helpt nieuwe ontwikkelaars of teamleden om de codebase snel eigen te maken, wat de onderhoudbaarheid vergroot.

Verder is de applicatie geconfigureerd met omgevingsvariabelen voor alle gevoelige configuraties en externe koppelingen. API-sleutels voor diensten als OpenAI en Pinecone, evenals endpoints en indexnamen, worden niet hard gecodeerd maar via een .env-bestand of Azure Key Vault ingelezen. Dit volgt security best practices (geen geheimen in de code) en maakt de toepassing flexibeler: bijvoorbeeld een switch van een API-sleutel of overstap naar een andere vector-database vereist enkel een configuratiewijziging, geen aanpassing van de code. Dependency management voor de Python-omgeving wordt verzorgd door Poetry, wat bijdraagt aan reproduceerbare builds en eenvoudig pakketbeheer. Voor het .NET-gedeelte zorgt het gebruik van de standaard .csproj/NuGet dependency-definities ervoor dat alle benodigde libraries expliciet vastliggen. Deze maatregelen voorkomen dependency hell en maken het opzetten van de ontwikkelomgeving eenduidig voor alle betrokkenen.

Tot slot zijn er ook processen ingericht om de codekwaliteit te bewaken gedurende de levenscyclus van het project. De code bevindt zich onder versiebeheer (Git), waardoor wijzigingen traceerbaar zijn en er via pull requests code reviews gehouden kunnen worden. Een best practice om kwaliteit af te dwingen. Deze aanpak voldoet aan de niet-functionele eisen rondom onderhoudbaarheid en codekwaliteit die voor dit project zijn gesteld.



Afbeelding 3.4.2 Commit geschiedenis op Github

3.4.2 Tools in gebruik

Tijdens de ontwikkeling van de chatapplicatie voor IBS Toeslagen is een breed scala aan tools en technologieën ingezet om een robuuste, schaalbare en efficiënte oplossing te realiseren. In dit hoofdstuk worden de belangrijkste tools besproken, met nadruk op hun specifieke rol binnen de softwareontwikkelingscyclus.

Visual Studio en Visual Studio Code

De frontend van de applicatie is ontwikkeld met Visual Studio, wat de integratie met Blazor WebAssembly en .NET Core vergemakkelijkte. Dit bood de ontwikkelaars een krachtige IDE met debugging mogelijkheden en ingebouwde ondersteuning voor C#. Voor de backend, Azure Functions en het beheer van cloud resources werd Visual Studio Code gebruikt. VS Code bood flexibiliteit door de integratie van extensies, waaronder ondersteuning voor Python en Azure-integraties.

Langsmith voor monitoring

Voor de monitoring van de RAG-service werd Langsmith ingezet. Langsmith biedt gedetailleerde logging en tracing van de interacties met het LLM, waardoor de prestaties en betrouwbaarheid van de AI-gestuurde zoekfunctionaliteit kunnen worden geoptimaliseerd. Dit systeem stelde ontwikkelaars in staat om foutieve queries te identificeren en modelafwijkingen te minimaliseren.

Langserve

De implementatie van Langserve bood een efficiënte manier om de interactie tussen de frontend en de achterliggende AI-componenten te stroomlijnen. Door gebruik te maken van Langserve konden API-verzoeken sneller en betrouwbaarder verwerkt worden, waardoor de gebruikerservaring werd verbeterd. De asynchrone aard van Langserve faciliteerde een real-time antwoordfunctionaliteit binnen de chatinterface.

Pinecone voor vectorgebaseerde zoekfunctionaliteit

De applicatie maakt gebruik van Pinecone als vector database voor semantische zoekopdrachten. Door documenten te transformeren in vectorrepresentaties en op te slaan in Pinecone, kunnen gebruikers vragen stellen zonder exacte trefwoorden te hoeven gebruiken. Pinecone faciliteert snelle en nauwkeurige zoekopdrachten, wat essentieel is voor de performance van de chatapplicatie.

GitHub voor versiebeheer

Voor versiebeheer en samenwerking tussen ontwikkelaars werd GitHub gebruikt. De codebase werd beheerd binnen een repository.

Docker voor containerisatie

De RAG-service werd gecontaineriseerd met Docker om platformonafhankelijkheid en eenvoudige deployment mogelijk te maken. Het gebruik van een Dockerfile zorgde ervoor dat de ontwikkelomgeving consistent bleef tussen verschillende ontwikkelaars en omgevingen. De container werd vervolgens gehost binnen Azure.

```
RAG-service > Dockerfile > ...
1 FROM python:3.11-slim
2
3 RUN pip install poetry==1.6.1
4
5 RUN poetry config virtualenvs.create false
6
7 WORKDIR /code
8
9 COPY ./pyproject.toml ./README.md ./poetry.lock* ./
10
11 RUN poetry install --no-interaction --no-ansi --no-root
12
13 COPY ./app ./app
14
15 RUN poetry install --no-interaction --no-ansi
16
17 EXPOSE 8080
18
19 CMD exec uvicorn app.server:app --host 0.0.0.0 --port 8080
20
```

Afbeelding 3.4.4 Dockerfile binnen het RAG-service project

Azure Cloud voor hosting en opslag

Voor de cloudinfrastructuur werd Microsoft Azure gebruikt. De Azure Functions verwerkten documentuploads en genereerden embeddings voor de vector database. Azure Blob Storage werd ingezet voor tijdelijke opslag van documenten met een automatische retentieperiode van één uur om opslagkosten te minimaliseren. Azure Container Instances werden gebruikt om de RAG-service te hosten en schaalbaar uit te rollen.

Command Line Interface (CLI) en PowerShell

Voor het automatiseren van deployment-processen en infrastructuurbeheer werden de Azure CLI en PowerShell scripts gebruikt. Hiermee konden resources in Azure worden beheerd, zoals het opzetten van storage accounts, het deployen van containers en het configureren van netwerkbeveiligingsinstellingen.

3.4.3 Packages gebruikt in het project

Het ontwikkelde systeem bestaat uit een gerichte selectie softwarepakketten verdeeld over drie onderdelen: client-app, serverless functie, en backend AI-service, gericht op schaalbaarheid, onderhoudbaarheid, en integratie met AI.

Client (Blazor WebAssembly & Azure SDK):

De frontend gebruikt Blazor WebAssembly (Microsoft.AspNetCore.Components.WebAssembly), wat snelle, interactieve single-page applicaties mogelijk maakt met gedeelde .NET-code. Azure-integratie verloopt via Azure.Storage.Blobs voor opslag en Azure.Identity (DefaultAzureCredential) voor veilige authenticatie zonder hard-coded secrets. Uploads activeren automatisch Azure Functions via blob-triggers, wat past binnen event-driven cloudarchitectuur. HttpClientFactory (Microsoft.Extensions.Http) beheert efficiënt en robuust HTTP-verkeer met de backend.

Serverless functie (Azure Functions in Python):

Documentverwerking vindt plaats in schaalbare Azure Functions (azure-functions, azure-storage-blob), getriggerd door geüploade documenten. Voor RAG-verwerking worden documenten via Unstructured verwerkt tot tekst, waarna OpenAI's embeddings (openai SDK) gegenereerd worden en opgeslagen in Pinecone's vector-database (pinecone-client), die snel en schaalbaar zoeken mogelijk maakt. LangChain (met langchain-pinecone en langchain-openai) biedt een abstractielaag om eenvoudig retrieval en generatieve processen te orkestreren. python-dotenv beheert configuratie, psutil monitort resources om binnen Azure Functions limieten te blijven.

Backend AI-service (FastAPI):

De backend API draait op FastAPI met Uvicorn voor snelle, streaming-gebaseerde HTTP-requests. FastAPI biedt automatische datavalidatie (via Pydantic) en snelle prestaties. LangChain's LangServe maakt eenvoudig endpoints aan voor bestaande AI-chains, waarmee consistentie in interactie met OpenAI-modellen en Pinecone gegarandeerd blijft. Middleware zoals CORS-configuratie garandeert veilige communicatie tussen client en backend.

Deze heldere taakverdeling, async verwerking in Azure Functions en snelle AI-reacties via FastAPI, benut de sterktes van de gekozen softwarepakketten, wat resulteert in optimale prestaties en betrouwbare interactie binnen het systeem.

3.5 Technische implementatie

De technische implementatie van de chatapplicatie is ontworpen om een efficiënte oplossing te bieden voor het snel en nauwkeurig vinden van informatie binnen IBS Toeslagen. De codebase is

opgedeeld in verschillende componenten die elk een duidelijk afgebakende rol vervullen en die samen het end-to-end proces mogelijk maken. In grote lijnen bestaat het systeem uit een frontend (een Blazor WebAssembly client) die de gebruikersinterface levert, een backend (een FastAPI-service in Python) die de vragen beantwoordt via AI-functionaliteit, een vector database (Pinecone) voor semantische zoekopdrachten, en een documentverwerkingspipeline (Azure Functions met blob-opslag) voor het toevoegen van nieuwe kennis aan het systeem. In de onderstaande subsecties worden de belangrijkste technische onderdelen en beslissingen besproken, inclusief de motivatie voor de gebruikte technologieën en frameworks. Hierbij wordt ook inzicht gegeven in hoe de verschillende componenten met elkaar samenwerken binnen de codebase, en welke best practices zijn toegepast om het systeem onderhoudbaar en uitbreidbaar te houden.

3.5.1 Vector-gebaseerde zoekfunctionaliteit

Afbeelding 3.5.1 Code in RAG-service – Opzetten van retriever om vectoren op te halen

```
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")
vectorstore = Pinecone.from_existing_index(index_name=PINECONE_INDEX_NAME,
                                          embedding=embeddings)

retriever = vectorstore.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={
        "score_threshold": 0.85,
        "k": 3,
    }
)
```

De vector-gebaseerde zoekfunctionaliteit maakt gebruik van cosine similarity voor semantische matching. Documenten worden gerepresenteerd als vectoren in een hoogdimensionale ruimte, waarbij de afstand tussen vectoren de semantische overeenkomst tussen documenten weergeeft. De threshold van 0.85 waarborgt hoge precisie in zoekresultaten door alleen resultaten terug te geven die een hoge mate van overeenkomst hebben met de zoekvraag.

Traditionele zoekmethoden zijn vaak beperkt tot exacte trefwoordovereenkomsten, wat inefficiënt is bij het doorzoeken van grote hoeveelheden ongestructureerde data. Door gebruik te maken van vectorrepresentaties en cosine similarity, kan het systeem semantische relaties tussen documenten identificeren, zelfs als de exacte woorden niet overeenkomen. Dit verhoogt de nauwkeurigheid en relevantie van de zoekresultaten aanzienlijk.

Deze technologie stelt medewerkers van IBS Toeslagen in staat om snel en efficiënt relevante informatie te vinden, wat de productiviteit verhoogt en de tijd die besteed wordt aan het zoeken naar informatie vermindert. Dit is cruciaal voor het succes van het project, aangezien het de kernfunctionaliteit van de applicatie ondersteunt.

3.5.2 Real-time query processing

Afbeelding 3.5.2.1 Code in RAG-service – Onderdeel van het terug streamen van antwoorden aan de gebruiker

```
prompt = ChatPromptTemplate.from_template(template)

# Chain is de runnable that will be executed
chain = RunnableSequence.from_iterable([
    retriever,
    RunnablePassthrough(),
    prompt,
    model,
    StrOutputParser()
])
```

Afbeelding 3.5.2.2 Code in Client (frontend) – Onderdeel van het terug streamen van antwoorden aan de gebruiker

```
public async Task StreamRAGAsync(string input, Func<string, Task> onChunkReceived)
{
    var requestData = new { input = input };
    var json = JsonSerializer.Serialize(requestData);
    var content = new StringContent(json, Encoding.UTF8, "application/json");

    var request = new HttpRequestMessage(HttpMethod.Post, "/RAG/stream")
    {
        Content = content
    };

    var response = await _httpClient.SendAsync(request, HttpCompletionOption.ResponseHeadersRead);

    if (response.IsSuccessStatusCode)
    {
        var stream = await response.Content.ReadAsStreamAsync();
        using (var reader = new StreamReader(stream))
        {
            while (!reader.EndOfStream)
            {
                var chunk = await reader.ReadLineAsync();
                if (!string.IsNullOrEmpty(chunk) && chunk.StartsWith("data: "))
                {
                    var chunkData = chunk.Substring(6);
                    await onChunkReceived(chunkData);
                }
            }
        }
    }
    else
    {
        var errorContent = await response.Content.ReadAsStringAsync();
        throw new ApplicationException($"Error: {response.StatusCode}, Details: {errorContent}");
    }
}
```

Het systeem implementeert asynchrone streaming responses voor real-time interactie, gebruikmakend van FastAPI's streaming mogelijkheden. Dit betekent dat gebruikers direct feedback krijgen terwijl hun vraag wordt verwerkt, wat de interactie met de applicatie vloeiender en efficiënter maakt.

Traditionele systemen voor vraag-antwoord verwerking kunnen traag en inefficiënt zijn, vooral bij complexe queries die veel data vereisen. Door gebruik te maken van asynchrone streaming responses, kan het systeem gebruikers constant bijwerken met real-time informatie, wat de gebruikerservaring aanzienlijk verbetert.

Real-time query processing is essentieel voor de gebruiksvriendelijkheid van de applicatie. Het zorgt ervoor dat medewerkers snel antwoorden krijgen op hun vragen, wat de efficiëntie verhoogt en de tevredenheid van de gebruikers verbetert. Dit draagt bij aan de algehele doelstellingen van het project om de informatiezoekprocessen binnen IBS Toeslagen te optimaliseren.

3.5.3 Automatische document analyse

Afbeelding 3.5.3 Code in Azure function – 1 methode, update document die content omzet voor de vector database

```
def update_document(blob_content, blob_name, index_name, embeddings):

    if not is_valid_file_type(blob_name):
        logging.error(f"Unsupported file type for {blob_name}. Supported file type")
        return

    prefix = blob_name # Use the blob name as prefix
    logging.info(f'filename: {blob_name} en prefix: {prefix}')

    # Delete existing embeddings based on the prefix
    ids_to_delete = []
    for ids in index.list(prefix=f"{prefix}#", namespace=''):
        ids_to_delete.extend(ids)
    if ids_to_delete:
        index.delete(ids=ids_to_delete, namespace='')
        logging.info(f"Existing embeddings for {blob_name} removed: {ids_to_delete}")
    else:
        logging.info(f"No existing embeddings found for filename: {blob_name}")

    # Process the new blob content and add the new embeddings
    files = shared.Files(content=blob_content, file_name=blob_name)

    req = shared.PartitionParameters(
        files=files,
        strategy="hi_res",
    )
    try:
        resp = unstructured_client.general.partition(req)
        elements = dict_to_elements(resp.elements)
    except SDKError as e:
        logging.error(f"Failed to partition document: {e}")
        return
```

```
documents = []
ids = []
for element in chunked_elements:
    metadata = element.metadata.to_dict()
    del metadata["languages"]
    doc_id = generate_id(metadata["filename"])
    ids.append(doc_id)
    documents.append(Document(page_content=element.text, metadata=metadata))

# Add the new documents to the Pinecone vector store
try:
    vectorstore = PineconeVectorStore.from_existing_index(
        index_name=index_name,
        embedding=embeddings
    )

    vectorstore.add_texts(
        texts=[doc.page_content for doc in documents],
        metadatas=[doc.metadata for doc in documents],
        ids=ids
    )
    logging.info(f"New embeddings for {blob_name} added.")
except Exception as e:
    logging.error(f"Failed to add new embeddings: {e}")
```

De Azure Function implementeert een event-driven architectuur voor documentverwerking, waarbij nieuwe uploads automatisch worden gedetecteerd en verwerkt. Dit betekent dat zodra een document wordt geüpload naar Azure Blob Storage, de Azure Function wordt geactiveerd om het document te analyseren en embeddings te genereren.

Embeddings zijn traditionele documentverwerkingssystemen vereisen vaak handmatige interventie om documenten te analyseren en te indexeren. Door gebruik te maken van een event-driven architectuur en AI-modellen voor het genereren van embeddings, kan het systeem automatisch en efficiënt documenten verwerken zonder menselijke tussenkomst. Dit verhoogt de snelheid en nauwkeurigheid van de documentanalyse.

Automatische documentanalyse is van cruciaal belang voor het succes van het project, omdat het de basis vormt voor de zoekfunctionaliteit en de vraag-antwoord verwerking. Door documenten automatisch te analyseren en in chunks op te splitsen, kan het systeem snel en nauwkeurig relevante informatie vinden en antwoorden genereren.

3.6 Systeemarchitectuur

De systeemarchitectuur van de chat applicatie is zorgvuldig ontworpen om een robuuste, schaalbare en efficiënte oplossing te bieden voor het snel en nauwkeurig vinden van informatie binnen IBS Toeslagen. De architectuur bestaat uit vijf hoofdcomponenten die naadloos samenwerken om de functionaliteit van de applicatie te waarborgen:

1. Frontend

De frontend is gebouwd met Blazor WebAssembly, een framework dat het mogelijk maakt om interactieve webapplicaties te ontwikkelen met behulp van C# en .NET. Dit biedt een consistente en responsieve gebruikerservaring.

2. Backend

De backend is ontwikkeld met FastAPI, een modern web framework voor het bouwen van API's met Python. FastAPI staat bekend om zijn hoge prestaties en gebruiksgemak.

3. Vector Database

Pinecone is een gespecialiseerde vector database die is ontworpen voor het opslaan en doorzoeken van vectorrepresentaties van documenten. Het maakt gebruik van geavanceerde zoekalgoritmen om semantische overeenkomsten te vinden.

4. Document Processing

Azure Functions is een serverless compute service die het mogelijk maakt om kleine stukjes code uit te voeren in de cloud zonder de noodzaak van serverbeheer. Het biedt schaalbaarheid en kostenbesparing.

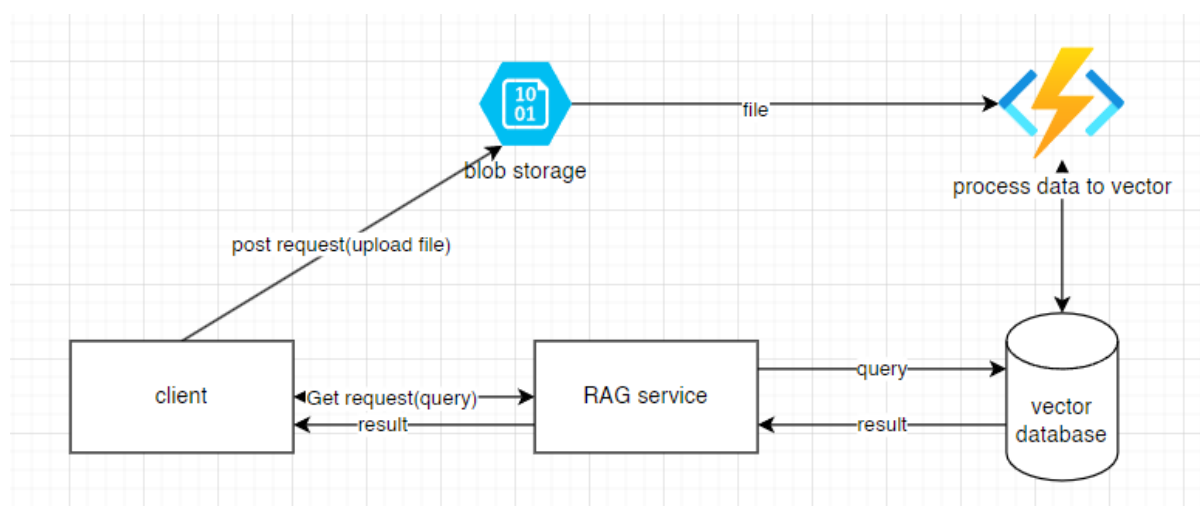
5. Blob storage

Blob Storage is in het leven geroepen om tijdelijk geuploadede bestanden van gebruikers op te slaan. Het heeft een ingebouwd mechanisme dat het na 1 uur de bestanden weer opruimt uit het systeem om kosten te besparen. Dit is 1 van de business doelstelling van IBS Toeslagen.

Deze vijf componenten werken samen volgens een helder interactiepatroon (zie afbeelding 3.6 voor een schematisch overzicht). Samengevat verloopt de werking als volgt: voor vraag-antwoord interacties stuurt de Blazor frontend een vraag naar de FastAPI backend, die op zijn beurt Pinecone en OpenAI inschakelt om een antwoord te genereren, waarna het antwoord gestreamd teruggaat naar de frontend om getoond te worden. Voor documentuploads stuurt de frontend het bestand naar Blob Storage (via de BlobService), waarna een Azure Function automatisch getriggerd wordt om dat document te verwerken en de resulterende kennis (embeddings) in Pinecone beschikbaar te maken.

Dankzij dit ontwerp kan een gebruiker direct na uploaden van een document via de chat vragen stellen over de inhoud van dat document – de backend zal de nieuwe embeddings in Pinecone immers meenemen bij het beantwoorden van vragen. De gehele architectuur is hiermee losjes gekoppeld: elke component heeft een duidelijke taak en communiceert via goed gedefinieerde interfaces of triggers met andere onderdelen. Dit verhoogt de veerkracht van het systeem (een probleem in één component heeft minimale impact op de rest) en maakt het eenvoudiger om componenten te vervangen of up te graden zonder het hele systeem te verstoren.

Interactiepatroon:



Afbeelding 3.6 interactiepatroon

3.7 Beveiligingsmaatregelen

Hieronder volgt een uitgebreide beschrijving van de beveiligingsmaatregelen die zijn geïmplementeerd om te voldoen aan de hoogste standaarden van software engineering, met specifieke aandacht voor beveiliging en gebruikersgemak binnen de context van IBS Toeslagen.

3.7.1 End-to-End encryptie voor datatransport

End-to-end encryptie zorgt ervoor dat gegevens tijdens het transport tussen de verschillende componenten van de applicatie versleuteld zijn. Dit betekent dat gegevens niet leesbaar zijn voor onbevoegde partijen die mogelijk toegang krijgen tot de gegevens tijdens het transport.

Technologische Implementatie:

- **TLS (Transport Layer Security):** Alle communicatie tussen de frontend, backend, en externe diensten zoals Pinecone en OpenAI wordt beveiligd met TLS. Dit protocol zorgt voor een veilige verbinding door gegevens te versleutelen voordat ze worden verzonden en te ontsleutelen zodra ze zijn ontvangen.
- **HTTPS:** Alle API-aanroepen maken gebruik van HTTPS om ervoor te zorgen dat de gegevens tijdens het transport versleuteld zijn.

End-to-end encryptie beschermt de gegevens van gebruikers tegen afluisteren en manipulatie tijdens het transport. Dit verhoogt de vertrouwelijkheid en integriteit van de gegevens.

3.7.2 Tijdelijke documentopslag (1-uurs retentie)

Documenten die door gebruikers worden geüpload, worden tijdelijk opgeslagen in Azure Blob Storage met een retentieperiode van één uur. Dit zorgt ervoor dat documenten alleen worden bewaard zolang als nodig is voor verwerking en daarna automatisch worden verwijderd. Dit geeft de Azure functie de tijd om de content op te pakken en verder te verwerken.

Technologische Implementatie:

- **Azure Blob Storage:** Documenten worden geüpload naar een beveiligde container in Azure Blob Storage. De opslag is geconfigureerd om documenten na één uur automatisch te verwijderen.
- **Versleuteling:** Tijdens de opslagperiode worden documenten versleuteld met industriestandaard encryptieprotocollen zoals TLS voor transport en AES-256 voor opslag.

De tijdelijke opslag van documenten minimaliseert het risico van langdurige blootstelling van gevoelige gegevens.

3.7.3 Lokale chatgeschiedenis opslag

De chatgeschiedenis van gebruikers wordt lokaal opgeslagen in de browser's localStorage. Dit zorgt ervoor dat gebruikers toegang hebben tot hun eerdere gesprekken zonder dat de gegevens naar een externe server worden verzonden.

Technologische Implementatie:

- **localStorage:** De chatgeschiedenis wordt opgeslagen in de localStorage van de browser, wat betekent dat de gegevens alleen toegankelijk zijn op het apparaat van de gebruiker.
- **JavaScript Interop:** De opslag en het ophalen van de chatgeschiedenis worden beheerd met behulp van JavaScript interop (IJSRuntime) in Blazor.

Deze aanpak verhoogt het gebruikersgemak: de gebruiker kan zijn vorige vragen en antwoorden zien bij het hervatten van een sessie, zonder extra wachttijd of afhankelijkheid van een server.

Tegelijkertijd worden serverbronnen ontzien, omdat er geen opslag en ophalen van chatlogs op de backend hoeft plaats te vinden. Het resultaat is een schaalbaardere oplossing (minder server load) en potentieel lagere kosten, omdat opslag en bandbreedte worden bespaard. Belangrijker nog, vanuit beveiligingsoogpunt blijven persoonlijke vraag-antwoordpatronen privé aan de gebruikers kant, wat past bij de vertrouwelijkheidseisen van een overheidsdienst.

3.7.4 Dummy data

Om de privacy van echte gebruikersgegevens te beschermen tijdens de ontwikkeling en testen van de applicatie, is ervoor gekozen om in niet-productie omgevingen met dummy-data te werken. Deze beslissing is genomen in overleg met een security-specialist en houdt in dat er geen live vertrouwelijke data in de ontwikkel- of testfase wordt gebruikt. In plaats daarvan worden gesynthetiseerde of geanonimiseerde gegevens toegepast die representatief zijn voor echte scenario's maar geen gevoelige informatie bevatten.

Technologische implementatie:


- **Dummy data in plaats van echte data:** In alle testcases en demonstraties worden gesimuleerde invoerdata en documenten gebruikt. Bijvoorbeeld, waar in een

productieomgeving een echt beleidsdocument zou worden geüpload, wordt in de test een openbaar of fictief document gebruikt. Ook voorbeeldvragen en -antwoorden zijn gebaseerd op verzonnen casussen.

- Data masking technieken: Indien het gebruik van enige vorm van echte data onvermijdelijk was, worden technieken zoals data masking toegepast. Dit houdt in dat identificerende velden (namen, BSN-nummers, bedragen, etc.) onleesbaar gemaakt of vervangen worden door willekeurige maar plausibele waarden, zodat de dataset realistisch blijft voor testdoeleinden maar geen herleidbare informatie bevat.

Door consequent dummy-data te gebruiken, wordt het risico op datalekken tijdens de ontwikkelcyclus tot een minimum beperkt. Zelfs als er logs of outputs van tests gedeeld zouden worden, bevatten deze geen privacy- of bedrijfsgevoelige informatie. Bovendien dwingt het gebruik van gesimuleerde data af dat het systeem flexibel genoeg is om met verschillende inputdata om te gaan, wat de robuustheid ten goede komt. Deze maatregel toont de nadruk die binnen het project is gelegd op security by design en verantwoord omgaan met data, in lijn met de AVG-richtlijnen en de interne veiligheidsbeleid van IBS Toeslagen.

langchain-unstructured-data •

METRIC	DIMENSIONS	HOST			
cosine	1536	https://pinecone-dummy-data-ibs-toeslagen-ccwlqfw.svc.aped-4627-b74a.pinecone.io			
CLOUD	REGION	TYPE	CAPACITY MODE	RECORD COUNT	
 AWS	us-east-1 	Dense	Serverless	454	

Afbeelding 3.7.4 Dummy data in Pinecone voor IBS Toeslagen

4. Installatiehandleiding

Deze handleiding biedt een uitgebreide uitleg over hoe je de "IBS Toeslagen Chat App" applicatie lokaal kunt opzetten en uitvoeren. De beschreven stappen zijn bedoeld om een gebruiksvriendelijke ervaring te bieden voor zowel beginners als ervaren ontwikkelaars. Het document is zorgvuldig samengesteld om ervoor te zorgen dat alle essentiële aspecten van de installatie worden behandeld. Voor een overzichtelijke versie van deze handleiding kun je ook terecht op de officiële GitHub-repository: <https://github.com/yannicklansink/central-search-engine-toeslagen/blob/main/how-to-run-locally.md>

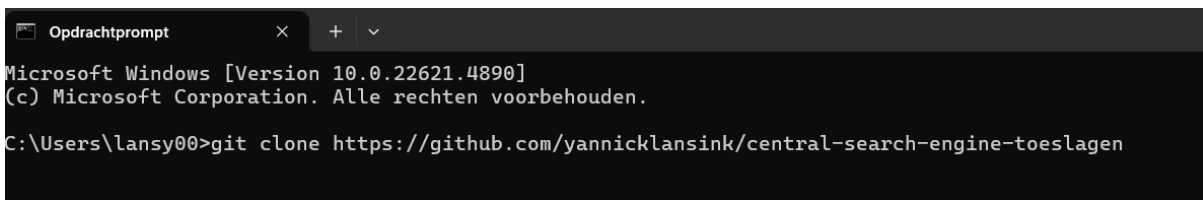
Daarnaast is er een technische demo-video beschikbaar waarin de werking van de chatapplicatie wordt uitgelegd: <https://www.youtube.com/watch?v=Gk66Wk0aq-c>

4.1 Voorbereiding

Voordat je begint, moet je de repository clonen om toegang te krijgen tot de broncode en de projectbestanden.

Open een terminal en voer het volgende commando uit:

```
git clone https://github.com/yannicklansink/central-search-engine-toeslagen
```



Afbeelding 4.1 git clone repo

Dit zal een kopie van de code downloaden naar je lokale machine. Navigeer daarna naar de projectmap:

```
cd central-search-engine-toeslagen
```

Installeer vereiste software:

- **Python 3.11 of hoger:** Dit is nodig om de backend services te draaien die in Python zijn geschreven.
- **Poetry:** Voor dependency management en virtual environments, wat helpt om de juiste versies van de benodigde pakketten te beheren.
- **.NET SDK:** Voor het draaien van de Blazor WebAssembly applicatie, die de frontend van de applicatie vormt.

Download de nieuwste versies van de bovengenoemde software vanaf hun officiële websites en volg de installatie-instructies voor jouw besturingssysteem.

- [python](#)
- [dotnet](#)
- [poetry](#)

Controleer na installatie of de software correct is geïnstalleerd:

```
python --version # Moet 3.11 of hoger tonen
```

```
poetry --version # Moet een geldige versie tonen
```

```
dotnet --version # Moet de .NET SDK versie tonen
```

4.2 Configureer de omgevingsvariabelen

De applicatie maakt gebruik van API-sleutels om verbinding te maken met externe diensten. Deze moeten worden ingesteld in een .env bestand. Dit bestand vind je in de root van het project.

Maak een .env bestand:

1. Kopieer de inhoud van env.example naar een nieuw bestand genaamd .env.

```
cp env.example .env
```

2. Vul de juiste API-sleutels in voor Pinecone, OpenAI en LangChain Tracing. Deze API-sleutels zijn essentieel voor het verbinden met externe diensten die de applicatie gebruikt voor het verwerken van gegevens en het genereren van antwoorden

```
PINECONE_API_KEY=<your-pinecone-api-key>
```

```
PINECONE_INDEX_NAME=langchain-unstructured-data
```

```
OPENAI_API_KEY=<your-openai-api-key>
```

```
LANGCHAIN_TRACING_V2=true
```

```
LANGCHAIN_ENDPOINT=https://api.smith.langchain.com
```

```
LANGCHAIN_API_KEY=<your-langchain-api-key>
```

Pinecone, OpenAI en LangChain werken samen als de kerncomponenten. Pinecone fungeert als de opslagplaats voor vectorgegevens, waardoor het mogelijk is om efficiënt informatie op te slaan en razendsnel op te halen. OpenAI speelt vervolgens de rol van de taalexpert, die op basis van deze gegevens diepgaande antwoorden genereert en complexe vragen begrijpt. Om alles soepel te laten verlopen, coördineert LangChain de interactie tussen deze technologieën. Het beheert de conversaties, houdt de context bij en zorgt ervoor dat alle onderdelen naadloos samenwerken. Samen vormen deze drie technologieën een krachtig geheel dat geavanceerde AI-gestuurde interacties mogelijk maakt. Ze zijn nodig om het project lokaal te laten draaien.

4.3 Start de backend (RAG-service)

De backend verwerkt gebruikersinvoer en genereert antwoorden met behulp van AI en zoektechnologieën.

1. Navigeer naar de RAG-service directory:

```
cd RAG-service
```

2. Installeer de afhankelijkheden met Poetry:

```
poetry install
```

3. Start de RAG-service:

```
poetry run langchain serve
```

```
PS D:\dev\fun\eindopdracht-vector-database\central-search-engine-toeslagen\RAG-service> poetry run langchain serve
INFO: Will watch for changes in these directories: ['D:\dev\fun\eindopdracht-vector-database\central-search-engine-toeslagen\RAG-service']
INFO: Uvicorn running on http://127.0.0.1:8080 (Press CTRL+C to quit)
INFO: Started reloader process [18720] using StatReload
INFO: Started server process [30196]
INFO: Waiting for application startup.

  LANGSERVE

LANGSERVE: Playground for chain "/RAG/" is live at:
LANGSERVE:   ↳ /RAG/playground/
LANGSERVE: See all available routes at /docs/

INFO: Application startup complete.
INFO: 127.0.0.1:40650 - "GET / HTTP/1.1" 307 Temporary Redirect
INFO: 127.0.0.1:40650 - "GET /docs HTTP/1.1" 200 OK
INFO: 127.0.0.1:40650 - "GET /openapi.json HTTP/1.1" 200 OK
INFO: 127.0.0.1:40767 - "POST /RAG/stream HTTP/1.1" 200 OK
```

Afbeelding 4.3 start de RAG service

De backend is nu actief en gereed om aanvragen te verwerken.

4.4 Start de frontend (Blazor WebAssembly)

De frontend is een Blazor WebAssembly-applicatie die de gebruikersinterface van de chatapplicatie weergeeft.

1. Navigeer naar de client/WebApp directory:

```
cd client/WebApp
```

2. Start de Blazor WebAssembly applicatie:

```
dotnet run
```

Hierna verschijnt er een localhost in je terminal die je in je (default) browser moet plakken. Dan kom je op het start scherm waar je uitleg vindt over de chatapplicatie, een technische demo en meta informatie over het project.



Afbeelding 4.4 start scherm frontend

Opmerkingen

Upload Functionaliteit: Om gebruik te maken van de upload functionaliteit, moet je verbinding maken met een Azure account. Volg de stappen Azure Blob Storage configuratie.

4.5 Optioneel: Azure Blob Storage configuratie

Om de upload functionaliteit te gebruiken, moet je een eigen Azure Blob Storage account aanmaken. Dit is nodig omdat de applicatie bestanden opslaat die door gebruikers worden geüpload, en deze opslag moet veilig en toegankelijk zijn.

1. **Maak een Azure Account aan:** Ga naar Azure Portal en maak een account aan als je er nog geen hebt.
2. **Maak een Storage Account aan:**
 - Navigeer naar "Storage accounts" in de Azure Portal.
 - Klik op "Create" en volg de stappen om een nieuw storage account aan te maken.
 - Noteer de connection string van je storage account, deze heb je nodig voor de configuratie.
3. **Configureer de Blob Storage:**
 - Maak een container aan binnen je storage account voor het opslaan van de bestanden.
 - Zorg ervoor dat de juiste toegangsrechten zijn ingesteld zodat de applicatie toegang heeft tot de container.
4. **Update de .env met je Azure Storage gegevens**
`AZURE_STORAGE_CONNECTION_STRING=<your-azure-storage-connection-string>`

Samenvatting

1. Clone de repository en installeer vereiste software.

2. Configureer de omgevingsvariabelen in een .env bestand.
3. Start de backend door naar de RAG-service directory te navigeren en poetry run langchain serve uit te voeren.
4. Start de frontend door naar de client/WebApp directory te navigeren en dotnet run uit te voeren.
5. Optioneel: setup Azure Blob Storage.

Met deze stappen zou je de applicatie lokaal moeten kunnen draaien.

5. Belangrijke functionaliteiten

De applicatie is opgebouwd uit 5 fundamentele kernfunctionaliteiten die gezamenlijk een geavanceerd Retrieval-Augmented Generation (RAG) systeem vormen. Deze 5 worden in dit hoofdstuk besproken.

5.1 Gedetailleerde Analyse van Kernfunctionaliteiten

5.1.1 Document Management en Intelligente Opslag

Technische Implementatie:

- Azure Blob Storage integratie voor veilige documentretentie
- Automatische document analyse pipeline
- Tijdelijk opslagmechanisme met 1-uurs retentiebeleid
- Industriestandaard TLS-encryptie voor data in transit

Gebruikersbehoefte: "Hoe kan ik waarborgen dat mijn documenten veilig worden opgeslagen en efficiënt kunnen worden teruggevonden?"

5.1.2 Real-time Query Processing Systeem

Architectuur:

- FastAPI backend implementatie
- Asynchrone streaming responses
- OpenAI-integratie voor semantische verwerking
- Vector database query optimalisatie

Functionaliteit: Het systeem verwerkt gebruikersverzoeken via geoptimaliseerde API-endpoints die direct relevante informatie extraheren uit de vector database en deze verrijken met contextuele analyse.

5.1.3 Geavanceerde Semantische Zoekfunctionaliteit

Technische Componenten:

- Vector-gebaseerde zoekarchitectuur
- Semantische matching algoritmes
- Pinecone vector database integratie
- Cosine similarity berekeningen voor nauwkeurige resultaten

Implementatie Details: De zoekfunctionaliteit maakt gebruik van geavanceerde vector embeddings om semantische relaties tussen documenten te identificeren, waardoor gebruikers relevante informatie kunnen vinden zonder exacte trefwoorden te kennen.

5.1.4 Automatische Document Analyse met AI

Procesflow:

- Document upload trigger activatie
- Conversie naar embeddings via OpenAI

- Vector opslag in Pinecone
- Automatische metadata extractie
- Semantische indexering

Privacy en Security:

- End-to-end encryptie
- Lokale data retentie

5.1.5 Gepersonaliseerde Chat Interface

Technische Features:

- Browser-based localStorage implementatie
- Real-time synchronisatie
- Persistente chat historie
- Gebruikersspecifieke contextuele verwerking

Deze architectuur stelt IBS Toeslagen in staat om efficiënt informatie te ontsluiten en te analyseren, waarbij geavanceerde AI-technologieën worden gecombineerd met robuuste beveiligingsmaatregelen.

Bronvermelding

Yannicklansink. (z.d.). GitHub - yannicklansink/central-search-engine-toeslagen. GitHub.
<https://github.com/yannicklansink/central-search-engine-toeslagen/tree/main>

Jansen, J. (2023). Lange termijn plannen IBS Toeslagen. IBS Toeslagen.

ICT bij de Belastingdienst - Werken bij de Belastingdienst | Werken bij de Belastingdienst. (z.d.).
Werken Bij de Belastingdienst. <https://werken.belastingdienst.nl/expertises/ict>

Lansink, Y. J. (2023). Plan van Aanpak.

Moesker, N. (2025, 7 februari). Internal Information Retrieval with AI-Powered Chatbot. DataNorth.
<https://datanorth.ai/blog/internal-information-retrieval-with-ai-powered-chatbot>

What are Vector Embeddings? Applications, Use Cases & More | DataStax. (2025, 19 februari).
DataStax. <https://www.datastax.com/guides/what-is-a-vector-embedding>

McLane, B. (2023, 22 augustus). Vector embedding in numeriek vorm en 2d. Datastax.
<https://www.datastax.com/guides/what-is-a-vector-embedding>

Ggaily. (2024, 5 augustus). Gebeurtenisgestuurd schalen in Azure Functions. Microsoft Learn.
<https://learn.microsoft.com/nl-nl/azure/azure-functions/event-driven-scaling?tabs=azure-cli>

Wikipedia contributors. (2025b, februari 5). Separation of concerns. Wikipedia.
https://en.wikipedia.org/wiki/Separation_of_concerns

Streaming Server-Side Rendering. (z.d.). <https://www.patterns.dev/react/streaming-ssr/>

Golden, R. (2023, 28 april). AI increased customer service agent productivity by 14%, study finds. HR Dive.
<https://www.hrdive.com/news/generative-ai-chatgpt-increased-customer-service-agent-productivity/648925>

NV, E., & StockTitan. (2024, 26 maart). New Research from Elastic Finds Conversational Search Could Yield Staggering Productivity Returns. StockTitan. <https://www.stocktitan.net/news/ESTC/new-research-from-elastic-finds-conversational-search-could-yield-b2c17fm1qs75.html>

Ggaily. (2023, 24 mei). Azure Functions Overview. Microsoft Learn. <https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview?pivots=programming-language-csharp>