

# **Integratie van AI in het softwareontwikkelp proces**

**Technology**

bachelor in de **elektronica-ICT**

afstudeerrichting **ICT**

campus **Brugge Xaverianenstraat**

**Simon Stijnen**

bedrijfsmentor: Sofie Depovere

VIVES-mentor: Ludovic Espeel

academiejaar **2025-2026**





# Integratie van AI in het softwareontwikkelp proces

**Technology**

bachelor in de **elektronica-ICT**

afstudeerrichting **ICT**

campus **Brugge Xaverianenstraat**

**Simon Stijnen**

bedrijfsmentor: Sofie Depovere

VIVES-mentor: Ludovic Espeel

academiejaar **2025-2026**

# Integration of AI in the Software Development Process

**Technology**

bachelor in de **elektronica-ICT**

afstudeerrichting **ICT**

campus **Brugge Xaverianenstraat**

**Simon Stijnen**

bedrijfsmentor: Sofie Depovere

VIVES-mentor: Ludovic Espeel

academiejaar **2025-2026**

## Voorwoord

Deze bachelorproef was niet mogelijk geweest zonder de steun en begeleiding van een aantal mensen aan wie ik oprecht dankbaar ben.

In de eerste plaats gaat mijn dank uit naar Sofie Depovere, mijn bedrijfsmentor bij CERM NV. Haar betrokkenheid, concrete feedback en vertrouwen waren een constante steun doorheen de hele stage- en bachelorproefperiode. Daarnaast wil ik Jan Heyse en Peter Heyse bedanken voor de kans die zij mij gaven om deel uit te maken van het CERM-team.

Ook het Lexis-team verdient een bijzondere vermelding. De vrijheid die ik kreeg om zelfstandig bij te dragen en de open, collegiale sfeer maakten van deze periode een bijzonder leerrijke ervaring.

Tot slot dank ik Ludovic Espeel, mijn VIVES-mentor, voor de begeleiding vanuit de opleiding en de waardevolle feedback tijdens het schrijfproces.



## Abstract

De softwareontwikkeling evolueert in snel tempo onder invloed van AI-gestuurde ontwikkeltools zoals Large Language Models (LLM's), prompt engineering en het Model Context Protocol (MCP). Ook roept deze evolutie vragen op over codekwaliteit, afhankelijkheid en de rolverschuiving van de ontwikkelaar. Bestaand onderzoek richt zich vooral op gecontroleerde experimenten, terwijl de praktijktoepassing in een professionele bedrijfscontext minder belicht blijft.

Deze bachelorproef onderzoekt hoe AI-ondersteuning het ontwikkelproces van een cloudgebaseerde applicatie kan versnellen en verbeteren, en welke voordelen en risico's daarmee samengaan. De casestudie is Lexis, een applicatie van CERM die inkomende e-mails van drukkerijkanten met een LLM analyseert en de geëxtraheerde bedrijfsgegevens doorstuurt naar het Management Informatie Systeem (MIS).

De aanpak bestaat uit zowel een theoretisch als een praktisch deel. De theorie richt zich op de bouwstenen van AI-ontwikkeling, zoals prompt engineering, MCP, skills en agents. Daarnaast zijn er binnen Lexis drie concrete implementaties opgeleverd: een authenticatie-oplossing voor toegang voor meerdere klanten, een interne tool voor testdatabasebeheer, en een gemoderniseerde API-testinfrastructuur. Het werk verliep volgens de Agile/Scrum-methodiek met wekelijkse sprints.

De resultaten tonen dat ontwikkeling met AI de oplevering van zowel de interne testdatatool als de migratie van de API-testcollectie aanzienlijk versnelde, en een gevoelig stuk authenticatielogica succesvol naar productie bracht. Lexis draait, na zes weken ontwikkeling, bij klanten van CERM. De ervaring legde ook beperkingen bloot: zonder gerichte sturing levert AI werkende, maar niet steeds productieklare code op, en blijft de outputkwaliteit sterk afhankelijk van de meegegeven context.

Voor CERM bevestigt dit dat AI-tools structureel kunnen worden geïntegreerd in het ontwikkelproces, op voorwaarde dat de ontwikkelaar de eindbeoordeling blijft maken. AI fungeert het sterkst als verlengstuk van de ontwikkelaar, niet als vervanger, en levert de meeste waarde wanneer ze wordt ingebed in duidelijke processen met heldere instructies.

**Kernwoorden:** AI-ondersteunde softwareontwikkeling, Large Language Models, prompt engineering, Model Context Protocol, skills, agents, cloud computing, Azure, Agile/Scrum, CERM



# Inhoudsopgave

|                                                                         |           |
|-------------------------------------------------------------------------|-----------|
| Voorwoord .....                                                         | 5         |
| Abstract .....                                                          | 7         |
| Inhoudsopgave .....                                                     | 9         |
| Lijst met illustraties .....                                            | 11        |
| Gebruikte symbolen en afkortingen .....                                 | 12        |
| Inleiding .....                                                         | 13        |
| <b>1. AI-ondersteunde ontwikkeling .....</b>                            | <b>15</b> |
| 1.1. Prompt engineering .....                                           | 15        |
| 1.1.1. Invloed van contextlengte op modelprestaties .....               | 16        |
| 1.1.2. Prompting technieken .....                                       | 17        |
| 1.1.3. Prompt engineering in de praktijk .....                          | 19        |
| 1.2. Model Context Protocol servers .....                               | 20        |
| 1.2.1. Introductie tot MCP-servers .....                                | 21        |
| 1.2.2. Onderliggende technologieën .....                                | 22        |
| 1.2.3. Werking van MCP .....                                            | 23        |
| 1.2.4. MCP binnen de context van een LLM .....                          | 25        |
| 1.2.5. Voordelen van een MCP .....                                      | 26        |
| 1.2.6. Beperkingen van een MCP .....                                    | 27        |
| 1.2.7. Praktijkvoorbeelden van MCP-servers .....                        | 27        |
| 1.3. Skills .....                                                       | 29        |
| 1.3.1. Introductie tot skills .....                                     | 29        |
| 1.3.2. Ontstaan en evolutie .....                                       | 30        |
| 1.3.3. Opbouw van een skill .....                                       | 30        |
| 1.3.4. Toepassing in de praktijk .....                                  | 31        |
| 1.3.5. Vergelijking met MCP-servers .....                               | 32        |
| 1.4. Agents .....                                                       | 33        |
| 1.4.1. AGENTS.md .....                                                  | 33        |
| 1.4.2. Agent modi .....                                                 | 34        |
| 1.4.3. Reasoning modi .....                                             | 35        |
| 1.4.4. Sub-agents .....                                                 | 36        |
| 1.4.5. Agent teams .....                                                | 36        |
| 1.4.6. Praktijk scenario: oplossen van een in Jira beschreven bug ..... | 37        |
| 1.5. Voordelen van AI-ondersteunde ontwikkeling .....                   | 40        |
| 1.5.1. Snellere ontwikkeling en prototyping .....                       | 40        |
| 1.5.2. Automatisering van repetitieve taken .....                       | 40        |
| 1.5.3. Ondersteuning bij testen en debuggen .....                       | 41        |
| 1.5.4. Verbetering van codekwaliteit .....                              | 41        |
| 1.6. Gevaren van AI-ondersteunde ontwikkeling .....                     | 42        |
| 1.6.1. Te grote afhankelijkheid van AI .....                            | 42        |
| 1.6.2. Kwaliteitsrisico's en hallucinaties .....                        | 42        |
| 1.6.3. Beveiligingsrisico's en kwetsbaarheden .....                     | 42        |
| 1.6.4. Verlies van domein- en codekennis .....                          | 43        |
| 1.6.5. Privacy en Data Compliance .....                                 | 43        |
| 1.6.6. Ethische overwegingen .....                                      | 44        |
| <b>2. OAuth2 .....</b>                                                  | <b>45</b> |
| 2.1. Wat is OAuth2 .....                                                | 45        |

|           |                                                           |           |
|-----------|-----------------------------------------------------------|-----------|
| 2.2.      | Terminologie en concepten .....                           | 46        |
| 2.2.1.    | <i>Tokens in OAuth2 .....</i>                             | 46        |
| 2.2.2.    | <i>Clienttypes en het client secret .....</i>             | 47        |
| 2.3.      | Varianten van OAuth2.....                                 | 47        |
| 2.3.1.    | <i>Authorization Code flow met PKCE .....</i>             | 48        |
| 2.3.2.    | <i>Client Credentials flow .....</i>                      | 50        |
| 2.4.      | Implementatie OAuth2 in Lexis.....                        | 51        |
| 2.4.1.    | <i>Architectuur .....</i>                                 | 51        |
| 2.4.2.    | <i>Flow.....</i>                                          | 51        |
| 2.4.3.    | <i>Gebruik van AI bij de implementatie.....</i>           | 52        |
| <b>3.</b> | <b>Dataset Explorer.....</b>                              | <b>53</b> |
| 3.1.      | Probleemstelling .....                                    | 53        |
| 3.1.1.    | <i>Context .....</i>                                      | 53        |
| 3.1.2.    | <i>Scope.....</i>                                         | 53        |
| 3.2.      | Implementatie .....                                       | 54        |
| 3.2.1.    | <i>Technologie .....</i>                                  | 54        |
| 3.2.2.    | <i>Vibe-coding met Cursor.....</i>                        | 54        |
| 3.2.3.    | <i>Datastructuur.....</i>                                 | 54        |
| 3.2.4.    | <i>Obstakels.....</i>                                     | 55        |
| 3.3.      | Resulterende applicatie.....                              | 55        |
| 3.3.1.    | <i>Functionaliteiten .....</i>                            | 56        |
| 3.3.2.    | <i>Navigatie en export.....</i>                           | 58        |
| 3.3.3.    | <i>Dashboard.....</i>                                     | 58        |
| 3.4.      | Reflectie.....                                            | 59        |
| 3.4.1.    | <i>Resultaten .....</i>                                   | 59        |
| 3.4.2.    | <i>Vibe coding als ontwikkelmethode .....</i>             | 59        |
| 3.4.3.    | <i>Beperkingen .....</i>                                  | 60        |
| 3.4.4.    | <i>Leermomenten en aanbevelingen.....</i>                 | 60        |
| <b>4.</b> | <b>Bruno.....</b>                                         | <b>61</b> |
| 4.1.      | Probleem .....                                            | 61        |
| 4.2.      | Keuze Bruno .....                                         | 61        |
| 4.3.      | Implementatie .....                                       | 62        |
| 4.3.1.    | <i>Opzet van de collectie .....</i>                       | 62        |
| 4.3.2.    | <i>Environments .....</i>                                 | 63        |
| 4.3.3.    | <i>Testen.....</i>                                        | 63        |
| 4.3.4.    | <i>CLI en pipeline-integratie .....</i>                   | 63        |
| 4.3.5.    | <i>Toekomstige uitbreidingen .....</i>                    | 63        |
| 4.4.      | Gebruik van AI .....                                      | 64        |
| 4.5.      | Resultaten .....                                          | 64        |
| 4.6.      | Reflectie.....                                            | 65        |
| 4.6.1.    | <i>Bestandsgebaseerde API-collecties als aanpak .....</i> | 65        |
| 4.6.2.    | <i>Beperkingen .....</i>                                  | 65        |
| 4.6.3.    | <i>Leermomenten en aanbevelingen.....</i>                 | 65        |
|           | <b>Algemeen besluit .....</b>                             | <b>67</b> |
|           | <b>Bibliografie .....</b>                                 | <b>69</b> |

## Lijst met illustraties

|                                                                                                                           |    |
|---------------------------------------------------------------------------------------------------------------------------|----|
| <b>Figuur 1</b> Logo van CERM NV .....                                                                                    | 13 |
| <b>Figuur 1.1</b> Accuraatheid van vier modellen op basis van hun contextvenster .....                                    | 16 |
| <b>Figuur 1.2</b> Prompting scenario 1 – basis .....                                                                      | 19 |
| <b>Figuur 1.3</b> Prompting scenario 1 – geavanceerd .....                                                                | 20 |
| <b>Figuur 1.4</b> Prompting scenario 2 – basis .....                                                                      | 20 |
| <b>Figuur 1.5</b> Prompting scenario 2 – geavanceerd .....                                                                | 20 |
| <b>Figuur 1.6</b> Architectuur van een MCP-integratie met twee servers .....                                              | 24 |
| <b>Figuur 1.7</b> Hypothetisch voorbeeld dat demonstreert hoe MCP wordt ingeladen in het contextvenster van een LLM ..... | 26 |
| <b>Figuur 1.8</b> Voorbeeld van een MCP-configuratie bestand.....                                                         | 28 |
| <b>Figuur 1.9</b> Voorbeeld van een prompt die MCP-servers zal aanroepen.....                                             | 29 |
| <b>Figuur 1.10</b> Fragment uit de team-workflow-skill, ontwikkeld voor het Lexis-project .....                           | 30 |
| <b>Figuur 1.11</b> Prompt van het praktijkscenario bij AI-agents .....                                                    | 38 |
| <b>Figuur 1.12</b> Resultaat van praktijkscenario AI-agents.....                                                          | 39 |
| <b>Figuur 2.1</b> Voorbeeld van de “Authorization” header .....                                                           | 46 |
| <b>Figuur 2.2</b> Grant types in Lexis Architecture .....                                                                 | 48 |
| <b>Figuur 2.3</b> Authorization Code flow .....                                                                           | 49 |
| <b>Figuur 2.4</b> Client Credentials flow .....                                                                           | 50 |
| <b>Figuur 3.1</b> Initiële prompt voor de Dataset Explorer-applicatie .....                                               | 54 |
| <b>Figuur 3.2</b> Folder structuur datasets.....                                                                          | 55 |
| <b>Figuur 3.3</b> Dataset explorer overview pagina .....                                                                  | 56 |
| <b>Figuur 3.4</b> Dataset explorer e-mail details pagina .....                                                            | 57 |
| <b>Figuur 3.5</b> Output editor van een e-mail .....                                                                      | 58 |
| <b>Figuur 3.6</b> Dashboardweergave met globale statistieken en een overzicht per klant .....                             | 59 |
| <b>Figuur 4.1</b> Overzicht van de Bruno-collectie voor de CERM REST API .....                                            | 62 |
| <b>Tabel 1.1</b> Overzicht van prompt engineering technieken.....                                                         | 19 |
| <b>Tabel 1.2</b> Vergelijking van MCP-transportmechanismen .....                                                          | 23 |
| <b>Tabel 1.3</b> Vergelijking tussen skills en MCP-servers .....                                                          | 32 |
| <b>Tabel 1.4</b> Vergelijking van agent modi in Claude Code.....                                                          | 35 |
| <b>Tabel 1.5</b> Overzicht van reasoning niveaus.....                                                                     | 36 |
| <b>Tabel 1.6</b> Vergelijking tussen sub-agents en agent teams.....                                                       | 37 |
| <b>Tabel 2.1</b> Vergelijking tussen de access token en refresh token .....                                               | 47 |
| <b>Tabel 2.2</b> Vergelijking tussen confidential en public clients .....                                                 | 47 |
| <b>Tabel 2.3</b> Vergelijking van OAuth2 grant types toegepast in Lexis.....                                              | 51 |

## Gebruikte symbolen en afkortingen

|         |                                              |
|---------|----------------------------------------------|
| AI      | ARTIFICIAL INTELLIGENCE                      |
| AAIF    | AGENTIC AI FOUNDATION                        |
| APA     | AMERICAN PSYCHOLOGICAL ASSOCIATION           |
| API     | APPLICATION PROGRAMMING INTERFACE            |
| ASP.NET | ACTIVE SERVER PAGES .NET                     |
| BFF     | BACKEND-FOR-FRONTEND                         |
| CI/CD   | CONTINUOUS INTEGRATION / CONTINUOUS DELIVERY |
| CLI     | COMMAND LINE INTERFACE                       |
| CoT     | CHAIN-OF-THOUGHT                             |
| CRUD    | CREATE, READ, UPDATE, DELETE                 |
| CSRF    | CROSS-SITE REQUEST FORGERY                   |
| CWE     | COMMON WEAKNESS ENUMERATION                  |
| DTO     | DATA TRANSFER OBJECT                         |
| GDPR    | GENERAL DATA PROTECTION REGULATION           |
| GPT     | GENERATIVE PRE-TRAINED TRANSFORMER           |
| HTTP    | HYPERTEXT TRANSFER PROTOCOL                  |
| IETF    | INTERNET ENGINEERING TASK FORCE              |
| JSON    | JAVASCRIPT OBJECT NOTATION <sup>1</sup>      |
| LLM     | LARGE LANGUAGE MODEL                         |
| LSP     | LANGUAGE SERVER PROTOCOL                     |
| MCP     | MODEL CONTEXT PROTOCOL                       |
| MIME    | MULTIPURPOSE INTERNET MAIL EXTENSIONS        |
| MIS     | MANAGEMENT INFORMATIE SYSTEEM                |
| MVP     | MINIMUM VIABLE PRODUCT                       |
| OAuth2  | OPEN AUTHORIZATION 2.0                       |
| OpenAPI | OPEN API INITIATIVE SPECIFICATION            |
| OWASP   | OPEN WORLDWIDE APPLICATION SECURITY PROJECT  |
| PKCE    | PROOF KEY FOR CODE EXCHANGE                  |
| REST    | REPRESENTATIONAL STATE TRANSFER              |
| RPC     | REMOTE PROCEDURE CALL                        |
| SDK     | SOFTWARE DEVELOPMENT KIT                     |
| SPA     | SINGLE PAGE APPLICATION                      |
| SSE     | SERVER-SENT EVENTS <sup>7</sup>              |
| UI      | USER INTERFACE                               |
| URI     | UNIFORM RESOURCE IDENTIFIER                  |
| XML     | EXTENSIBLE MARKUP LANGUAGE <sup>4</sup>      |
| XSS     | CROSS-SITE SCRIPTING                         |
| YAML    | YAML Ain't Markup Language                   |

## Inleiding

Softwareontwikkelaars beschikken vandaag over een groeiend arsenaal aan AI-tools die beloven hun werk sneller en beter te maken. Toch blijft de vraag hoe deze tools het beste worden ingezet in een realistisch bedrijfsproject grotendeels onbeantwoord. De centrale onderzoeksvraag van deze bachelorproef luidt daarom: op welke manieren kan AI worden ingezet om het ontwikkelproces van een cloudgebaseerde applicatie te versnellen en te verbeteren? Aanvullend wordt onderzocht welke concrete voordelen en risico's dit met zich meebrengt, en hoe ontwikkelaars hun werkwijze moeten aanpassen om AI effectief te integreren.

Het onderzoek naar AI-ondersteunde softwareontwikkeling is nog relatief jong. Zo toonden Peng et al. (2023) in een gecontroleerd experiment aan dat ontwikkelaars die GitHub Copilot gebruikten een taak gemiddeld 55,8% sneller voltooiden dan de controlegroep. Bestaande studies richten zich echter voornamelijk op de productiviteitswinst van individuele tools in gecontroleerde omgevingen. Deze bachelorproef onderscheidt zich door de focus op de praktijktoepassing: hoe worden AI-tools ingezet in een realistisch bedrijfsproject met meerdere ontwikkelaars, een complexe architectuur en concrete deadlines?

### Voorstelling van het bedrijf

CERM NV (zie Figuur 1), gevestigd in Oostkamp, is een grote ontwikkelaar van Management Informatie Systemen (MIS) voor de label- en verpakkingsindustrie. Het bedrijf automatiseert het volledige zakelijke en technische proces van drukkerijen, van calculatie en orderverwerking tot productieplanning, voorraadbeheer en facturatie, en opereert vanuit kantoren in België, de Verenigde Staten, Australië en Zweden.

**Figuur 1**

*Logo van CERM NV*



*Opmerking.* Overgenomen uit de website van CERM NV (2026). <https://www.cerm.net/>. Copyright 2026, CERM NV.

Tijdens de stage- en bachelorproefperiode werd meegewerkt aan de ontwikkeling van Lexis, een cloudgebaseerde applicatie die inkomende e-mails van drukkerijkanten automatisch analyseert met behulp van een Large Language Model (LLM) en de geëxtraheerde bestelgegevens via de CERM REST API doorstuurt naar het CERM MIS. De applicatie combineert C# en ASP.NET voor de backend, Python voor de AI-module, en Microsoft Azure als cloudplatform. Het project werd ontwikkeld volgens de Agile/Scrum-methodiek met wekelijkse sprints, daily stand-ups en sprint reviews.

Lexis vormt de praktijkcontext waarbinnen de AI-ondersteunde werkwijzen die in deze bachelorproef worden onderzocht, concreet werden toegepast. Het project bood een realistisch en uitdagend kader: een gedistribueerde architectuur, een multi-tenantomgeving, en strikte deadlines maakten het bijzonder geschikt om de mogelijkheden én grenzen van AI-tools in professionele software-ontwikkeling te evalueren. Voor CERM levert dit onderzoek inzichten op over hoe AI-tools structureel kunnen worden ingezet om de ontwikkelsnelheid te verhogen en de kwaliteit van het ontwikkelproces te borgen.

## Doelstelling

Het doel van deze bachelorproef is tweeledig. Enerzijds wordt een overzicht geboden van de concepten, tools en technieken die vandaag beschikbaar zijn voor AI-ondersteunde software-ontwikkeling, waaronder prompt engineering, het Model Context Protocol (MCP) en skills. Anderzijds wordt aan de hand van het Lexis-project geïllustreerd hoe deze tools in de praktijk werden ingezet, welke technische uitdagingen zich voordeden en hoe AI het ontwikkelproces beïnvloedde.

Concreet komen de volgende onderwerpen aan bod: de rol van prompt engineering bij het aansturen van LLM's voor codegeneratie en domeinspecifieke taken, het gebruik van het Model Context Protocol om AI-modellen toegang te geven tot externe tools en systemen, de implementatie van OAuth2-authenticatie in een multi-tenantarchitectuur, de ontwikkeling van een interne tool voor het beheren en valideren van testdata, de integratie van tooling zoals Bruno voor API-testing in een teamcontext, en de voordelen en risico's van AI-ondersteunde ontwikkeling op basis van praktijkervaring.

## Afbakening

Deze bachelorproef beperkt zich tot de inzet van AI binnen het domein van softwareontwikkeling. Toepassingen van AI in andere domeinen, zoals marketing, design of bedrijfsvoering, vallen buiten de scope van dit onderzoek. De technische werking van de gebruikte LLM's wordt niet op modelniveau besproken; de focus ligt op het gebruik ervan als hulpmiddel voor de ontwikkelaar. De bevindingen zijn gebaseerd op de praktijkervaring opgedaan bij CERM en zijn niet het resultaat van een gecontroleerd wetenschappelijk experiment.

## Leeswijzer

De bachelorproef is als volgt opgebouwd. Hoofdstuk 1 behandelt de theoretische en praktische aspecten van AI-ondersteunde ontwikkeling: prompt engineering, het Model Context Protocol, skills, en de voordelen en risico's van deze werkwijze. Hoofdstuk 2 gaat dieper in op de OAuth2-implementatie binnen de Lexis-architectuur. Hoofdstuk 3 bespreekt de Dataset Explorer, een interne tool voor het beheren en valideren van testdata. Hoofdstuk 4 behandelt de integratie van Bruno als API-testingtool in een teamcontext. Het algemeen besluit reflecteert op de behaalde resultaten en formuleert aanbevelingen voor de toekomst.

# 1. AI-ondersteunde ontwikkeling

De opkomst van Large Language Models (LLM's) heeft de manier waarop software wordt ontwikkeld fundamenteel veranderd. Waar AI-tools eerder vooral werden ingezet voor eenvoudige taken zoals autocomplete, zijn hedendaagse modellen in staat om volledige codeblokken te genereren, architectuurbeslissingen te motiveren, tests op te stellen en documentatie te schrijven. Deze verschuiving biedt ontwikkelteams aanzienlijke productiviteitswinst, maar stelt tegelijk nieuwe eisen aan de vaardigheden van de ontwikkelaar (Anysphere, 2026; Anthropic, 2026a).

Binnen het Lexis-project werd AI-ondersteunde ontwikkeling beschouwd als een kans om nieuwe werkwijzen te verkennen. Het team was niet vertrouwd met AI-tools, en zag het project als een ideaal moment om daar ervaring mee op te doen. Als eerste stap werd de gebruikersinterface van Lexis gegenereerd met Lovable, een AI-gestuurde UI-builder (Lovable, 2026). Het resultaat was visueel sterk en werd door het team goed onthaald, maar de functionele logica diende volledig manueel uitgewerkt te worden. Daarnaast werd Cursor ingezet als AI-ondersteunde code-editor (Anysphere, 2026). Op die manier fungeerde Lexis als een testomgeving voor AI-tooling, waarbij de meerwaarde en de beperkingen ervan in de praktijk werden afgetoetst.

## 1.1. Prompt engineering

Naarmate LLM's een steeds centralere rol innemen in softwareontwikkeling, wordt de manier waarop een ontwikkelaar met deze modellen communiceert bepalend voor de kwaliteit van de output. Prompt engineering is de discipline die zich hierop richt: het systematisch ontwerpen en verfijnen van prompts om een LLM aan te sturen naar het gewenste resultaat (Anthropic, z.d.c). Waar traditionele softwaresystemen worden aangestuurd via deterministische functies en configuraties, reageert een LLM op natuurlijke taal. Die flexibiliteit is tegelijkertijd een uitdaging: zelfs kleine verschillen in formulering kunnen de output sterk beïnvloeden.

Een prompt bestaat doorgaans uit meerdere onderdelen die samen het gedrag van het model sturen. Binnen die onderdelen is het onderscheid tussen de system prompt en de user prompt het meest fundamenteel. De system prompt is een set instructies die voorafgaat aan het eigenlijke gesprek en het model een rol, context en gedragsregels meegeeft. In de praktijk wordt de system prompt niet door de eindgebruiker ingesteld, maar door de ontwikkelaar of het platform dat de applicatie beheert. Platforms zoals Claude Code stellen deze instructies automatisch in op basis van de configuratie van de applicatie. Een typische system prompt definieert bijvoorbeeld de toon (*“je bent een behulpzame assistent”*), het domein (*“je antwoordt uitsluitend over software-ontwikkeling”*) en beperkingen (*“antwoord altijd in het Nederlands”*). De user prompt is de eigenlijke invoer van de gebruiker bij elke interactie (Anthropic, z.d.c).

Naast de tekstuele inhoud beïnvloeden ook modelparameters de output. De belangrijkste is de temperatuur: een waarde tussen 0 en 1 die bepaalt hoe deterministisch het model reageert. Bij een lage temperatuur (dicht bij 0) is de output voorspelbaar en consistent; bij een hogere waarde introduceert het model meer variatie en creativiteit. Ook deze parameters worden doorgaans door de ontwikkelaar geconfigureerd en niet door de eindgebruiker. Voor toepassingen waarbij een

vaste structuur vereist is, zoals de JSON-uitvoer<sup>1</sup> in het Lexis-project, wordt de temperatuur laag ingesteld om consistente output te garanderen.

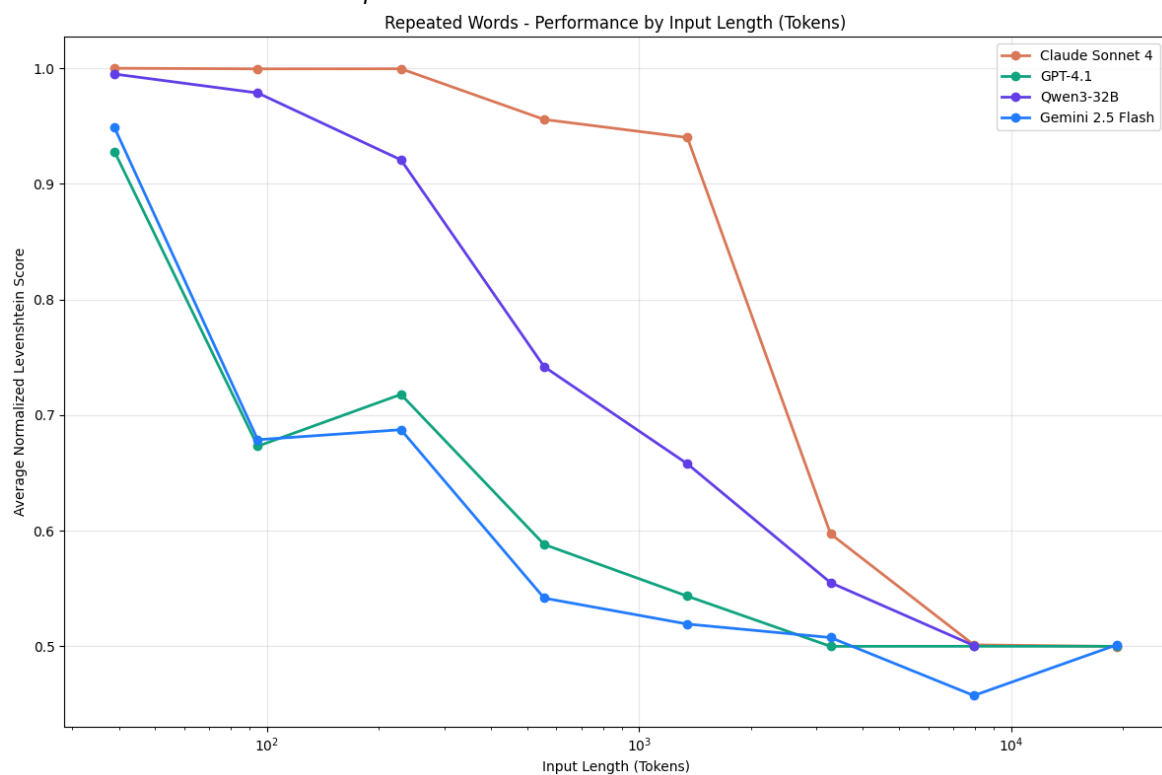
### 1.1.1. Invloed van contextlengte op modelprestaties

Moderne LLM's beschikken over steeds grotere contextvensters, tot miljoenen tokens<sup>2</sup>. Maar het is een misvatting dat een model alle informatie in zijn contextvenster even betrouwbaar verwerkt.

Onderzoek van Chroma (Hong et al., 2025) toonde aan dat alle 18 geteste modellen aantoonbaar slechter presteren naarmate de input langer wordt. Hieronder vallen onder andere de toenmalige topmodellen GPT-4.1, Claude Sonnet 4 en Gemini 2.5 Flash. Dit fenomeen wordt *context rot* genoemd. Figuur 1.1 toont hoe de accuraatheid bij alle geteste modellen daalt met toenemende inputlengte.

**Figuur 1.1**

*Accuraatheid van vier modellen op basis van hun contextvenster.*



*Opmerking.* Overgenomen uit *Context Rot: How Increasing Input Tokens Impacts LLM Performance* door Hong et al. (2025). <https://www.trychroma.com/research/context-rot>. Copyright 2026 Chroma.

De oorzaak is aandachtsdilutie (attention dilution): transformerarchitecturen<sup>3</sup> verdelen aandacht over alle tokens in de context. Naarmate de context groeit, wordt de aandacht voor elke afzonderlijke token verminderd, waardoor het model relevante passages minder sterk gewicht toekent. Dit is geen tijdelijk tekort maar een structurele eigenschap van de transformerarchitectuur. Ondanks

<sup>1</sup> Een tekstgebaseerd dataformaat voor gestructureerde gegevens. Veel gebruikt wordt voor gegevens-uitwisseling tussen applicaties en API's.

<sup>2</sup> De basiseenheid waarmee een taalmodel tekst verwerkt. Een token komt ruwweg overeen met  $\frac{3}{4}$  van een woord (OpenAI, z.d.).

<sup>3</sup> De neurale netwerkarchitectuur die de basis vormt van vrijwel alle moderne taalmodellen.

theoretische contextvensters van 128K tot 1M tokens ligt het effectieve bereik in de praktijk heel wat lager (Hong et al., 2025).

Binnen de ontwikkeling van Lexis werd hier bewust rekening mee gehouden. Chatgesprekken met de AI werden zo kort en gefocust mogelijk gehouden: elk nieuw onderwerp startte een nieuwe chat. Daarnaast werd een AGENTS.md-bestand bijgehouden dat de AI-assistent bij elke sessie voorziet van de projectcontext (zie sectie 1.4.1). Hierdoor hoeft het model deze informatie niet opnieuw op te zoeken in de codebase, wat onnodig tokengebruik en bijhorende prestatiedegradatie vermijdt.

### 1.1.2. Prompting technieken

Binnen prompt engineering zijn er verschillende technieken, elk met een eigen toepassingsdomein en complexiteitsniveau.

#### 1.1.2.1. Zero-shot prompting

Bij zero-shot prompting worden geen voorbeelden meegegeven. Het model baseert zich uitsluitend op zijn voorgetrainde kennis en de instructie in de prompt. Dit is de eenvoudigste vorm van prompting en volstaat voor eenvoudige taken waarbij het model de taak herkent op basis van zijn training, zoals het vertalen van een tekst of het samenvatten van een tekst (Anthropic, z.d.c). Het voordeel is de eenvoud: er is geen voorbereiding van voorbeelddata nodig. Het nadeel is dat het model bij complexere of domeinspecifieke taken minder betrouwbaar is.

#### 1.1.2.2. Few-shot prompting

Few-shot prompting breidt de prompt uit met een klein aantal voorbeelden van de gewenste invoer-uitvoerrelatie, doorgaans twee tot acht paren. Het model leert uit die voorbeelden het verwachte patroon en past dit toe op de nieuwe invoer. Deze techniek is bijzonder effectief wanneer een specifiek outputformaat vereist is of wanneer de taak te specifiek is voor zuiver zero-shot gebruik (Anthropic, z.d.c). Een nadeel is dat de prompt langer wordt, wat meer tokens verbruikt en de latentie verhoogt. Bovendien kan het model onbedoelde patronen uit de voorbeelden overnemen als de steekproef niet representatief is.

#### 1.1.2.3. Chain-of-Thought prompting

Chain-of-Thought (CoT) prompting is een techniek waarbij het model expliciet wordt aangezet om tussenstappen in zijn redenering zichtbaar te maken alvorens een eindantwoord te formuleren. Wei et al. (2023) tonen aan dat dit de prestaties van grote taalmodellen op complexe taken aanzienlijk verbetert. De techniek werkt echter pas effectief bij zeer grote modellen. De techniek wordt toegepast door de gewenste redeneerstappen als few-shot voorbeelden (volgens de vorm: input, chain of thought, output) aan de prompt toe te voegen.

Een verdere evolutie hierop is het zogenoemde *extended thinking of reasoning mode*, waarbij het model intern een uitgebreide redeneerchain doorloopt vóór het antwoordt. Dit wordt besproken in sectie 0.

Plan mode is een praktische toepassing van CoT op het niveau van de ontwikkelomgeving. De agent analyseert eerst het probleem en formuleert een plan, waarna de implementatie volgt. Deze kan soms door een sneller en lichter model worden uitgevoerd. Dit wordt verder besproken in sectie 1.4.2.3.

CoT-prompting werd ook toegepast binnen het Lexis-project, concreet bij het debuggen van de e-mailverwerking. Een uitgewerkt voorbeeld hiervan is te vinden in sectie 1.1.3.2.

#### 1.1.2.4. Structured output prompting

In productieomgevingen is het vaak noodzakelijk dat de output van een LLM machinaal verwerkbaar is. Structured output prompting instrueert het model om te antwoorden in een vooraf bepaald formaat, doorgaans JSON<sup>1</sup> of XML<sup>4</sup>. Onderzoek heeft aangetoond dat LLM's minder geneigd zijn tot hallucinaties<sup>5</sup> wanneer ze in een gestructureerd formaat antwoorden, omdat de structuur zelf de antwoordruimte inperkt. Een lage temperatuurinstelling versterkt dit effect verder door de variatie in de gegenereerde output te beperken (Anthropic, z.d.c).

In het Lexis-project werd hiervoor een dedicated *LLM-wrapper*<sup>6</sup> ontwikkeld: een tussenlaag tussen de C#-backend en het LLM die alle communicatie met het taalmodel centraliseert en beheert. Deze stelt daarnaast een set tools ter beschikking die het model tijdens de verwerking kan aanroepen, zoals het opvragen van productinformatie. Om de output van het model te structureren, wordt gebruikgemaakt van de Python-bibliotheek pydantic-ai, waarmee het verwachte uitvoerformaat expliciet gedefinieerd wordt (Pydantic Services Inc, 2026). Dit garandeert dat de backend steeds een voorspelbaar en verwerkbaar resultaat ontvangt.

#### 1.1.2.5. Role prompting

Bij role prompting krijgt het model via de system prompt een expliciete rol toegewezen, zoals *“een ervaren softwarearchitect”* of *“een strenge code reviewer”*. Voor tekstuele taken waarbij de communicatieve vorm primair staat, zoals het genereren van documentatie of code-commentaar in een specifieke stijl, is dit een effectieve manier om vocabulaire en schrijfstijl te sturen (Anthropic, z.d.c).

Voor exacte, logische taken en complexe codegeneratie blijkt de techniek echter weinig tot geen positieve impact te hebben op de feitelijke accuraatheid (Schulhoff et al., 2024). Het toewijzen van een rol voegt geen extra kennis toe die het model niet al heeft. Bovendien kan een te sterke focus op de rol ertoe leiden dat het model antwoorden geeft die klinken alsof ze kloppen, maar dat niet zijn. Dit verhoogt de kans op hallucinaties (Schulhoff et al., 2024). Binnen het Lexis-project wordt daarom de voorkeur gegeven aan duidelijke, neutrale instructies boven het toewijzen van een persona.

Tabel 1.1 geeft een samenvattend overzicht van de besproken prompt engineering technieken, met voor elke techniek de aanbevolen toepassingssituatie, de voornaamste voordelen en de relevante beperkingen.

---

<sup>4</sup> Een tekst gebaseerde opmaaktaal voor het structureren van gegevens via geneste tags, voornamelijk gebruikt in enterprise-systemen en webservices.

<sup>5</sup> Een hallucinatie in een LLM is het fenomeen waarbij het model vol vertrouwen feitelijk onjuiste, verzonnen of onzinnige informatie genereert die niet gebaseerd is op de werkelijkheid of de verstrekte context.

<sup>6</sup> Een softwarecomponent binnen Lexis die alle communicatie met het taalmodel centraliseert

Tabel 1.1  
Overzicht van prompt engineering technieken

| Techniek          | Wanneer gebruiken                           | Voordelen                             | Beperkingen                                                       |
|-------------------|---------------------------------------------|---------------------------------------|-------------------------------------------------------------------|
| Zero-shot         | Eenvoudige, generieke taken                 | Snel, geen voorbereiding nodig        | Minder precies bij complexe instructies                           |
| Few-shot          | Taken met een vaste invoer-uitvoerstructuur | Hoge consistentie in output           | Voorbeelden verbruiken contextruimte                              |
| Chain-of-Thought  | Debugging, redeneerwerk, complexe logica    | Vermindert hallucinaties, transparant | Trager, meer tokens                                               |
| Structured output | Productieomgevingen, API-integraties        | Machinaal verwerkbare output          | Vereist een schema en is minder consistent zonder Pydantic (2026) |
| Role prompting    | Creatieve taken, toonzetting                | Stijl en perspectief stuurbaar        | Weinig effect bij exacte/logische taken                           |

Opmerking. Eigen werk.

1.1.3. Prompt engineering in de praktijk

In de praktijk is prompt engineering een iteratief proces. Een eerste versie van een prompt levert zelden meteen het gewenste resultaat op. Er worden meerdere cycli van testen, analyseren en verfijnen doorlopen. Aanbevolen werkwijzen zijn onder meer: instructies zo specifiek en ondubbelzinnig mogelijk formuleren, de gewenste outputstructuur expliciet beschrijven, en prompts te testen op een diverse set invoergevallen om robuustheid te beoordelen (Anthropic, z.d.c).

Binnen het Lexis-project werd de system prompt doorheen de ontwikkeling meerdere keren herzien op basis van observaties uit de testdatasets. Dankzij het gebruik van pydantic-ai voor outputdefinitie werden problemen met inconsistente of incorrecte structuren grotendeels vermeden (Pydantic Services Inc, 2026). De bibliotheek dwingt een vast schema af op het niveau van de LLM-wrapper, waardoor de backend op een betrouwbare en uniforme output kan rekenen zonder dat aanvullende verwerkingslogica vereist is. Dit illustreert hoe een goede toolkeuze in een productiecontext de afhankelijkheid van uitgebreide prompt engineering kan verminderen.

Om de theoretische concepten van prompt engineering concreet te maken, worden hieronder twee veelvoorkomende scenario's uit de ontwikkeling van het Lexis-project uitgewerkt. Deze voorbeelden illustreren de iteratieve evolutie van een initiële, ongestructureerde prompt naar een geoptimaliseerde variant die betrouwbaardere en direct inzetbare resultaten oplevert.

1.1.3.1. Scenario 1: Prompt voor backend-code met sterke context

Bij het genereren van backend-code is het cruciaal om het LLM van voldoende architecturale context te voorzien. Een algemene prompt leverde doorgaans een generieke implementatie op die niet verenigbaar was met de complexe autorisatiestructuur van Lexis:

Figuur 1.2  
Prompting scenario 1 – basis

|                                                         |
|---------------------------------------------------------|
| Schrijf een C# API-endpoint om gebruikers in te loggen. |
|---------------------------------------------------------|

Opmerking. Eigen werk.

Door de prompt te verrijken met projectspecifieke voorwaarden (constraints), steeg de bruikbaarheid en kwaliteit van de gegenereerde code sterk. Een geoptimaliseerde prompt specificeerde de architectuur en de exacte flow:

**Figuur 1.3**

*Prompting scenario 1 – geavanceerd*

```
Je schrijft code voor een ASP.NET Core 8 backend met een multi-tenant architectuur.
Implementeer het HTTP GET `/bff/login` endpoint voor de Authorization Code flow met
PKCE. Maak gebruik van de gegenereerde `code_challenge`, voorzie een CSRF-state
parameter ter beveiliging en geef uitsluitend de C#-code terug.
```

*Opmerking.* Eigen werk.

Hierdoor genereerde het taalmodel logica in Cursor die veel sterker was dan met een generieke prompt.

### 1.1.3.2. Scenario 2: Debugging via Chain-of-Thought

Taalmodellen vormen een krachtig hulpmiddel bij het identificeren en oplossen van programmeerfouten, mits ze correct worden aangestuurd. Wanneer de ontwikkelaar simpelweg een foutmelding en een stuk code invoerde, toonde het model de neiging om oppervlakkig te gokken naar de meest voorkomende oorzaak:

**Figuur 1.4**

*Prompting scenario 2 – basis*

```
Mijn applicatie gooit een `invalid_grant` error tijdens de token exchange bij de
autorisatieserver. Hier is mijn C# code. Hoe los ik dit op?
```

*Opmerking.* Eigen werk.

Om dit te voorkomen, werd Chain-of-Thought prompting ingezet. De prompt werd geherformuleerd om het model de code analytisch te laten benaderen:

**Figuur 1.5**

*Prompting scenario 2 – geavanceerd*

```
Controleer in de onderstaande C#-code of de base64-URL-encoding voor de PKCE
`code_verifier` correct is geïmplementeerd volgens de OAuth2 standaard. Denk stap
voor stap na over mogelijke knelpunten bij deze transformatie en formuleer pas daarna
een gerichte oplossing.
```

*Opmerking.* Eigen werk.

Door het model expliciet instructies te geven om tussenstappen te formuleren (“denk stap voor stap na”), werd de validatie van de onderliggende logica aanzienlijk betrouwbaarder.

## 1.2. Model Context Protocol servers

De opkomst van LLM’s heeft een nieuwe generatie AI-tools mogelijk gemaakt die ontwikkelaars ondersteunen bij het schrijven, testen en documenteren van code. Toch blijven deze modellen van nature beperkt tot de informatie die zich binnen hun context window bevindt. Zonder toegang tot externe systemen kunnen zij geen bestanden openen, geen API-requests versturen en geen gegevens opvragen uit projectmanagementtools, zoals later besproken in sectie 1.2.7.1. Het Model Context Protocol (MCP) werd ontwikkeld om precies dit probleem op te lossen: het biedt een gestandaardiseerde manier waarop AI-modellen verbinding maken met externe tools en databronnen, waardoor zij niet langer beperkt zijn tot enkel tekstgeneratie (Anthropic, 2024). Het protocol neemt inspiratie van het Language Server Protocol (LSP) van Microsoft, dat gestandaardiseerde hoe ontwikkelomgevingen ondersteuning voor programmeertalen aanbieden.

Op een vergelijkbare manier standaardiseert MCP hoe AI-applicaties context en tools integreren (Model Context Protocol, 2025).

Dit hoofdstuk bespreekt de oorsprong, architectuur en werking van MCP-servers. Er wordt stilgestaan bij de onderliggende technologieën, de werking van MCP, de rol van MCP binnen de context van een LLM, en de voordelen en beperkingen van het protocol. Tot slot worden twee praktijkvoorbeelden besproken die relevant zijn voor het Lexis-project: de Atlassian MCP-server en de Azure DevOps MCP-server.

### 1.2.1. Introductie tot MCP-servers

Vóór de introductie van MCP moest elke AI-applicatie voor elke externe tool of databron een eigen, op maat gemaakte connector bouwen. Dit leidde tot wat in de industrie het  $M * N$ -probleem wordt genoemd: de combinatorische explosie die ontstaat wanneer  $M$  verschillende AI-modellen elk met  $N$  verschillende tools of databronnen moeten communiceren. Elke combinatie vereiste een afzonderlijke integratie, wat resulteerde in een grote hoeveelheid niet-herbruikbare code en een hoge onderhoudslast (Anthropic, 2024).

Het Model Context Protocol lost dit op door een universele standaard te definiëren. In plaats van voor elke combinatie van AI-model en tool een aparte connector te ontwikkelen, volstaat het om één MCP-server per tool te bouwen en één MCP-client per AI-applicatie. Elke MCP-compatibele client kan vervolgens met elke MCP-compatibele server communiceren, ongeacht de specifieke tool of het specifieke AI-model. Dit reduceert het integratieprobleem van  $M * N$  naar  $M + N$  (Anthropic, 2024).

Een MCP-server is concreet een programma dat context en functionaliteit aanbiedt aan AI-applicaties via het MCP-protocol. De server kan lokaal draaien op dezelfde machine als de AI-applicatie, of op afstand worden gehost als een web-service. Het protocol definieert hoe de AI-applicatie beschikbare tools ontdekt, hoe parameters worden meegegeven bij het aanroepen van een tool, en hoe resultaten worden teruggestuurd (Model Context Protocol, z.d.).

#### 1.2.1.1. Beschikbaarheid en ecosysteem

Het Model Context Protocol werd in november 2024 publiek gelanceerd door Anthropic. Ondertussen ondersteunen meerdere grote AI-platformen het protocol, waaronder Claude Desktop, Claude Code, Cursor, Visual Studio Code (via GitHub Copilot), Windsurf, Cline en Zed (Model Context Protocol, z.d.). Het feit dat MCP een open standaard is, maakt servers universeel: een server die ontwikkeld werd voor gebruik met Claude, werkt eveneens met Cursor of VS Code zonder aanpassingen.

Het aantal beschikbare MCP-servers is in de loop van 2025 en 2026 exponentieel gegroeid. Volgens Anthropic waren er eind 2025 meer dan tienduizend actieve publieke MCP-servers en werden de officiële SDK's maandelijks meer dan 97 miljoen keer gedownload (Anthropic, 2025). Naast deze publieke servers ontwikkelen veel organisaties ook interne MCP-servers die specifiek zijn afgestemd op hun eigen systemen en workflows.

#### 1.2.1.2. Ontwikkelaars en leveranciers

MCP-servers worden ontwikkeld door een breed scala aan partijen. Enerzijds zijn er de officiële servers van grote technologiebedrijven die hun eigen producten toegankelijk maken via MCP. Atlassian biedt bijvoorbeeld een officiële MCP-server aan waarmee AI-modellen rechtstreeks Jira-

tickets en Confluence-pagina's kunnen raadplegen en aanmaken (Atlassian, 2025). Microsoft heeft een MCP-server ontwikkeld voor Azure DevOps, waarmee work items, pull requests en pipelines toegankelijk worden voor AI-agents (Microsoft, 2025b) (zie in sectie 1.2.7).

Anderzijds speelt de open-source community een belangrijke rol. Veel MCP-servers worden ontwikkeld en onderhouden door individuele ontwikkelaars of kleine teams die servers bouwen voor uiteenlopende diensten, van databases en bestandssystemen tot communicatieplatformen en API's. De open specificatie en de beschikbaarheid van SDK's in TypeScript en Python verlagen de drempel om zelf een MCP-server te ontwikkelen aanzienlijk (Model Context Protocol, 2025).

In december 2025 werd het MCP-protocol door Anthropic gedoneerd aan de Agentic AI Foundation (AAIF), een stichting onder de Linux Foundation die mede werd opgericht door Anthropic, Block en OpenAI (Anthropic, 2025). Deze stap bevestigt de ambitie om MCP als een leveranciers-onafhankelijke industriestandaard te positioneren.

## 1.2.2. Onderliggende technologieën

De architectuur van MCP bestaat uit twee lagen: de data laag en de transport laag (Model Context Protocol, z.d.).

De data laag implementeert een protocol gebaseerd op JSON-RPC 2.0, een lichtgewicht Remote Procedure Call-protocol dat gebruikmaakt van JSON als dataformaat (Model Context Protocol, 2025). JSON-RPC definieert drie typen berichten: requests (met een uniek identificatienummer), responses (antwoorden die het identificatienummer van het oorspronkelijke request bevatten) en notifications (eenrichtingsberichten waarvoor geen antwoord wordt verwacht). Deze berichtenstructuur maakt het mogelijk om een gestructureerde, bidirectionele communicatie op te zetten tussen client en server.

De transport laag definieert hoe berichten fysiek worden uitgewisseld. MCP ondersteunt twee transportmechanismen: stdio & streamable HTTP (Hypertext Transfer Protocol) (Model Context Protocol, z.d.).

### 1.2.2.1. Stdio

Stdio (Standard Input/Output) wordt gebruikt voor lokale MCP-servers die als apart proces op dezelfde machine draaien als de AI-applicatie. De communicatie verloopt via de standard input and output van het proces (stdio), zonder netwerkoverhead. Dit mechanisme is geschikt voor tools die lokale bronnen aanspreken, zoals het bestandssysteem of een lokale database (Model Context Protocol, z.d.).

### 1.2.2.2. Streamable HTTP

Streamable HTTP wordt gebruikt voor MCP-servers die op afstand worden gehost. De client stuurt berichten naar de server via HTTP POST-requests. De server kan optioneel Server-Sent Events (SSE)<sup>7</sup> gebruiken om antwoorden te streamen. Dit transportmechanisme maakt het mogelijk om MCP-servers als web servers aan te bieden en ondersteunt standaard HTTP-authenticatiemethoden, waaronder OAuth-tokens en API-sleutels (Model Context Protocol, z.d.).

---

<sup>7</sup> Een webstandaard waarmee een server eenzijdig een stroom berichten naar een client kan sturen over een bestaande HTTP-verbinding, zonder dat de client telkens een nieuw request moet sturen.

De scheiding tussen datalaag en transportlaag zorgt ervoor dat hetzelfde JSON-RPC 2.0 berichten-formaat wordt gebruikt ongeacht het transportmechanisme. Een MCP-server die lokaal via Stdio communiceert, gebruikt dezelfde berichtenstructuur als een server die op afstand via Streamable HTTP bereikbaar is (Model Context Protocol, z.d.).

Tabel 1.2 vergelijkt de twee transportmechanismen op basis van hostinglocatie, communicatie-wijze, opstartgedrag en typisch gebruik.

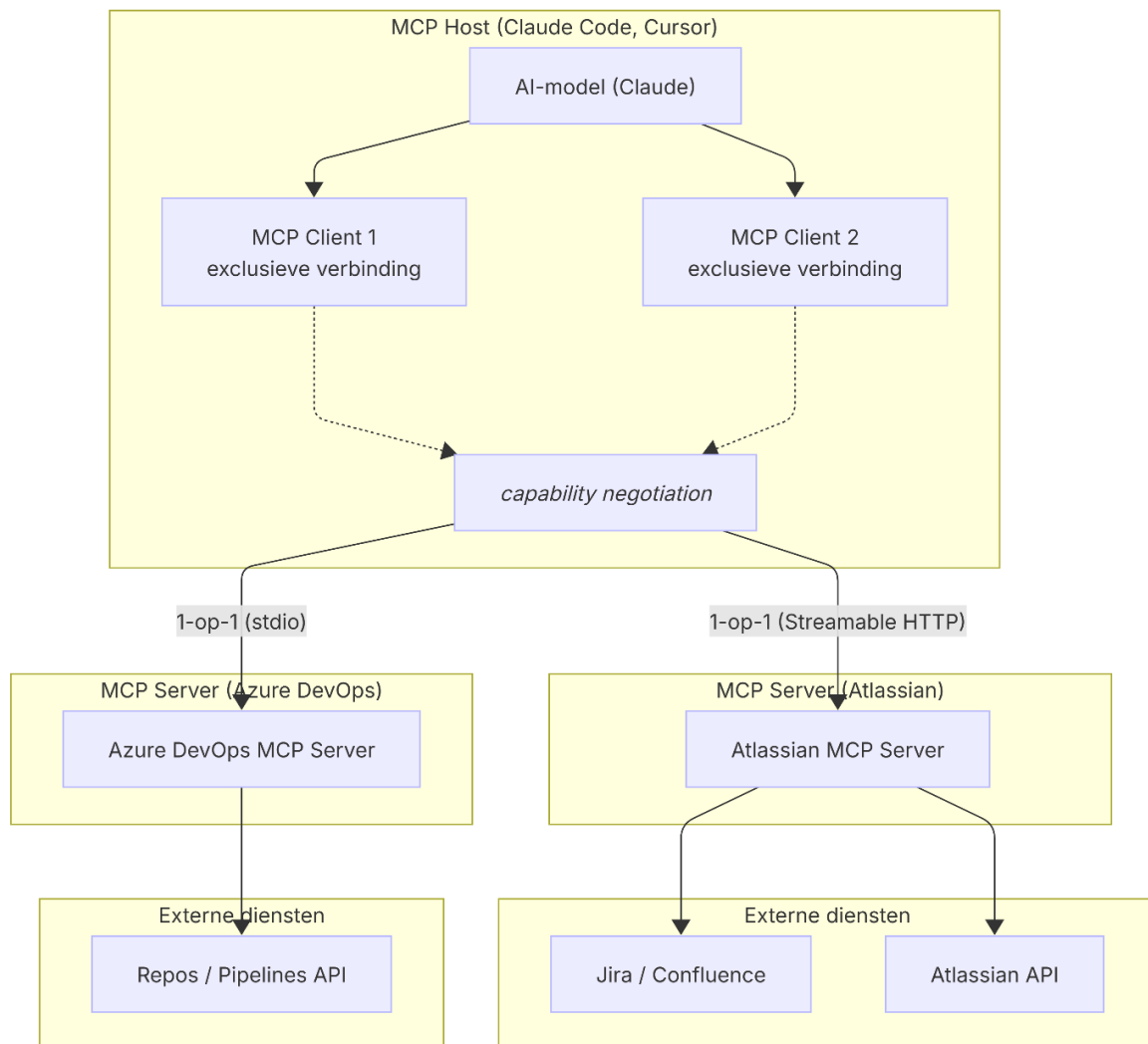
Tabel 1.2  
Vergelijking van MCP-transportmechanismen

|                | Stdio                         | Streamable HTTP                     |
|----------------|-------------------------------|-------------------------------------|
| Hostinglocatie | Lokaal (zelfde machine)       | Remote (extern gehost)              |
| Communicatie   | Via standaardinvoer/-uitvoer  | Via HTTP-requests                   |
| Opstart        | Als apart proces door de host | Draait onafhankelijk als webservice |
| Gebruik        | Lokale tools, filesystem, CLI | Cloud-diensten, SaaS-platformen     |

Opmerking. Eigen werk, gebaseerd op *Model Context Protocol* (z.d.). <https://modelcontextprotocol.io/docs/learn/architecture>. Copyright z.d., Model Context Protocol a Series of LF Projects, LLC.

### 1.2.3. Werking van MCP

De MCP-architectuur definieert drie deelnemers: de *MCP Host*, de *MCP Client* en de *MCP Server* (Model Context Protocol, z.d.). De MCP host is de AI-applicatie die de interactie coördineert, zoals Claude Code of Cursor (Anthropic, 2026a; Anysphere, 2026). Voor elke MCP Server waarmee de host verbinding maakt, wordt een afzonderlijke MCP Client opgestart. Deze client onderhoudt een exclusieve verbinding met de bijbehorende server. Eén host kan gelijktijdig meerdere clients beheren, elk verbonden met een andere server.

**Figuur 1.6***Architectuur van een MCP-integratie met twee servers*

*Opmerking.* Eigen werk, gebaseerd op *Model Context Protocol (z.d.)*. <https://modelcontextprotocol.io/docs/learn/architecture>. Copyright z.d., Model Context Protocol a Series of LF Projects, LLC.

De verbinding tussen client en server begint met een initialisatiefase waarin beide partijen hun mogelijkheden uitwisselen via een zogenaamde *capability negotiation* (Model Context Protocol, 2025). De client declareert welke functies hij ondersteunt, en de server antwoordt met zijn eigen mogelijkheden. Dit mechanisme zorgt ervoor dat client en server alleen functies gebruiken die door beide partijen worden ondersteund. De onderliggende verbinding verloopt via één van de transportprotocollen die besproken werden in sectie 1.2.2. In Figuur 1.6 gebruikt de Atlassian MCP Server Streamable HTTP, terwijl de Azure DevOps MCP Server via stdio communiceert.

Eens de verbinding is opgezet, kan elke MCP Server acties uitvoeren op externe diensten. De Azure DevOps MCP Server biedt toegang tot Boards, Repositories en de Pipelines API; de Atlassian MCP Server ontsluit Jira, Confluence en de Atlassian API.

Na de initialisatiefase identificeert de client de beschikbare functionaliteiten van de server. Het MCP definieert hierbij drie concepten (Model Context Protocol, 2025).

- Tools zijn uitvoerbare functies die de AI-applicatie kan aanroepen om acties uit te voeren, zoals het versturen van een API-request, het uitvoeren van een databasequery of het aanmaken van

een bestand. Elke tool heeft een naam, een beschrijving en een invoerschema dat de verwachte parameters definieert (Model Context Protocol, 2025).

- Resources zijn databronnen die contextuele informatie leveren, zoals bestandsinhouden, databaserecords of API-antwoorden. Resources worden opgevraagd via een URI en zijn bedoeld om het AI-model van relevante achtergrondinformatie te voorzien (Model Context Protocol, 2025).
- Prompts zijn herbruikbare sjablonen die helpen bij het structureren van interacties met het taalmodel, zoals systeemprompts of few-shot voorbeelden (Model Context Protocol, 2025).

Elk onderdeel heeft bijbehorende methoden voor ontdekking en gebruik. Een client kan bijvoorbeeld eerst alle beschikbare tools opvragen via een `tools/list-request`, en vervolgens een specifieke tool aanroepen via `tools/call` met de juiste parameters. De server verwerkt de request en stuurt het resultaat terug als een gestructureerd antwoord (Model Context Protocol, z.d.).

De interactie verloopt concreet als volgt. Wanneer een gebruiker een vraag stelt aan het AI-model, analyseert het model de vraag en bepaalt of een externe tool nodig is om de vraag te beantwoorden. Indien dat het geval is, genereert het model een *tool-call* met de juiste parameters. De AI-applicatie onderschept deze tool-call, stuurt de request naar de juiste MCP-server, ontvangt het resultaat en voegt dit toe aan de conversatie. Het AI-model verwerkt vervolgens het resultaat en formuleert een antwoord voor de gebruiker (Model Context Protocol, z.d.).

Dit mechanisme wordt ook wel “*tool use*” of “*function calling*” genoemd. Het AI-model kiest zelf wanneer en welke tool het aanroept, op basis van de gebruikersvraag en de beschrijvingen van de beschikbare tools. De server heeft hierbij geen directe interactie met het taalmodel: alle communicatie verloopt via de MCP-client die door de host wordt beheerd (Model Context Protocol, z.d.).

#### 1.2.4. MCP binnen de context van een LLM

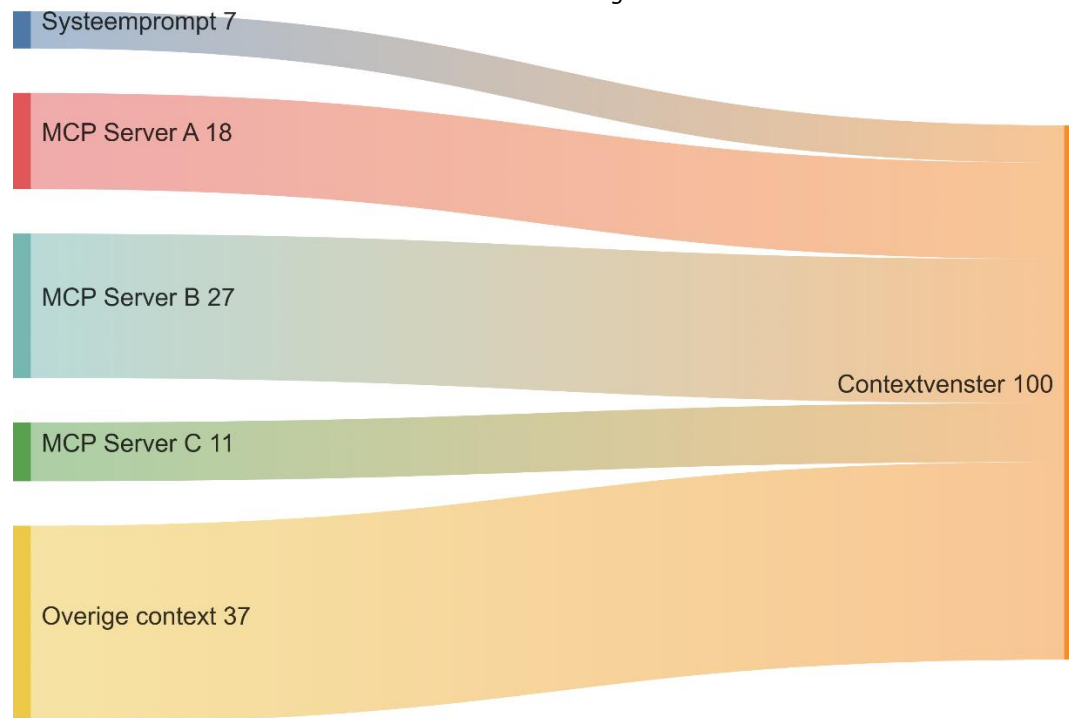
Zoals besproken in sectie 1.1.1 beschikt een LLM over een contextvenster met een beperkte effectieve capaciteit.

Wanneer een MCP-client verbinding maakt met een of meerdere servers, worden de tool- en resourcebeschrijvingen van die servers in het contextvenster geladen (Model Context Protocol, z.d.). Het model gebruikt deze beschrijvingen om te bepalen welke tool relevant is voor een gebruikersvraag en hoe de parameters correct samengesteld moeten worden.

Het aantal verbonden MCP-servers heeft daardoor een directe invloed op de beschikbare ruimte in het contextvenster. Figuur 1.7 illustreert dit aan de hand van een hypothetisch voorbeeld: zelfs vóór de eerste gebruikersinvoer nemen de systeemprompt en toolbeschrijvingen al een deel van de beschikbare ruimte in. Bij een groot aantal servers of uitgebreide toolschema's kan dit aandeel aanzienlijk oplopen (zie sectie 1.2.6).

**Figuur 1.7**

Hypothetisch voorbeeld dat demonstreert hoe MCP wordt ingeladen in het contextvenster van een LLM



Opmerking. Eigen werk.

De weergegeven aandelen in Figuur 1.7 zijn relatieve, illustratieve waarden en geven geen exacte tokentellingen weer; ze dienen enkel om de verhoudingen conceptueel te visualiseren.

### 1.2.5. Voordelen van een MCP

Het Model Context Protocol biedt verschillende voordelen ten opzichte van traditionele, op maat gemaakte integraties.

Ten eerste biedt MCP standaardisatie. Door een uniform protocol te hanteren voor de communicatie tussen AI-modellen en externe tools, wordt de noodzaak voor aangepaste connectoren per combinatie van model en tool geëlimineerd. Dit vereenvoudigt de ontwikkeling en het onderhoud van integraties aanzienlijk (Anthropic, 2024).

Ten tweede is de herbruikbaarheid van MCP. Een MCP-server die voor een bepaalde tool wordt ontwikkeld, is bruikbaar vanuit elke MCP-compatibele AI-applicatie. Dit voorkomt dubbel werk en maakt het mogelijk om eenmaal ontwikkelde integraties breed in te zetten (Anthropic, 2024).

Ten derde ondersteunt MCP dynamische ontdekking van tools. In tegenstelling tot traditionele API-integraties, waarbij de beschikbare eindpunten en parameters op voorhand bekend moeten zijn, kan een MCP-client bij het verbinden met een server runtime opvragen welke tools beschikbaar zijn (Anthropic, z.d.c). Dit maakt het systeem flexibeler en adaptief: nieuwe tools kunnen worden toegevoegd zonder dat de AI-applicatie opnieuw moet worden geconfigureerd of geïnstalleerd.

Ten vierde biedt MCP een beveiligingsmodel dat gebaseerd is op expliciete toestemming van de gebruiker. De specificatie schrijft voor dat gebruikers controle behouden over welke gegevens worden gedeeld en welke acties worden uitgevoerd (Model Context Protocol, 2025). Tools worden pas aangeroepen na goedkeuring, en de host-applicatie fungeert als poortwachter die alle communicatie tussen het taalmodel en externe servers beheert.

### 1.2.6. Beperkingen van een MCP

Ondanks de voordelen kent het Model Context Protocol ook beperkingen die relevant zijn voor productieomgevingen.

Het protocol is relatief jong. De eerste publieke specificatie dateert van november 2024, en hoewel de adoptie snel verloopt, zijn er nog aspecten die in ontwikkeling zijn (Anthropic, 2024). De foutafhandeling is op protocolniveau beperkt: wanneer een MCP-server een fout retourneert, is het aan het AI-model om daar zinvol mee om te gaan, wat niet altijd betrouwbaar verloopt.

De kwaliteit en betrouwbaarheid van MCP-servers varieert sterk. Officiële servers van grote leveranciers zoals Atlassian of Microsoft worden doorgaans goed onderhouden, maar community-servers bieden minder garanties op het vlak van stabiliteit, beveiliging en documentatie. Bij het gebruik van externe MCP-servers moet rekening worden gehouden met potentiële beveiligingsrisico's, aangezien de server toegang kan krijgen tot gevoelige gegevens (Model Context Protocol, 2025).

Daarnaast zijn er schaalbaarheidsuitdagingen. Grote hoeveelheden data, paginatie van resultaten en complexe datatransformaties zijn moeilijk te realiseren via het huidige MCP-protocol. Voor dergelijke scenario's blijft een directe API-integratie in de praktijk vaak de meest geschikte keuze.

Tot slot, en misschien één van de belangrijkste, neemt elke geconfigureerde MCP-server ruimte in binnen het context window van het taalmodel, zoals eerder beschreven (zie sectie 1.2.4; Figuur 1.7). Servers die geladen worden maar tijdens een sessie nooit effectief aangesproken worden, verbruiken zo onnodig contextruimte. Bij een groot aantal geconfigureerde servers kan dit de prestaties en de kwaliteit van de antwoorden negatief beïnvloeden (zie sectie 1.1.1).

Het is mede vanwege deze beperkingen dat aanvullende concepten zoals skills zijn ontstaan, die worden besproken in volgende sectie 1.3.

### 1.2.7. Praktijkvoorbeelden van MCP-servers

De voorgaande secties beschreven de architectuur, werking en voor- en nadelen van MCP-servers vanuit een theoretisch perspectief. Om de meerwaarde in een concrete ontwikkelomgeving te illustreren, worden in deze sectie twee praktijkvoorbeelden uitgewerkt: de Atlassian MCP-server, die toegang biedt tot Jira en Confluence, en de Azure DevOps MCP-server van Microsoft. Tot slot toont een gecombineerd praktijkscenario hoe meerdere MCP-servers gezamenlijk ingezet kunnen worden om een volledige ontwikkelcyclus te delegeren aan een AI-agent. Beide servers werden ook effectief ingezet binnen het Lexis-project.

#### 1.2.7.1. Atlassian

Atlassian is een Australisch softwarebedrijf dat een reeks samenwerkingstools ontwikkelt voor softwareteams. De twee meest gebruikte producten zijn Jira, een projectmanagementtool voor het bijhouden van taken, sprints en bugs, en Confluence, een kennismanagementplatform voor het documenteren van projecten en processen.

Atlassian biedt een officiële Remote MCP Server aan waarmee AI-modellen rechtstreeks kunnen communiceren met Jira en Confluence (Atlassian, 2025). Via deze server kan een AI-model Jira-tickets opvragen, aanmaken en bijwerken, en Confluence-pagina's raadplegen of genereren. De server maakt gebruik van OAuth-authenticatie en respecteert de bestaande gebruikersrechten: een

AI-model kan alleen die gegevens benaderen waartoe de gebruiker zelf toegang heeft binnen de Atlassian-omgeving (Atlassian, 2025).

In het Lexis-project werd de Atlassian MCP-server gebruikt om vanuit de AI-ontwikkelomgeving rechtstreeks Jira-stories te raadplegen en Confluence-documentatie te doorzoeken, zonder de ontwikkelomgeving te verlaten. Dit vermindert het wisselen tussen applicaties en versnelt het opzoeken van projectinformatie.

### 1.2.7.2. Azure DevOps

Azure DevOps is een platform van Microsoft dat een geïntegreerde reeks van diensten biedt voor softwareontwikkeling, waaronder versiebeheer (Git-repositories), CI/CD-pipelines en pull request management.

Microsoft heeft een officiële MCP-server voor Azure DevOps ontwikkeld die in oktober 2025 algemeen beschikbaar werd (Microsoft, 2025a). Via deze server kunnen AI-modellen pull requests raadplegen en beoordelen, pipelineresultaten inspecteren en repository-informatie opvragen (Microsoft, 2025a). De server ondersteunt toegangscontrole: beheerders kunnen bepalen welke tools een AI-agent mag gebruiken, zodat een agent die een repository mag lezen niet noodzakelijk pipelines kan wijzigen.

Binnen het Lexis-project werd de Azure DevOps MCP-server ingezet voor het aanmaken en reviewen van pull requests vanuit de AI-ontwikkelomgeving. Door deze integratie konden veelvoorkomende DevOps-taken, zoals het opzoeken van de status van een pipeline of het aanmaken van een pull request, worden uitgevoerd zonder de ontwikkelomgeving te verlaten.

### 1.2.7.3. Praktijkscenario

De ware kracht van MCP manifesteert zich niet bij het gebruik van één geïsoleerde tool, maar wanneer meerdere MCP-servers worden gecombineerd binnen één instructie. Om dit mogelijk te maken, worden de servers geconfigureerd in bijvoorbeeld een JSON-bestand dat door de AI-ontwikkelomgeving wordt ingelezen. Volgende configuratie is een voorbeeld van hoe MCP-servers, vaak achterliggend, worden geconfigureerd:

**Figuur 1.8**

*Voorbeeld van een MCP-configuratie bestand*

```
{
  "mcpServers": {
    "Atlassian": {
      "url": "https://mcp.atlassian.com/v1/mcp",
      "headers": {}
    },
    "Azure DevOps": {
      "command": "npx -y @azure-devops/mcp target",
      "env": {},
      "args": []
    }
  }
}
```

*Opmerking.* Eigen werk.

Naast het manueel bewerken van dit configuratiebestand bieden sommige AI-ontwikkelomgevingen ook een grafische interface aan waarmee MCP-servers via een instellingenscherm kunnen worden toegevoegd. De onderliggende werking blijft identiek: de UI genereert het configuratiebestand automatisch op de achtergrond.

De configuratie illustreert de twee transportmechanismen uit de MCP-architectuur in de praktijk. De Atlassian-server communiceert via Streamable HTTP (zie sectie 1.2.2.2): de AI-applicatie stuurt requests naar een extern beschikbare URL. De Azure DevOps-server daarentegen wordt lokaal opgestart als een apart proces via npx, waarbij de communicatie verloopt via stdio (zie sectie 1.2.2.1). Beide servers zijn vanuit de AI-ontwikkelomgeving gelijktijdig beschikbaar, zonder dat daarvoor afzonderlijke integraties per tool vereist zijn.

Met beide servers actief volstaat één instructie om een volledige ontwikkelcyclus, van taakopvraging tot rapportering, te delegeren aan het AI-model:

#### Figuur 1.9

*Voorbeeld van een prompt die MCP-servers zal aanroepen*

```
<https://cerm.atlassian.net/browse/LEXIS-50> Implement this feature in a new branch and open a pull request on Azure. Then put a comment on the issue describing how you implemented it.
```

*Opmerking.* Eigen werk.

Deze eenvoudige instructie activeert een keten van vier opeenvolgende tool calls. Eerst roept het AI-model de Atlassian MCP-server aan om het ticket LEXIS-50 op te halen en de vereisten te lezen. Vervolgens maakt het via de Azure DevOps MCP-server een nieuwe branch aan en commit het de implementatie. Daarna opent het een pull request van de nieuwe branch naar de master branch. Tot slot plaatst het een comment op het originele Jira-ticket met een samenvatting van de implementatie, opnieuw via de Atlassian MCP-server. De ontwikkelaar formuleert het eindresultaat; het model bepaalt de weg ernaartoe.

## 1.3. Skills

Na de bespreking van Model Context Protocol-servers en hun beperkingen, komt een aanvullend concept aan bod dat specifiek inspeelt op de gevallen waar MCP te zwaar of te complex is: skills. In deze sectie wordt uitgelegd wat skills zijn, hoe ze zijn ontstaan en wat hun meerwaarde is binnen een AI-ondersteunde ontwikkel-workflow. Vervolgens wordt aan de hand van concrete voorbeelden uit het Lexis-project getoond hoe skills in de praktijk worden ingezet, en wordt een vergelijking gemaakt met MCP-servers om duidelijk te maken wanneer welk concept wordt toegepast.

### 1.3.1. Introductie tot skills

Naarmate AI-assistenten meer taken kunnen overnemen in de softwareontwikkeling moet er nagedacht worden over hoe we domeinspecifieke kennis efficiënt en herbruikbaar aan een model kunnen overdragen. Model Context Protocol-servers bieden hiervoor een krachtige maar zware oplossing: ze vereisen configuratie, authenticatie en verbruiken soms onnodige ruimte in het contextvenster. Voor taken waarbij geen live databron of externe API nodig is, maar wel gespecialiseerde instructies, biedt het concept van skills een alternatief.

Een skill is een herbruikbaar instructiebestand dat een AI-agent contextspecifieke kennis en werkwijzen meegeeft voor een welomschreven taak. In tegenstelling tot een MCP-server vereist een skill geen netwerkverbinding, geen toolregistratie en geen authenticatieflow. Een skill bestaat in essentie uit tekst: instructies, richtlijnen, voorbeelden en sjablonen, samengebracht in een of meerdere Markdown-bestanden (Scheire et al., 2026; Anthropic, z.d.a).

Het concept werd geïntroduceerd binnen het ecosysteem van Claude Code, een software tool van Anthropic in de *command line interface* (CLI) voor AI-gestuurde softwareontwikkeling. Skills worden

daar ingezet om de agent te voorzien van domeinkennis die te specifiek is voor een generiek taalmodel, maar te licht is om als MCP-server te implementeren (Anthropic, z.d.a). In bredere zin sluiten skills aan bij het idee van *prompt libraries*: gestructureerde, herbruikbare promptsjablonen die in een workflow worden geïntegreerd (Schulhoff et al., 2024).

### 1.3.2. Ontstaan en evolutie

Skills zijn ontstaan als een directe reactie op de beperkingen van MCP-servers (zie sectie 1.2.6): context window verbruik, variabele kwaliteit van community-servers en de beheerslast van server-verbindingen. Een skill introduceert geen extra toolbeschrijvingen in het context window, is volledig transparant als leesbare tekst, en kan worden gedeeld als een gewone map zonder aanvullende infrastructuur.

Skills vormen daarmee geen vervanging voor MCP-servers, maar een aanvullend concept: waar MCP-servers dienen voor dynamische, actieve interacties met externe systemen, zijn skills geschikt voor het overdragen van statische domeinkennis en procedurele instructies.

### 1.3.3. Opbouw van een skill

Een skill is opgebouwd rond een centraal Markdown-bestand, meestal noemt deze `SKILL.md`. Deze dient als toegangspunt voor de AI-agent. Dit bestand bevat de volledige instructieset voor de betrokken taak: een omschrijving van het doel, de te volgen werkwijze, relevante context, eventuele beperkingen en concrete voorbeelden van gewenste uitvoer.

De structuur van een skill is bewust eenvoudig gehouden. Markdown biedt voldoende opmaakmogelijkheden, zoals koppen, lijsten, tabellen en codeblokken, om complexe instructies helder en gelaagd te presenteren, zonder dat daarvoor een apart bestandsformaat of tooling nodig is. Een ervaren ontwikkelaar kan een skill schrijven, lezen en bijwerken met niets meer dan een tekst editor.

#### Figuur 1.10

*Fragment uit de team-workflow-skill, ontwikkeld voor het Lexis-project*

```
---
name: team-workflow
description: Execute any non-trivial codebase task using a structured
  4-phase workflow. Use this skill whenever the user asks to add, remove,
  refactor, or migrate something that spans multiple projects or files –
  e.g. "remove X", "migrate X to Y", "add X everywhere". Always use this
  skill for cross-cutting changes, even if the user doesn't explicitly
  ask for a team approach.
---

## CRITICAL: Orchestration-Only Rule

**You are the orchestrator. You do NOT do the work yourself.**

- NEVER use Read, Grep, Glob, or Bash to investigate the codebase. Spawn agents for that.
- NEVER use Edit or Write on source code. Spawn agents for that.

## Phase 1 – Discover

Use `TeamCreate` to spawn one Explore agent per logical area of the solution.
Run all agents in parallel.
```

*Opmerking.* Eigen werk.

Figuur 1.10 toont een fragment uit de team-workflow-skill die werd ontwikkeld voor het Lexis-project. In het YAML-blok bovenaan bevindt zich het `description` veld: dit is de tekst die de agent gebruikt om te bepalen of de skill van toepassing is. De beschrijving somt expliciete activatiepatronen op ("*remove X*", "*migrate X to Y*", "*add X everywhere*") zodat de agent de skill automatisch inlaadt wanneer de taakomschrijving van de gebruiker overeenkomt met een van die patronen. De gebruiker hoeft de skill nooit bij naam te noemen.

De "Orchestration-Only Rule" legt vervolgens vast hoe de agent de taak uitvoert. Door de agent te verbieden zelf broncode te lezen of te bewerken, wordt de orkestrerende agent gedwongen om sub-agents op te spinnen voor elk afgebakend deel van het werk. Die sub-agents kunnen parallel worden uitgevoerd, wat bij grootschalige wijzigingen een aanzienlijke tijdswinst oplevert ten opzichte van sequentiële verwerking.

Het resultaat is een werkwijze die bij elke gelijksoortige taak op dezelfde manier verloopt: gestructureerd, traceerbaar en onafhankelijk van wie de taak start of wanneer. De kennis over hoe zo'n taak het best wordt aangepakt, is niet langer impliciet in het hoofd van een ontwikkelaar, maar expliciet vastgelegd in een bestand dat door het hele team kan worden gedeeld en in versiebeheer kan worden opgenomen.

Naast het centrale instructiebestand kan een skill ondersteunende bestanden bevatten: sjablonen voor terugkerende outputs, configuratiebestanden, voorbeeldinput of referentiemateriaal. Al deze bestanden worden samengebracht in één map, die als geheel de skill vormt. Voor distributie kan de map worden gecomprimeerd tot een zip-bestand of worden opgenomen in een bredere plugin. Een plugin is een bundel van meerdere skills, MCP-configuraties en projectinstellingen die als één eenheid kan worden geïnstalleerd (Anthropic, z.d.a).

De agent activeert een skill op het moment dat de taakomschrijving overeenkomt met de beschrijving van de skill, of wanneer de gebruiker dit expliciet aanvraagt. Op dat moment leest de agent de inhoud van `SKILL.md` en past de instructies toe op de verdere uitvoering van de taak. Dit mechanisme vertoont gelijkenissen met few-shot prompting (zie sectie 1.1.2.2), waarbij voorbeelden en context worden meegegeven om het modelgedrag te sturen, maar dan in een modulaire, herbruikbare en versiebeerbare vorm (Schulhoff et al., 2024; Anthropic, 2026a).

Een belangrijk onderscheid met een system prompt is de modulariteit. Een system prompt is globaal en geldt voor de volledige sessie. Een skill is contextgebonden: ze wordt enkel ingeladen wanneer relevant, waardoor het context window efficiënter wordt benut.

Dit activatiemechanisme is een van de meest onderscheidende kenmerken van skills ten opzichte van manuele promptinstructies: de kennis wordt niet elke keer opnieuw meegegeven door de gebruiker, maar is ingebakken in de infrastructuur van de workflow.

### 1.3.4. Toepassing in de praktijk

De concrete meerwaarde van skills manifesteert zich op drie vlakken: consistentie van output, transparantie van werkwijze en herbruikbaarheid over projecten heen. Eenmaal een skill is geschreven, verloopt elke gelijksoortige taak op dezelfde gestructureerde manier, ongeacht de ontwikkelaar die de taak indient of de sessie waarin ze wordt uitgevoerd.

Een concreet voorbeeld hiervan is de team-workflow-skill die werd ontwikkeld voor gebruik in het Lexis-project. Deze skill richt zich op complexe, domeinoverschrijdende codeer taken die meerdere projecten of bestanden raken, zoals het verwijderen van een component, het migreren van een patroon of het toevoegen van een feature aan meerdere lagen van de applicatie tegelijk.

De skill werd opgesteld als een Markdown-instructiebestand dat de agent verplicht een gestructureerd vierfasenproces te volgen: eerst worden alle impactpunten in de codebase in kaart gebracht, vervolgens wordt een gedetailleerd uitvoeringsplan opgesteld, daarna wordt de wijziging stapsgewijs geïmplementeerd, en ten slotte wordt het resultaat gevalideerd via een interne review. Elke fase schrijft zijn resultaat weg naar een vaste locatie in het project, zodat de voortgang traceerbaar en herstelbaar is.

Een bijzonder kenmerk van deze skill is dat de ontwikkelaar het nooit expliciet hoeft te activeren. De beschrijving van de skill omvat een reeks patroonherkenningscriteria, zoals woorden als “verwijder”, “migreer” of “voeg toe aan”. Wanneer de taakomschrijving van de gebruiker overeenkomt met een van deze criteria, activeert de agent de skill automatisch. Dit maakt het gebruik voor de ontwikkelaar volledig transparant: men beschrijft de taak in gewone taal, en de skill zorgt er achter de schermen voor dat de juiste aanpak wordt gevolgd.

Een ander opvallend aspect is de strikte scheiding tussen orkestratie<sup>8</sup> en uitvoering. De skill instrueert de agent uitdrukkelijk om zelf geen broncode te lezen of te bewerken. In de plaats daarvan spint de agent per afgebakend deel van de codebase een sub-agent op die het onderzoek of de implementatie op zich neemt. Op die manier wordt de parallelisatiemogelijkheid van agent teams benut, terwijl de orkestrerende agent het overzicht bewaart en de samenhang tussen de deelresultaten bewaakt.

De kracht van een skill neemt verder toe wanneer ze wordt gecombineerd met MCP-servers. Zo kan een skill de agent instrueren over de werkwijze, de stijl en de vereisten van een taak, terwijl een MCP-server zorgt voor de data die daarvoor nodig is. In de praktijk bij CERM werd deze combinatie gebruikt: de Atlassian MCP-server leverde de Jira-storygegevens aan, terwijl een skill de agent instrueerde over de gewenste structuur en taal van de documentatie. Agents worden verder beschreven in sectie 1.4.

### 1.3.5. Vergelijking met MCP-servers

Skills en MCP-servers vullen elkaar aan, maar zijn fundamenteel verschillend in aard en toepassingsgebied. Tabel 1.3 geeft een overzicht van de voornaamste verschillen.

**Tabel 1.3**

*Vergelijking tussen skills en MCP-servers*

| criterium                      | MCP-server                             | Skill                            |
|--------------------------------|----------------------------------------|----------------------------------|
| <b>Vereiste infrastructuur</b> | Geen (lokaal) / Server + auth (remote) | Geen                             |
| <b>Context window gebruik</b>  | Hoog (tool- en resourcedefinities)     | Laag (enkel tekst bij activatie) |
| <b>Type kennis</b>             | Dynamisch (live data, API-calls)       | Statisch (instructies)           |
| <b>Deelbaarheid</b>            | Configuratiebestand vereist            | Map of zip-bestand               |
| <b>Onderhoud</b>               | Server-side update vereist             | Markdown-bestand bewerken        |

*Opmerking.* Lokale MCP-servers vereisen geen aparte serverinfrastructuur en draaien als subprocess op de gebruikersmachine. Eigen werk.

Het onderscheid ligt voornamelijk in het type kennis dat wordt overgedragen. Een MCP-server is geschikt wanneer de agent actief met een extern systeem moet interageren: bestanden ophalen,

<sup>8</sup> De term orkestratie verwijst hier naar het aansturen van sub-agents door een overkoepelende agent. Dit concept wordt uitgebreid behandeld in sectie 1.4.4.

tickets aanmaken, code uitvoeren. Een skill is geschikt wanneer de agent moet beschikken over taakgerichte of domeinspecifieke kennis die niet van buitenaf hoeft te worden opgevraagd.

Een praktisch beslissingscriterium is de volgende vraag: vereist de taak een actieve verbinding met een extern systeem, of volstaat het overdragen van kennis en instructies? In het eerste geval is een MCP-server de aangewezen keuze. In het tweede geval biedt een skill een eenvoudigere, efficiëntere en beter beheersbare oplossing.

Het is ook mogelijk om beide te combineren. Dit wordt in de praktijk ook vaak gedaan. De MCP-server levert de ruwe data aan, de skill stuurt de agent in zijn verwerking en presentatie van die data. Op die manier wordt het voordeel van beide concepten benut, terwijl de nadelen worden beperkt (Anthropic, z.d.a).

## 1.4. Agents

Waar de voorgaande secties focusten op het sturen en uitbreiden van een taalmodel via prompts, contextbeheer, MCP-servers en skills, introduceert het concept van een AI-agent een fundamenteel andere werkwijze. Een agent is een AI-systeem dat autonoom een reeks acties kan plannen en uitvoeren om een doel te bereiken, zonder dat voor elke stap menselijke sturing vereist is. In het kader van deze bachelorproef werd Claude Code van Anthropic gebruikt als primaire agentomgeving: een command-line tool die een taalmodel inzet als actieve uitvoerder van softwareontwikkelingstaken, met toegang tot het bestandssysteem, de terminal en externe services via MCP-servers (Anthropic, 2026a).

Het onderscheid met gewone LLM-interactie is belangrijk. Bij traditioneel taalmodelgebruik is de mens de regisseur: elke actie wordt expliciet gevraagd en het model antwoordt. Bij een agent verschuift die rol. De mens formuleert een doel; de agent bepaalt zelf welke stappen nodig zijn, in welke volgorde, en welke tools daarvoor worden ingezet. Fouten worden door de agent zelf gedetecteerd en gecorrigeerd.

Concrete kenmerken van een agent zijn autonome taakplanning, het gebruik van tools zoals MCP-servers en bash-commando's, geheugen over meerdere stappen heen, en het vermogen tot zelfcorrectie bij fouten.

### 1.4.1. AGENTS.md

Naarmate agents taken uitvoeren die meerdere sessies kunnen overbruggen, is de vraag hoe de agent bij elke nieuwe sessie snel over de nodige projectcontext kan beschikken. Een taalmodel heeft immers geen geheugen tussen sessies in: zonder aanvullende mechanismen moet de agent bij elke herstart opnieuw de codebase verkennen en werkafspraken afleiden uit de broncode.

De gangbare oplossing is een agentgeheugenbestand: een Markdown-bestand dat bij het opstarten van een sessie automatisch wordt ingeladen en de agent voorziet van actuele projectcontext. Verschillende omgevingen hanteren hiervoor een eigen bestandsnaam, zoals CLAUDE.md of AGENTS.md, maar het onderliggende principe is identiek. De inhoud omvat doorgaans een architectuurbeschrijving, de gebruikte technologiystack, terugkerende commando's en codeerstijlafspraken. Het bestand is bewust beperkt in omvang om onnodig context window gebruik te vermijden (zie sectie 1.1.1) (Anthropic, z.d.a).

Binnen het Lexis-project werd gedurende de volledige bachelorproefperiode een AGENTS.md-bestand bijgehouden in de projectroot. Het bevatte de projectarchitectuur, de relatie tussen de C#-

backend, de Python LLM-wrapper en de Azure-infrastructuur, en een doorlopend bijgewerkte sectie met de focus van de lopende sprint. Dit verkortte de opstarttijd van elke werksessie en zorgde voor consistentere uitvoer door sessies heen.

### 1.4.2. Agent modi

AI-agents in software zoals Claude Code bieden verschillende operationele modi die bepalen hoeveel autonomie de agent krijgt en op welk moment menselijke goedkeuring vereist is. De keuze voor een modus is een bewuste afweging tussen controle, snelheid en risico, en bepaalt in welke mate de ontwikkelaar tussenbeide komt tijdens de uitvoering. Afhankelijk van de taak en de context kiest de ontwikkelaar een modus die aansluit bij de vereiste mate van toezicht.

#### 1.4.2.1. Ask mode

In ask mode analyseert de agent de codebase, leest bestanden en beantwoordt vragen, maar voert geen enkele actie uit. Er worden geen bestanden gewijzigd, geen commando's uitgevoerd en geen externe systemen aangesproken. De agent functioneert als een interactieve gesprekspartner die op basis van de beschikbare context redeneert en aanbevelingen formuleert (Anthropic, 2026a).

Deze modus is het aangewezen vertrekpunt wanneer een taak nog onvoldoende gedefinieerd is, of wanneer de ontwikkelaar de redenering van de agent wil toetsen alvorens uitvoeringsrechten toe te kennen. Binnen het Lexis-project werd ask mode ingezet om onbekende codedelen snel te verkennen, de architectuur van bestaande modules te begrijpen en de aanpak van een nieuwe feature af te stemmen. Dit zonder risico op ongewenste wijzigingen.

#### 1.4.2.2. Agent mode

Agent mode is de standaardmodus en biedt de agent volledige uitvoeringsrechten. De agent kan bestanden lezen en schrijven, shell-commando's uitvoeren, tools aanroepen via MCP-servers en de resultaten van die acties verwerken in volgende stappen. De gebruiker formuleert een doel; de agent bepaalt zelf de aanpak, voert uit en corrigeert fouten zonder verdere sturing (Anthropic, 2026a).

Het voordeel is de snelheid waarmee een complexe taak van begin tot einde kan worden afgewerkt. Het risico is dat een fout vroeg in de uitvoering doorwerkt in latere stappen. Agent mode is het meest geschikt voor taken met een duidelijk afgebakende scope en meetbaar resultaat, zoals het genereren van een nieuwe klasse op basis van een bestaand patroon of het oplossen van een gedocumenteerde bug (zoals in sectie 1.4.6). Menselijk toezicht via pull request reviews en geautomatiseerde tests blijft nog altijd zeer belangrijk.

#### 1.4.2.3. Plan mode

Plan mode combineert de redeneerdiepte van ask mode met de uitvoeringsmogelijkheden van agent mode, maar plaatst een bewust controlepunt tussen beide. De agent analyseert de opdracht, verkent de relevante context en stelt een gedetailleerd stappenplan op dat aan de ontwikkelaar wordt voorgelegd. Pas na expliciete goedkeuring gaat de agent over tot uitvoering in de agent mode (Anthropic, 2026a).

Dit controlemechanisme biedt drie voordelen. Ten eerste vergroot het de veiligheid: onjuiste aannames worden gedetecteerd vooraleer ze resulteren in ongewenste wijzigingen. Ten tweede legt het plan de redenering van de agent vast, wat het bruikbaar maakt als basis voor documentatie

of code review. Ten derde maakt het de taakverdeling inzichtelijk wanneer de agent meerdere deeltaken parallel zou uitvoeren via sub-agents (zie sectie 1.4.4): het plan is het moment waarop die structuur zichtbaar en aanpasbaar is.

Plan mode vertoont een structurele gelijkenis met Chain-of-Thought prompting (zie sectie 1.1.2.3). In de praktijk impliceert dit een taakverdeling op modelniveau: de planningsfase vereist een sterk, redenerend model, terwijl de uitvoering van afzonderlijke stappen door een kleiner, sneller model kan worden afgehandeld.

Tabel 1.4 vat de drie beschreven modi samen en plaatst ze naast elkaar op basis van uitvoeringsrecht, goedkeuringsvereiste en typisch gebruik.

**Tabel 1.4**  
*Vergelijking van agent modi in Claude Code*

| Modus | Autonomie                     | Goedkeuring vereist?  | Typisch gebruik                     |
|-------|-------------------------------|-----------------------|-------------------------------------|
| Ask   | Lezen en analyseren           | Nooit                 | Vragen beantwoorden                 |
| Agent | Volledig uitvoeringsrecht     | Optioneel (per actie) | Taken van begin tot einde uitvoeren |
| Plan  | Eerst plannen, dan uitvoering | Ja, vóór uitvoering   | Complexe of risicovolle wijzigingen |

*Opmerking.* Overgenomen uit Claude Code [Software] door Anthropic (2026a). Copyright 2025, Anthropic PBC.

1.4.3. Reasoning modi

Naast de operationele modi die bepalen wanneer menselijke goedkeuring vereist is, bieden hedendaagse taalmodellen ook configureerbare denkcapaciteit: een instelling die bepaalt hoeveel rekentijd het model spendeert aan interne redenering vóór het een antwoord formuleert. Dit staat bekend als *extended thinking of reasoning mode* (Anthropic, z.d.b).

In de lage redenering modus genereert het model zijn antwoord direct vanuit de invoer. Bij hogere redenering doorloopt het model eerst een interne gedachtegang, vergelijkbaar met de werking van Chain-of-Thought prompting (zie sectie 1.1.2.3), maar dan volledig intern en zonder dat de tussenliggende stappen zichtbaar zijn in de prompt. Het model weegt alternatieven af, controleert de eigen redenering en past die indien nodig aan, alvorens een definitief antwoord te formuleren.

Er zijn vier niveaus beschikbaar: *low*, *medium*, *high* en *max*. Bij low en medium is de interne redenering beperkt, wat snelle en voorspelbare antwoorden oplevert. Voor softwareontwikkeling wordt echter aangeraden om standaard de high- of max-modus te gebruiken: de iets langere wachttijd weegt niet op tegen de kwaliteitswinst die hogere modi bieden, zeker bij taken die meerdere afhankelijke beslissingen vereisen, zoals architectuurkeuzes of het debuggen van complexe fouten (Anthropic, z.d.b).

Tabel 1.5 geeft een overzicht van de vier beschikbare reasoning niveaus en vergelijkt ze op het vlak van redeneerdiepte, snelheid en aanbevolen toepassingsgebied.

**Tabel 1.5***Overzicht van reasoning niveaus*

| Niveau | Redeneerdiepte | Snelheid  | Aanbevolen voor                           |
|--------|----------------|-----------|-------------------------------------------|
| Low    | Minimaal       | Zeer snel | Niet aangeraden voor softwareontwikkeling |
| Medium | Beperkt        | Snel      | Niet aangeraden voor softwareontwikkeling |
| High   | Diep           | Trager    | Standaard codetaken, uitleg               |
| Max    | Maximaal       | Traagst   | Complexe taken                            |

*Opmerking.* Overgenomen uit Claude Code [Software] door Anthropic (2026a). Copyright 2025, Anthropic PBC.

De keuze voor een reasoning modus is een afweging tussen kwaliteit en snelheid. Voor routinetaken zoals het genereren van een unit test op basis van een bestaande functie volstaat een lage instelling. Voor taken waarbij de agent autonome architectuurbeslissingen neemt, beveiligingsanalyses uitvoert of meerdere deeltaken coördineert via sub-agents, verhoogt een hogere modus de betrouwbaarheid van het resultaat aanzienlijk.

Binnen het Lexis-project werd de high-modus standaard gebruikt bij taken met een brede impact op de codebase, en de max-modus bij het ontwerpen van de OAuth2-authenticatieflow (zie hoofdstuk 2).

#### 1.4.4. Sub-agents

Complexe taken kunnen worden opgesplitst via sub-agents: gespecialiseerde agents die door een hoofdagent worden aangestuurd om een deeltaak zelfstandig uit te voeren. De hoofdagent fungeert als orchestrator<sup>9</sup>. Deze bepaalt de taakverdeling, verdeelt context en combineert de deelresultaten tot een samenhangend geheel. Elke sub-agent ontvangt een afgebakende opdracht en werkt daarbinnen autonoom, met toegang tot dezelfde tools als een gewone agent (Anthropic, 2026a).

Het voordeel van sub-agents is tweedelig. Enerzijds laat het parallelle uitvoering toe: deeltaken die onafhankelijk zijn, kunnen gelijktijdig worden verwerkt, wat de totale doorlooptijd aanzienlijk verkort. Anderzijds beperkt elke sub-agent zijn eigen context tot het relevante deel van de taak, wat de context van de hoofdagent minder snel vult en de kwaliteit van de individuele uitvoering verhoogt. Een sub-agent die uitsluitend de frontend analyseert, heeft geen last van de irrelevante context die een gelijktijdige backend-analyse zou introduceren.

In het praktijkscenario dat in sectie 0 wordt beschreven, werden twee sub-agents gelijktijdig ingezet: één voor de verkenning van de frontend, één voor de backend. De hoofdagent combineerde daarna hun bevindingen tot één geïntegreerd plan met twee deelfixes in één branch.

#### 1.4.5. Agent teams

Naast de hiërarchische structuur van hoofd- en sub-agents bestaat een meer horizontale organisatievorm: agent teams. In een agent team werken meerdere gespecialiseerde agents samen als gelijken, elk met een eigen rol in het proces. Een gangbare verdeling omvat een planneragent

---

<sup>9</sup> Een orchestrator is de sturende hoofdagent die taken delegeert aan sub-agents en hun resultaten bundelt tot één eindproduct.

die de taakverdeling coördineert, een uitvoeragent die de implementatie verzorgt en een reviewagent die de output toetst op correctheid en consistentie (Anthropic, 2026c).

De kracht van agent teams ligt in de specialisatie: elke agent is geconfigureerd voor één specifieke taak, met een geoptimaliseerde system prompt en een beperkte, taakgerichte toolset. Tegelijkertijd vereist deze aanpak een goed doordacht communicatieprotocol: agents moeten tussenresultaten kunnen uitwisselen en voortgang kunnen rapporteren aan de coördinerende laag of de menselijke operator.

In productieomgevingen worden agent teams ingezet wanneer de complexiteit of de vereiste specialisatie de capaciteit van één agent overstijgt: grootschalige code refactoring, het gelijktijdig opstellen van technische documentatie en het uitvoeren van regressietests, of het parallel verwerken van meerdere onafhankelijke features binnen één sprint.

Agent teams zijn in de context van Claude Code een bijzonder recente ontwikkeling: de feature werd geïntroduceerd op 5 februari 2026 met de release van Claude Opus 4.6 en bevindt zich momenteel nog in research preview (Anthropic, 2026).

Tabel 1.6 plaatst sub-agents en agent teams naast elkaar en vergelijkt beide architecturen op structuur, coördinatiemechanisme, mogelijkheid tot parallelle uitvoering en mate van specialisatie.

**Tabel 1.6**  
*Vergelijking tussen sub-agents en agent teams*

|                      | Sub-agents                            | Agent teams                             |
|----------------------|---------------------------------------|-----------------------------------------|
| Structuur            | Hiërarchisch (hoofd- en sub-agents)   | Horizontaal (gelijkwaardige agents)     |
| Coördinatie          | Hoofdagent stuurt aan                 | Agents werken onafhankelijk samen       |
| Parallele uitvoering | Ja, voor onafhankelijke deeltaken     | Ja, elke agent zijn eigen specialisatie |
| Specialisatie        | Beperkt tot de taak van de hoofdagent | Elke agent apart geconfigureerd         |

*Opmerking.* Overgenomen uit Claude Code [Software] door Anthropic (2026a). Copyright 2025, Anthropic PBC.

### 1.4.6. Praktijk scenario: oplossen van een in Jira beschreven bug

Om de besproken concepten concreet te maken, wordt hieronder een praktijkscenario beschreven dat tijdens de bachelorproefperiode plaatsvond binnen het Lexis-project.

#### 1.4.6.1. Aanleiding

CERM heeft een Aikido-beveiligingsscaner (Aikido Security BV, 2026). Deze detecteerde twee XSS-kwetsbaarheden (Cross-Site Scripting) in de Lexis-codebase. Deze werden geregistreerd als Jira-issues LEXIS-100 en LEXIS-101. LEXIS-100 bevatte een React-component die ruwe e-mail HTML rechtstreeks doorgaf aan dangerouslySetInnerHTML attribuut zonder *sanitation*<sup>10</sup>. LEXIS-101 betrof een ASP.NET Core-controller die het inhoudstype van e-mailbijlagen ongefilterd doorgaf aan de bestandsresponse, waardoor een browser de bijlage als HTML kon renderen of zelf HTTP-headerinjectie mogelijk werd.

<sup>10</sup> Sanitation is het verwijderen van kwaadaardige code (zoals scripts) uit onbetrouwbare HTML om XSS-aanvallen te voorkomen.

### 1.4.6.2. De prompt

Voor het implementeren van een fix werd Claude Code gebruikt, een command-line AI-agent van Anthropic. De instructie werd bewust eenvoudig gehouden:

**Figuur 1.11**

*Prompt van het praktijkscenario bij AI-agents*

```
Los de XSS-kwetsbaarheden op die beschreven staan in deze Jira-issues:  
https://cerm.atlassian.net/browse/LEXIS-100  
https://cerm.atlassian.net/browse/LEXIS-101  
  
Maak daarna een pull request aan op Azure DevOps.
```

*Opmerking.* Eigen werk.

De simpliciteit van de prompt is een bewuste keuze. Alle technische context bevindt zich in de Jira-issues; de agent haalt die zelf op via de Atlassian MCP-server (zie sectie 1.2.7.1). Dit illustreert een belangrijk principe van *agentic*<sup>11</sup> werken: de prompt formuleert het doel, niet de aanpak.

### 1.4.6.3. Het plan

Vóór de uitvoering werd plan mode geactiveerd, zoals beschreven in sectie 1.4.2.3. Dit om drie redenen. Ten eerste biedt plan mode veiligheid: wijzigingen aan code in bijvoorbeeld configuratiebestanden vereisen een review vooraleer ze worden doorgevoerd. Ten tweede legt het gegenereerde plan de redenering van de agent vast en vormt het zo een directe bron voor documentatie. Ten derde bood het plan inzicht in hoe de agent de twee gerelateerde issues zou groeperen, vóór de uitvoering.

De agent haalde beide Jira-issues parallel op via de Atlassian MCP-server en startte gelijktijdig twee sub-agents (zie sectie 1.4.4): één voor de verkenning van de frontend, één voor de backend. Op basis van die verkenning stelde de agent een plan op met twee deelfixes in één branch en één pull request.

Voor LEXIS-100 werd voorgesteld de bibliotheek DOMPurify te installeren en `message.bodyHtml` te sanitiseren vóór doorgave aan `dangerouslySetInnerHTML`.

Voor LEXIS-101 werd een private methode `SanitizeContentType` voorgesteld en deze dan te implementeren, die null-waarden, controlekarakters, HTML types en ongeldige MIME-structuren afvangt en vervangt door `application/octet-stream`.

### 1.4.6.4. Uitvoering

Na goedkeuring van het plan voerde de agent de implementatie volledig autonoom uit. Daarbij deed zich een illustratief moment voor: een bash-commando met een Windows-pad faalde, waarna de agent zonder menselijke tussenkomst overschakelde naar PowerShell en verder ging. Dit zelfcorrigerend gedrag is kenmerkend voor een agent.

Na de implementatie verifieerde de agent beide builds (`dotnet build` en `npm run build`) vóór de commit. Beide slaagden zonder fouten. Vervolgens werden de wijzigingen gecommited, de branch gepusht en een pull request aangemaakt via de Azure DevOps MCP-server (zie sectie 1.2.7.2).

---

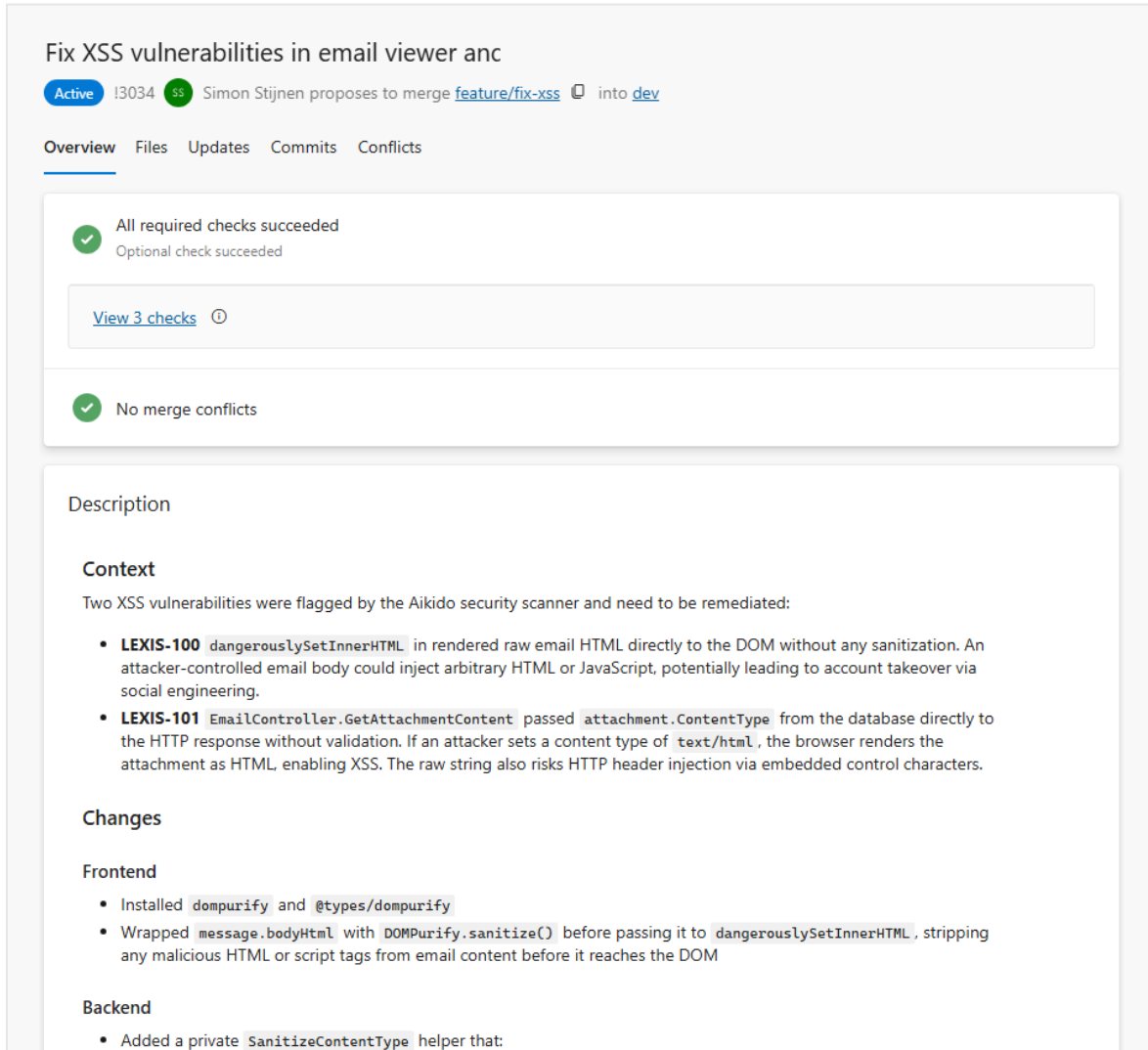
<sup>11</sup> Agentic is de eigenschap van een AI-systeem om autonoom te redeneren, zelfstandig beslissingen te nemen en acties uit te voeren om een specifiek doel te bereiken, zonder constante menselijke sturing.

#### 1.4.6.5. Resultaat

De volledige workflow verliep in 4 minuten en 23 seconden, volledig autonoom. Er werden 4 bestanden gewijzigd met 53 toevoegingen en 2 verwijderingen. Een branch voor deze fix werd aangemaakt. Pull request #3034 werd aangemaakt zonder verdere menselijke tussenkomst na de plangoedkeuring.

**Figuur 1.12**

*Resultaat van praktijkscenario AI-agents*



*Opmerking.* Schermafbeelding van de gegenereerde pull request in Azure DevOps.

De resulterende pull request is weergegeven in Figuur 1.12. De agent voorzag de pull request automatisch van een naam, een target branch en een beschrijving met daarin een toelichting van het probleem en de gekozen oplossing. Menselijke controle blijft echter noodzakelijk vooraleer wijzigingen effectief naar productie worden gebracht. De risico's die hieraan verbonden zijn, worden verder besproken in sectie 1.6.

#### 1.4.6.6. Reflectie

Dit scenario toont aan dat een AI-agent kwalitatief verschilt van een taalmodel dat code suggereert. De agent begreep de beveiligingscontext via Jira, analyseerde de codebase via sub-agents, stelde een technisch plan op, voerde het uit, corrigeerde een fout onderweg en sloot de ontwikkelcyclus

af met een pull request. De prompt bevatte geen technische instructies; die redenering kwam van de agent zelf, zonder expliciete technische sturing in de prompt zelf.

Een belangrijk voorbehoud blijft de verificatie van de correctheid van de gegenereerde code. De agent controleerde of de builds slaagden, maar functionele en beveiligingsgerichte tests bleven de verantwoordelijkheid van de ontwikkelaar.

*Agentic* werken verschuift menselijke input naar de plaatsen waar die het meest waarde toevoegt: probleemstelling, planreview en eindvalidatie. In dit geval bevestigde die validatie dat de gegenereerde oplossing correct was: de pull request kon zonder verdere wijzigingen worden gemerged.

## 1.5. Voordelen van AI-ondersteunde ontwikkeling

Naast de theoretische concepten die in de voorgaande secties aan bod kwamen, namelijk prompt engineering (1.1), model context protocol servers (1.2), skills (1.3) en agents (1.4), stelt zich de vraag welke concrete voordelen AI-ondersteunde ontwikkeling biedt in een professionele context. In deze sectie worden vier centrale voordelen besproken, telkens geïllustreerd aan de hand van het Lexis-project: snellere ontwikkeling en prototyping, de automatisering van repetitieve taken, ondersteuning bij testen en debuggen, en een verbetering van codekwaliteit. Elk van deze voordelen draagt bij aan een efficiëntere ontwikkelcyclus, al gaan ze niet zonder risico's gepaard. Die risico's komen aan bod in sectie 1.6.

### 1.5.1. Snellere ontwikkeling en prototyping

Een van de meest tastbare voordelen van AI-ondersteunde ontwikkeling is de aanzienlijke verkorting van de tijd tussen een idee en een werkend prototype. Waar een traditioneel software-project maanden aan opzet, architectuurkeuzes en boilerplate-implementatie vereist, kunnen AI-tools grote delen van deze initiële last overnemen. Peng et al. (2023) toonden in een gecontroleerd experiment aan dat ontwikkelaars die AI-assistentie gebruikten taken gemiddeld 55,8% sneller voltooiden dan een controlegroep zonder AI-ondersteuning.

Dit gegeven vertaalt zich direct naar de praktijk van het Lexis-project. Binnen een relatief korte periode, minder dan twee maanden, werd een volledig functioneel MVP (Minimum Viable Product) opgeleverd die een werkende e-mailverwerking via een Large Language Model (LLM), een koppeling met de CERM REST API en een eerste frontend-interface omvatte. Dit was mogelijk dankzij de gecombineerde inzet van AI-tools: Lovable (2026) genereerde de initiële gebruikers-interface op basis van een tekstuele beschrijving, terwijl Cursor (2026) instond voor de codegeneratie in de backend. De ontwikkelaar kon zich zo concentreren op de architecturale en domeinspecifieke beslissingen, terwijl de AI de uitwerking ervan in code versnelde.

Het voordeel van snellere prototyping reikt verder dan alleen tijdswinst. Een vlotter iteratieproces laat toe om vroeg in het traject feedback te verzamelen, foute aannames tijdig bij te sturen en de uiteindelijke oplossing beter af te stemmen op de behoeften van de gebruiker. AI fungeert in dit opzicht als een verlengstuk van de ontwikkelaar (Peng et al., 2023).

### 1.5.2. Automatisering van repetitieve taken

Softwareontwikkeling bestaat voor een aanzienlijk deel uit taken die weinig creatieve inspanning vereisen, maar wel tijdrovend zijn. Voorbeelden hiervan zijn het schrijven van CRUD-operaties

(Create, Read, Update, Delete), het opstellen van data transfer objects (DTO's), het documenteren van methoden en het genereren van migratiebestanden. AI-tools zijn bijzonder sterk in het automatiseren van precies dit soort herhaalwerk (Peng et al., 2023).

Binnen het Lexis-project werd dit op meerdere vlakken benut. Nieuwe API-endpoints volgden steeds een vast patroon, bestaande uit een controller en service laag, dat de AI op basis van een eerder geschreven voorbeeld kon reproduceren voor een nieuw domein. Eenzelfde principe gold voor unit tests bij standaard scenario's: de AI genereerde een eerste versie op basis van de bestaande code, waarna de ontwikkelaar enkel nog de domeinspecifieke randgevallen hoefde toe te voegen.

Ook buiten de eigenlijke code bleek automatisering waardevol. Commit messages, documentatie in Jira, pull requests in Azure behoren tot de administratieve last die met AI-ondersteuning aanzienlijk vlotter verloopt. De kwaliteit van deze output is doorgaans voldoende voor een eerste versie en beperkt de cognitieve belasting van de ontwikkelaar.

### 1.5.3. Ondersteuning bij testen en debuggen

Het opsporen en oplossen van fouten is een tijdsintensief onderdeel van elk softwareproject. AI-tools bieden hier ondersteuning op twee manieren: door actief te helpen bij het debuggen van bestaande fouten en door het genereren van testcases die de ontwikkelaar zelf mogelijk over het hoofd zou zien.

Bij debuggen werkt AI het effectiefst wanneer het zijn redenering stap voor stap toelicht. Deze aanpak, bekend als Chain-of-Thought (CoT) prompting en eerder besproken in de sectie over prompt engineering (zie sectie 1.1.2.3), laat de AI toe om een fout systematisch te analyseren: de stacktrace en error interpreteren, de oorzaak lokaliseren en een oplossing voorstellen met bijhorende uitleg (Wei et al., 2023). Dit maakt het debug proces niet alleen sneller, maar ook transparanter: de ontwikkelaar volgt de redenering en leert tegelijk van het proces.

### 1.5.4. Verbetering van codekwaliteit

Naast snelheid en automatisering draagt AI-ondersteunde ontwikkeling ook bij aan de kwaliteit van de geproduceerde code. Een AI-assistent fungeert als een permanente reviewpartner die naamgeving, structuur, herbruikbaarheid en naleving van goede ontwerpprincipes doorlopend evalueert (Peng et al., 2023). Waar een traditionele codereview een tijd kan duren, is AI-feedback onmiddellijk beschikbaar op het moment van schrijven.

In de praktijk vertaalt dit zich in concrete suggesties: methoden die te lang worden, worden opgesplitst, variabelen worden hernoemd naar meer beschrijvende namen en geduplicateerde code wordt samengebracht in een gedeelde hulpfunctie. Over een grotere codebase zorgt dit voor een consistentere stijl en een lagere onderhoudslast. Bovendien helpt AI bij het detecteren van antipatronen die anders onopgemerkt zouden blijven, zoals overmatige koppeling tussen klassen of te diepe nesting van condities.

Het is evenwel belangrijk om dit voordeel genuanceerd te bekijken. AI-gegenereerde code is niet per definitie correcte of veilige code. Een onkritische feedbackloop, waarbij de ontwikkelaar AI-suggesties klakkeloos overneemt zonder ze te verifiëren, kan de kwaliteit net verslechteren. Hallucinaties, subtiele logische fouten en beveiligingskwetsbaarheden zijn risico's die in detail besproken worden in de volgende secties over kwaliteitsrisico's (sectie 1.6.2) en beveiligingsrisico's (sectie 1.6.3).

## 1.6. Gevaren van AI-ondersteunde ontwikkeling

De vorige sectie besprak de voordelen van AI-ondersteunde ontwikkeling, maar deze voordelen gaan gepaard met risico's. Een onkritisch gebruik van AI-tools kan leiden tot afhankelijkheid, kwaliteitsverlies, beveiligingsproblemen en ethische vragen.

### 1.6.1. Te grote afhankelijkheid van AI

Eén van de meest onderschatte gevaren van AI-ondersteunde ontwikkeling is de geleidelijke opbouw van afhankelijkheid. Wanneer een ontwikkelaar systematisch AI inzet voor taken zoals debuggen, codegeneratie en het begrijpen van onbekende bibliotheken, neemt de nood aan eigen redenering af. Anthropic-onderzoek toont aan dat ontwikkelaars die AI inzetten bij het aanleren van nieuwe codebibliotheken gemiddeld 17% lager scoren op begripstoetsen dan ontwikkelaars die zonder AI werkten. De AI-groep haalde een gemiddelde van 50% op quizzen tegenover 67% voor de manuele groep, met het grootste verschil bij vragen over foutopsporing (Shen & Tamkin, 2026).

Dit risico is bijzonder acuut bij junior-ontwikkelaars, die vroeg in hun carrière terugvallen op AI voor fundamentele taken en zo het risico lopen deze vaardigheden nooit zelfstandig te ontwikkelen. Onderzoek van Becker et al. (2025) nuanceert bovendien de veronderstelde productiviteitswinst: ervaren ontwikkelaars met AI-assistentie deden gemiddeld 19% langer over taken dan zonder, terwijl zij zelf een tijdswinst van 24% hadden verwacht.

### 1.6.2. Kwaliteitsrisico's en hallucinaties

AI-modellen genereren niet altijd correcte en optimale code, soms introduceren ze ook een specifiek risico dat bekend staat als packagehallucinaties. Hierbij verwijst een model naar softwarepakketten die niet bestaan in publieke repositories. Krishna et al. (2025) analyseerden het gedrag van 16 codemodellen en stelden een gemiddeld hallucinatiepercentage van 19,6% vast. Aanvallers kunnen de gehallucineerde pakketnamen registreren en er kwaadaardige code in stoppen, waarna ontwikkelaars die de AI-suggestie klakkeloos overnemen deze malware onbewust importeren in hun project. Dit aanvalstype staat bekend als slopsquatting.

Een tweede kwaliteitsrisico schuilt in de feedbackloop bij iteratief verfijnen van AI-gegenereerde code. Een gecontroleerd experiment waarbij 400 codesamples over 40 rondes werden "verbeterd" via vier verschillende promptstrategieën, toonde een toename van 37,6% in kritieke kwetsbaarheden na slechts vijf iteraties (Shukla et al., 2025). De aanname dat herhaaldelijk verfijnen de kwaliteit verhoogt, wordt hier dus tegengesproken. Elke iteratie zonder menselijke tussenkomst en verificatie kan de codebase aantasten in plaats van verbeteren.

### 1.6.3. Beveiligingsrisico's en kwetsbaarheden

Een van de minder zichtbare gevaren bij het gebruik van AI-tools voor softwareontwikkeling is het risico op beveiligingsproblemen in de gegenereerde code. AI-tools zoals GitHub Copilot zijn getraind op grote hoeveelheden publieke broncode van het internet. Omdat een deel van die broncode zelf onveilig was, kunnen AI-tools dezelfde fouten herhalen in nieuw gegenereerde code (Fu et al., 2025).

Uit onderzoek bleek dat bijna 30% van de door AI gegenereerde Python-codefragmenten beveiligingsproblemen bevatten; voor JavaScript lag dit op ongeveer 24% (Fu et al., 2025). De gevonden problemen vielen onder 43 verschillende categorieën van de Common Weakness

Enumeration (CWE), een internationale standaard voor het classificeren van softwarefouten. Acht van die categorieën stonden op de CWE Top-25, een lijst met de gevaarlijkste en meest voorkomende kwetsbaarheden.

Wat dit gevaar extra groot maakt, is dat ontwikkelaars de gegenereerde code vaak te snel als correct beschouwen. Wanneer de code op het eerste gezicht werkt, is de verleiding groot om ze te accepteren zonder grondige beveiligingscontrole. Bekende kwetsbaarheden zoals SQL-injectie of Cross-Site Scripting (XSS) kunnen zo ongemerkt in een codebase terechtkomen. Bij een project als Lexis, dat gevoelige bedrijfsdata verwerkt, kan dit ernstige gevolgen hebben. Het is dan ook aangewezen dat AI-gegenereerde code altijd wordt onderworpen aan een kritische beoordeling, bij voorkeur met behulp van gespecialiseerde beveiligingstools.

#### 1.6.4. Verlies van domein- en codekennis

Een risico dat minder direct zichtbaar is maar op lange termijn groot kan zijn, is het verlies van technische kennis bij ontwikkelaars die te sterk leunen op AI. Wanneer AI het grootste deel van de code schrijft, bestaat het gevaar dat een ontwikkelaar de onderliggende logica niet meer volledig begrijpt. Dit maakt het moeilijker om fouten op te sporen of om goede technische beslissingen te nemen.

Onderzoek van Shen & Tamkin (2026) toont aan dat ontwikkelaars die AI inzetten bij het aanleren van nieuwe codebibliotheken gemiddeld 17% lager scoren op begripstoetsen dan collega's die de code zelfstandig schreven, een bevinding die aansluit bij de afhankelijkheid besproken in sectie 1.6.1. Aanvullend onderzoek bij studenten bevestigt dit patroon: wie AI gebruikte voor conceptuele vragen scoorde gemiddeld 65% of hoger, terwijl studenten die codegeneratie volledig aan AI overlieten onder de 40% bleven.

Het probleem ligt dus niet bij het gebruik van AI zelf, maar bij de manier waarop het wordt ingezet. Wie AI gebruikt om te begrijpen om te vragen hoe iets werkt en de gegenereerde code te analyseren, zal zijn kennis versterken. Wie AI enkel gebruikt als codegenerator, waarbij de output zonder analyse wordt overgenomen, riskeert zijn eigen vaardigheden te laten afnemen. Op lange termijn leidt dit tot een situatie waarbij de ontwikkelaar onvoldoende in staat is om de output van de AI te beoordelen en te corrigeren, wat de betrouwbaarheid van het eindproduct in gevaar brengt.

#### 1.6.5. Privacy en Data Compliance

Bij het gebruik van AI-tools voor softwareontwikkeling is er een privacyprobleem dat vaak over het hoofd wordt gezien: wanneer een ontwikkelaar een tool zoals GitHub Copilot of Cursor gebruikt, wordt de geschreven code samen met de context naar externe servers gestuurd voor verwerking. Dit betekent dat bedrijfseigen code het bedrijf verlaat en op servers van derden terechtkomt.

Vanuit het oogpunt van de General Data Protection Regulation (GDPR), is dit een aandachtspunt. GDPR vereist dat organisaties controle hebben over welke data wordt verwerkt en door wie. Sommige AI-tools gebruiken de ingevoerde code standaard om hun modellen verder te trainen, tenzij de gebruiker actief kiest om dit uit te schakelen. Zo kondigde GitHub in 2026 aan dat interactiedata van Copilot Free, Pro en Pro+-gebruikers voortaan standaard wordt ingezet voor modeltraining (Rodriguez, 2026). Dit geldt niet voor Business- en Enterprise-abonnementen, maar toont aan hoe snel privacyinstellingen kunnen veranderen en hoe belangrijk het is om die actief op te volgen.

Naast het gebruik van AI-tools tijdens de ontwikkeling stelt ook het Lexis-project zelf privacyvragen. Lexis verwerkt inkomende e-mails van klanten van CERM, waarbij die mails bedrijfsgevoelige informatie kunnen bevatten zoals klantnamen, bestellingsdetails en leveringsgegevens. Het is dan ook essentieel dat zowel de gebruikte AI-tools als het eindproduct voldoen aan de geldende privacyregelgeving.

### 1.6.6. Ethische overwegingen

Het gebruik van AI bij softwareontwikkeling roept ook ethische vragen op die nog geen eenvoudig antwoord hebben. Twee belangrijke onderwerpen hierbij zijn het auteursrecht op AI-gegenereerde code en de aansprakelijkheid wanneer die code problemen veroorzaakt.

Wat betreft auteursrecht heeft de Europese Unie stappen gezet via de AI-verordening (ook wel de EU AI Act), die op 1 augustus 2024 in werking trad. De verordening verplicht aanbieders van generatieve AI-systemen om transparant te zijn over welk auteursrechtelijk beschermd materiaal werd gebruikt voor de training van hun modellen (Europees Parlement, 2023). Daarnaast bepaalde het Europees Parlement dat inhoud die volledig door AI is gegenereerd niet automatisch door het auteursrecht beschermd mag worden, en dat de creatieve sector recht heeft op een eerlijke vergoeding wanneer haar werken worden gebruikt als trainingsdata (Europees Parlement, 2026).

Een tweede ethische vraag gaat over verantwoordelijkheid. De AI-verordening introduceert een risicoclassificatie voor AI-systemen en legt verplichtingen op aan zowel aanbieders als gebruikers. De AI-verordening introduceert een risicoclassificatie voor AI-systemen en legt verplichtingen op aan zowel aanbieders als gebruikers (Europese Commissie, 2024). Een eenduidig wettelijk kader over hoe aansprakelijkheid precies wordt verdeeld bij AI-gerelateerde schade is echter nog in ontwikkeling.

Voor projecten zoals Lexis, waarbij de applicatie automatisch bestellingen plaatst op basis van AI-interpretatie van e-mails, is dit bijzonder relevant. Als de AI een e-mail verkeerd interpreteert en daardoor een foutieve bestelling plaatst, wie is dan aansprakelijk? Dit soort vragen dwingt organisaties na te denken over wie de output van de AI valideert, en welke processen zijn er om fouten tijdig op te vangen?

Dit hoofdstuk behandelde vier manieren waarop AI het softwareontwikkelp proces ondersteunt: prompt engineering, MCP-servers, skills en agents. Binnen het Lexis-project werden al deze concepten in de praktijk toegepast. AI-ondersteuning bleek niet alleen een manier om sneller te werken, maar ook een andere manier van werken: de ontwikkelaar focust op de probleemstelling en de validatie, terwijl de uitvoering steeds meer aan de AI wordt overgelaten. Tegelijk toonden de gevaren aan dat een kritische houding noodzakelijk blijft. Hoofdstuk 2 bespreekt de implementatie van OAuth2 binnen Lexis, een concreet voorbeeld waarbij AI een rol speelde in zowel het ontwerp als de uitvoering.

## 2. OAuth2

Veilige toegang tot gevoelige bedrijfsgegevens is een fundamentele vereiste voor elke cloud-applicatie die in een professionele omgeving wordt ingezet. Binnen het Lexis-project verwerkt de applicatie inkomende bestelmails namens meerdere klanten, zogenaamde tenants, en plaatst zij bestellingen via de CERM REST API. Dit vereist een robuust autorisatiemechanisme dat garandeert dat enkel geauthenticeerde gebruikers en geautoriseerde services toegang krijgen tot de juiste gegevens.

In dit hoofdstuk wordt OAuth 2.0 (OAuth2) besproken als het autorisatieprotocol dat binnen Lexis werd geïmplementeerd. Eerst wordt het protocol zelf geïntroduceerd: wat het inhoudt, welke centrale rollen het definieert en waarom het de logische keuze was voor dit project (2.1). Vervolgens worden de kernconcepten en terminologie toegelicht die noodzakelijk zijn om de werking van het protocol correct te begrijpen (2.2). Daarna komen de twee OAuth2-flows aan bod die binnen Lexis worden toegepast: de Authorization Code flow met PKCE voor interactieve gebruikersauthenticatie en de Client Credentials flow voor machine-to-machine-communicatie (0). Het hoofdstuk sluit af met de concrete implementatie van OAuth2 in de multi-tenantarchitectuur van het project (0).

### 2.1. Wat is OAuth2

OAuth2 is een open standaard voor autorisatie die het mogelijk maakt om applicaties beperkte toegang te verlenen tot beschermde bronnen van een gebruiker, zonder dat die gebruiker zijn wachtwoord hoeft te delen met de applicatie. Het protocol werd in 2012 gepubliceerd door de Internet Engineering Task Force (IETF) en is inmiddels uitgegroeid tot de standaard voor autorisatie op het web (Hardt, 2012).

Het OAuth2-framework werkt via een tijdelijk toegangstoken dat de client ontvangt van een autorisatieserver, met specifieke scopes die de toegang beperken. Daarvoor definieert het framework vier centrale rollen.

Het OAuth2-framework definieert vier centrale rollen. De resource owner is de eindgebruiker die eigenaar is van de beschermde gegevens (Hardt, 2012, p. 6). De client is de applicatie die namens de gebruiker toegang vraagt tot die gegevens. De authorization server verifieert de identiteit van de gebruiker en geeft tokens uit. De resource server, ten slotte, is de server die de beschermde gegevens beheert en alleen toegang verleent aan requests met een geldig token.

Een belangrijk onderscheid moet gemaakt worden tussen authenticatie en autorisatie (Hardt, 2012). Authenticatie beantwoordt de vraag “wie ben je?”, terwijl autorisatie de vraag beantwoordt “wat mag je doen?”. OAuth2 is in de eerste plaats een autorisatieprotocol: het verleent een applicatie toestemming om namens een gebruiker bepaalde acties uit te voeren, zonder noodzakelijk de identiteit van die gebruiker te verifiëren. Binnen Lexis volstaat dit, omdat de applicatie uitsluitend geautoriseerde toegang tot de CERM REST API vereist en geen identiteitslaag bovenop het protocol nodig heeft.

Voor Lexis was OAuth2 de logische keuze: de applicatie verwerkt gevoelige bedrijfsdata via de CERM REST API in een multi-tenantarchitectuur. OAuth2 biedt de nodige isolatie per tenant, vermijdt dat wachtwoorden rechtstreeks worden opgeslagen, en sluit aan bij de bestaande OAuth2-autorisatieserver van CERM.

## 2.2. Terminologie en concepten

Om het OAuth2-protocol correct toe te passen, is een grondige kennis van de bijhorende terminologie onmisbaar. De rollen die in sectie 2.1 werden geïntroduceerd communiceren met elkaar via een aantal specifieke mechanismen. In wat volgt, worden de twee meest fundamentele bouwstenen toegelicht: de tokens waarmee autorisatie wordt afgehandeld (2.2.1) en de clienttypes die bepalen hoe een applicatie zich identificeert bij de authorization server (0).

### 2.2.1. Tokens in OAuth2

Tokens vormen het kernmechanisme waarmee OAuth2 autorisatie afhandelt. Nadat een gebruiker toestemming heeft verleend, geeft de authorization server tokens uit aan de client. Er bestaan meerdere typen tokens, elk met een eigen doel en levensduur (Hardt, 2012).

Het access token is het centrale token in OAuth2. Het is een tekenreeks die de client meestuurt bij elke request naar de resource server. Het access token vertegenwoordigt de toegang die werd verleend: welke bronnen benaderd mogen worden (scope) en voor hoe lang (levensduur). Een access token werkt als een Bearer token: iedereen die het token bezit, kan het gebruiken voor API-requests zonder verdere verificatie van de houder. Een access token heeft daarom meestal een korte levensduur, typisch van enkele minuten tot maximaal een uur. Dit is een bewuste beveiligingskeuze: mocht een token worden onderschept, dan is de schade beperkt tot de resterende geldigheid (Hardt, 2012). Het token wordt als HTTP-header meegestuurd bij elk API-request in de vorm:

**Figuur 2.1**

Voorbeeld van de “Authorization” header

```
Authorization: Bearer <token>
```

*Opmerking.* Eigen werk, gebaseerd op *The OAuth 2.0 Authorization Framework*, door Hardt (2012). Internet Engineering Task Force. <https://datatracker.ietf.org/doc/html/rfc6749/>. Copyright 2012, IETF Trust.

Omdat het access token bewust een korte levensduur heeft, biedt het refresh token een oplossing voor de gebruikerservaring. Een refresh token is een langlevend token dat de client kan gebruiken om bij de authorization server een nieuw access token op te vragen, zonder dat de gebruiker opnieuw hoeft in te loggen. Het refresh token wordt uitsluitend uitgewisseld tussen de client en de authorization server en wordt nooit naar de resource server gestuurd. Dit beperkt de blootstelling: zelfs als de communicatie met de resource server zou worden onderschept, blijft het refresh token buiten bereik (Hardt, 2012). De levensduur van een refresh token varieert van enkele dagen tot weken. Sommige implementaties hanteren refresh token rotation: elke keer dat een refresh token wordt gebruikt, wordt het ongeldig verklaard en wordt een nieuwe terug gegeven. Dit zorgt ervoor dat een gestolen refresh token slechts eenmalig bruikbaar is (Lodderstedt et al., 2025, p. 58).

Tabel 2.1 geeft een samenvattend overzicht van de voornaamste verschillen tussen het access token en het refresh token op vlak van doel, levensduur, gebruik en blootstelling aan de client.

**Tabel 2.1**  
*Vergelijking tussen de access token en refresh token*

|                         | Access token                      | Refresh token                |
|-------------------------|-----------------------------------|------------------------------|
| Doel                    | Toegang verlenen tot een resource | Nieuw access token ophalen   |
| Levensduur              | Kort (minuten tot uren)           | Lang (dagen tot weken)       |
| Meegestuurd bij         | Elke API-request                  | Alleen bij token refresh     |
| Blootgesteld aan client | Ja                                | Nee (enkel via BFF in Lexis) |

*Opmerking.* Eigen werk, gebaseerd op *The OAuth 2.0 Authorization Framework*, door Hardt (2012). Internet Engineering Task Force. <https://datatracker.ietf.org/doc/html/rfc6749/>. Copyright 2012, IETF Trust.

2.2.2. Clienttypes en het client secret

Het OAuth2-framework maakt een fundamenteel onderscheid tussen twee typen clients op basis van hun vermogen om geheimen veilig te bewaren. Een confidential client is een applicatie die in staat is om een client secret vertrouwelijk op te slaan. Dit zijn typisch server-side applicaties die draaien in een beveiligde omgeving die niet toegankelijk is voor eindgebruikers. Een public client daarentegen kan geen geheimen veilig bewaren. Single Page Applications (SPA's) die volledig in de browser draaien en native mobiele applicaties zijn de meest voorkomende voorbeelden: hun code is zichtbaar via de developer tools in de browser of decompileerbaar, waardoor elk ingebakken secret publiek wordt (Hardt, 2012).

Een client secret is een geheim wachtwoord dat de authorization server toekent aan een geregistreerde client. Bij een tokenrequest stuurt de client naast de autorisatiecode ook zijn client ID en client secret mee. De authorization server verifieert zo dat het request afkomstig is van de legitieme client en niet van een partij die enkel de autorisatiecode heeft onderschept. Het client secret biedt dus alleen bescherming als het daadwerkelijk geheim blijft, wat uitsluitend het geval is bij confidential clients. Dit is precies de reden waarom PKCE werd ontwikkeld als aanvullend mechanisme, besproken in sectie 2.3.1 (Hardt, 2012; Lodderstedt et al., 2025).

Tabel 2.2 vat de belangrijkste kenmerken van confidential en public clients samen en geeft aan welke OAuth2-flow voor elk type aanbevolen wordt, inclusief de toepassing ervan binnen Lexis.

**Tabel 2.2**  
*Vergelijking tussen confidential en public clients*

|                    | Confidential client                  | Public client                         |
|--------------------|--------------------------------------|---------------------------------------|
| Kan secret bewaren | Ja                                   | Nee                                   |
| Typisch voorbeeld  | Server-side webapplicatie            | SPA, mobiele app                      |
| Aanbevolen flow    | Authorization Code (+PKCE optioneel) | Authorization Code + PKCE (verplicht) |
| Gebruik in Lexis   | ASP.NET backend (BFF)                | Niet van toepassing                   |

*Opmerking.* Eigen werk, gebaseerd op *The OAuth 2.0 Authorization Framework*, door Hardt (2012). Internet Engineering Task Force. <https://datatracker.ietf.org/doc/html/rfc6749/>. Copyright 2012, IETF Trust.

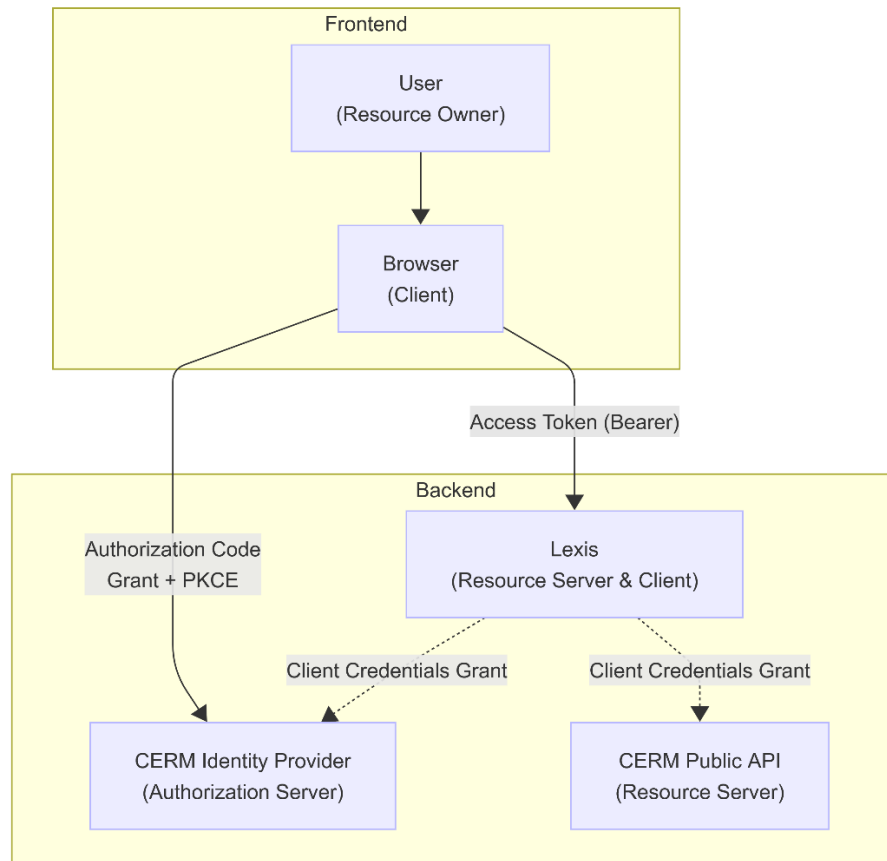
2.3. Varianten van OAuth2

Het OAuth2-framework definieert meerdere *grant types*, die in de praktijk overeenkomen met verschillende autorisatieflows. Elke grant type is geschikt voor een ander type applicatie of

gebruikssituatie. Hieronder worden de twee flows besproken die relevant zijn voor het Lexis-project: de Authorization Code flow met Proof Key for Code Exchange (PKCE) en de Client Credentials flow.

**Figuur 2.2**

*Grant types in Lexis Architecture*



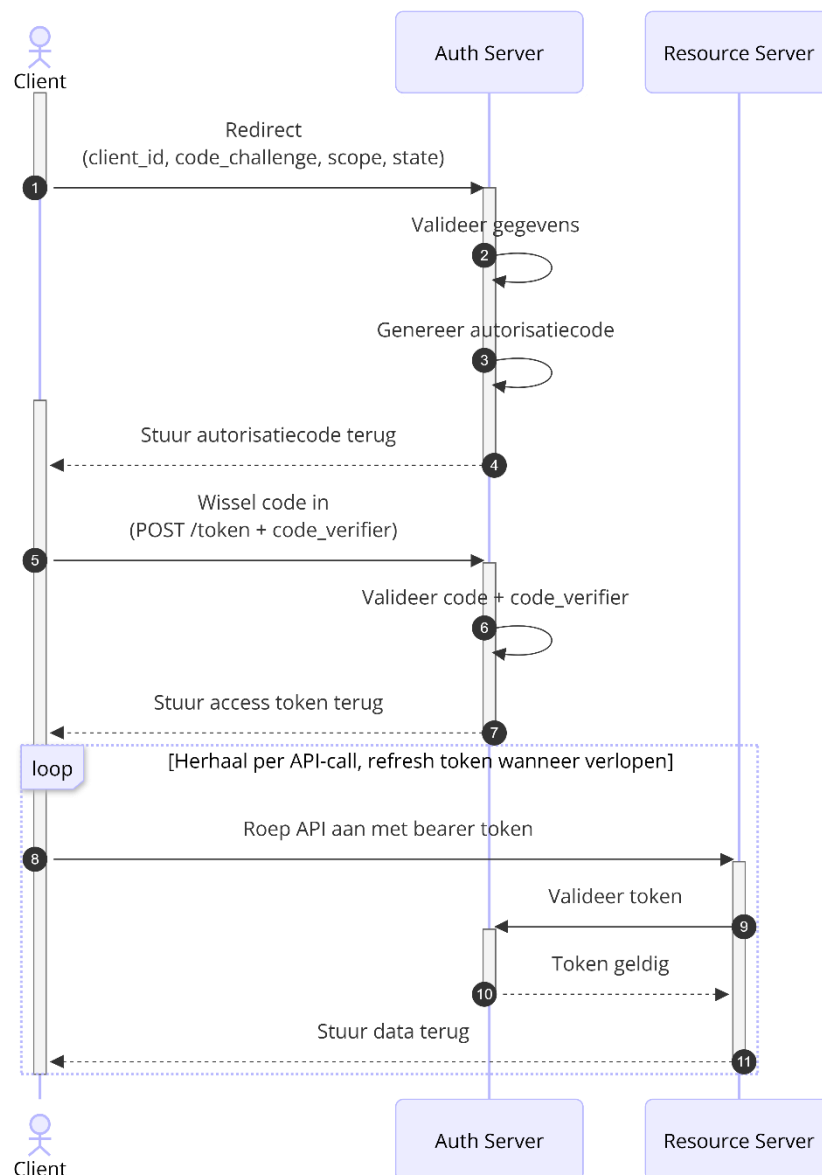
*Opmerking.* Eigen werk.

### 2.3.1. Authorization Code flow met PKCE

De Authorization Code flow is de meest gebruikte en aanbevolen OAuth2-flow voor webapplicaties waarbij een gebruiker interactief inlogt (Hardt, 2012). Het proces verloopt in verschillende stappen (Figuur 2.3):

1. De gebruiker wordt doorgestuurd naar de autorisatieserver, samen met een code challenge en de gewenste scopes.
2. De autorisatieserver valideert de ingevoerde gegevens van de gebruiker.
3. De autorisatieserver genereert een tijdelijke autorisatiecode.
4. Die code wordt via een redirect teruggestuurd naar de client.
5. De client wisselt de autorisatiecode achter de schermen in bij de autorisatieserver via een POST-request, samen met de originele code verifier.
6. De autorisatieserver valideert de code en de verifier.
7. De autorisatieserver retourneert een access token aan de client.
8. De client roept de resource server aan met dit token als Bearer token.
9. De resource server valideert het token bij de autorisatieserver.
10. De autorisatieserver bevestigt dat het token geldig is.
11. De resource server retourneert de gevraagde data aan de client.

**Figuur 2.3**  
Authorization Code flow



*Opmerking.* Eigen werk, gebaseerd op *The OAuth 2.0 Authorization Framework*, door Hardt (2012). Internet Engineering Task Force. <https://datatracker.ietf.org/doc/html/rfc6749/>. Copyright 2012, IETF Trust.

De Proof Key for Code Exchange (PKCE)<sup>12</sup> is een uitbreiding op de Authorization Code flow die bescherming biedt tegen code-onderscheppingsaanvallen (Sakimura et al., 2015). Bij het starten van de flow genereert de client een willekeurige waarde, de zogenaamde code verifier. Van deze waarde wordt een hash berekend, de code challenge, die wordt meegestuurd met het eerste autorisatierequest. Wanneer de client later de autorisatiecode inwisselt voor een token, stuurt hij de originele code verifier mee. De autorisatieserver controleert of de hash van de ontvangen verifier overeenkomt met de eerder ontvangen challenge. Als een kwaadwillende partij de autorisatiecode onderschept, kan deze de code niet gebruiken zonder de bijbehorende verifier.

PKCE werd oorspronkelijk ontwikkeld voor mobiele en single-page applicaties die geen client secret veilig kunnen opslaan, maar wordt tegenwoordig aanbevolen voor alle typen applicaties, ook

<sup>12</sup> PKCE wordt uitgesproken als “pixy”.

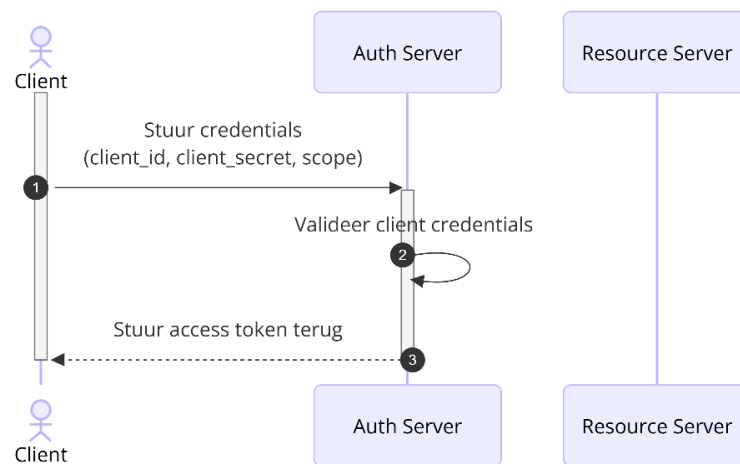
server-side webapplicaties zoals Lexis (Lodderstedt et al., 2025). Deze flow wordt in Lexis gebruikt voor de interactieve aanmelding van gebruikers in de webinterface.

### 2.3.2. Client Credentials flow

De Client Credentials flow is fundamenteel anders dan de Authorization Code flow met PKCE, omdat er geen eindgebruiker bij betrokken is (Hardt, 2012). De applicatie authenticceert zichzelf rechtstreeks bij de autorisatieserver door haar client ID en client secret mee te sturen (Figuur 2.4). De autorisatieserver verifieert deze gegevens en retourneert een access token. Eenmaal in het bezit van dit token roept de client de resource server aan als Bearer token. Dit is identiek aan stappen 8 tot en met 11 in Figuur 2.3.

**Figuur 2.4**

*Client Credentials flow*



*Opmerking.* Eigen werk, gebaseerd op *The OAuth 2.0 Authorization Framework*, door Hardt (2012). Internet Engineering Task Force. <https://datatracker.ietf.org/doc/html/rfc6749/>. Copyright 2012, IETF Trust.

Deze flow is geschikt voor machine-to-machine-communicatie, waarbij een server-applicatie op de achtergrond taken uitvoert zonder tussenkomst van een gebruiker. In het Lexis-project wordt de Client Credentials flow gebruikt voor de geautomatiseerde verwerking van e-mails. Wanneer een e-mail binnenkomt in Lexis, moet de backend zelfstandig communiceren met de CERM API om bestellingen te plaatsen. Omdat er op dat moment geen gebruiker is die interactief kan inloggen, authenticceert de Lexis-backend zich rechtstreeks als machine-client bij de autorisatieserver van de betreffende tenant.

Tabel 2.3 vergelijkt de twee OAuth2 grant types die binnen Lexis worden toegepast op basis van de betrokken partijen, de aanwezigheid van een eindgebruiker, het beoogde gebruik en de vereiste voor PKCE.

**Tabel 2.3**  
*Vergelijking van OAuth2 grant types toegepast in Lexis*

|                        | Authorization Code + PKCE               | Client Credentials              |
|------------------------|-----------------------------------------|---------------------------------|
| Betrokken partijen     | Gebruiker, client, authorization server | Client, authorization server    |
| Eindgebruiker aanwezig | Ja                                      | Nee                             |
| Gebruik                | Interactieve login                      | Machine-to-machine communicatie |
| Gebruik in Lexis       | Gebruikersauthenticatie via CERM7       | Communicatie met CERM REST API  |
| PKCE vereist           | Ja                                      | Nee                             |

*Opmerking.* Eigen werk, gebaseerd op *The OAuth 2.0 Authorization Framework*, door Hardt (2012). Internet Engineering Task Force. <https://datatracker.ietf.org/doc/html/rfc6749/>. Copyright 2012, IETF Trust.

## 2.4. Implementatie OAuth2 in Lexis

De implementatie van OAuth2 binnen Lexis bracht specifieke uitdagingen met zich mee door de multi-tenantarchitectuur van de applicatie. Elke tenant beschikt over een eigen CERM OAuth2-server, wat betekent dat de autorisatieconfiguratie dynamisch moet worden geladen op basis van de tenant die de applicatie benadert. In deze sectie wordt de gekozen architectuur en de concrete flow beschreven.

### 2.4.1. Architectuur

Voor de OAuth2-implementatie werd gekozen voor het Backend-for-Frontend (BFF) patroon (Microsoft, 2025a). Bij dit architectuurpatroon verloopt alle communicatie met de identity provider uitsluitend via de backend. De browser ontvangt nooit rechtstreeks een access token of refresh token. In plaats daarvan beheert de server de volledige sessie en ontvangt de browser enkel een HTTP-only sessiecookie.

De keuze voor het BFF-patroon biedt meerdere voordelen. Tokens worden nooit blootgesteld aan de clientzijde, waardoor het risico op tokendiefstal via Cross-Site Scripting (XSS) sterk gereduceerd wordt. Daarnaast wordt de complexiteit van tokenverversing en sessiebeheer volledig afgehandeld door de server, wat de frontend aanzienlijk eenvoudiger maakt.

Binnen de multi-tenantopzet van Lexis wordt de tenant geïdentificeerd op basis van het subdomein. Wanneer een gebruiker navigeert naar bijvoorbeeld `tenant.lexis.com`, herkent de ASP.NET-middleware dat het om de tenant “drukkerij” gaat en laadt de bijbehorende OAuth2-configuratie dynamisch in. Dit vereiste een aangepaste configuratiestrategie in ASP.NET, omdat de standaard OAuth-middleware ervan uitgaat dat er één vaste identity provider is.

### 2.4.2. Flow

De concrete autorisatieflow voor interactieve gebruikers in Lexis verloopt als volgt. Wanneer een niet-geauthenticeerde gebruiker de applicatie benadert, genereert de ASP.NET-backend een PKCE-code verifieer en berekent daarvan de code challenge (Sakimura et al., 2015). De gebruiker wordt doorgestuurd naar de CERM OAuth2-server van de betreffende tenant, samen met de code challenge, de gewenste scopes en een state-parameter ter bescherming tegen Cross-Site Request Forgery (CSRF) (Figuur 2.3) (Hardt, 2012).

Na succesvolle aanmelding stuurt de CERM-server de gebruiker terug naar Lexis met een autorisatiecode. De ASP.NET-backend wisselt deze code, samen met de originele code verifier, in bij de autorisatieserver voor een access token en een refresh token. Deze tokens worden server-side opgeslagen in de sessie en nooit naar de browser gestuurd. De gebruiker ontvangt enkel een beveiligde sessiecookie waarmee volgende requests worden geauthenticeerd.

Naast de PKCE-bescherming vereist Lexis voor alle state-wijzigende HTTP-requests (POST, PUT en DELETE) een antiforgery-token. Dit CSRF-token wordt door de server gegenereerd en door de ASP.NET-middleware automatisch gevalideerd bij elk muterend request (OWASP, 2025; Microsoft, 2025a). Zo worden aanvallen waarbij een kwaadwillige website onbewust requests namens een ingelogde gebruiker verstuurt, structureel geblokkeerd.

### 2.4.3. Gebruik van AI bij de implementatie

De OAuth2-implementatie werd gerealiseerd met ondersteuning van Cursor (2026), een AI-gestuurde ontwikkelomgeving die een conversatie met de codebase mogelijk maakt. Omdat OAuth2 voor dit project voor het eerst werd geïmplementeerd, vervulde Cursor een driedelige rol: kennisopbouw, codegeneratie en debugging.

In een eerste fase werd Cursor ingezet om de OAuth2-concepten te begrijpen en te vertalen naar de concrete projectcontext. Door gerichte vragen te stellen over de werking van de Authorization Code flow met PKCE, het BFF-patroon en de multi-tenant structuur, kon snel bijgeleerd worden zonder uitsluitend te steunen op officiële documentatie. Dit versnelde het leerproces aanzienlijk, maar verving de officiële documentatie niet.

Vervolgens werd Cursor gebruikt voor de codegeneratie van de meest complexe onderdelen, zoals de dynamische laadstrategie voor de OAuth2-configuratie per tenant en de PKCE-logica in de ASP.NET-middleware. De gegenereerde code diende telkens als vertrekpunt dat nadien manueel werd gecontroleerd, aangepast en getest.

Ten slotte werd Cursor ook ingezet bij het debuggen van problemen in de autorisatieflow, zoals foutief opgebouwde redirect URI's en ontbrekende scopes in het tokenrequest. Door foutmeldingen en relevante codefragmenten samen aan de AI voor te leggen, konden oorzaken sneller worden geïdentificeerd dan via enkel manuele inspectie. De uiteindelijke oplossing werden steeds zelfstandig geverifieerd.

## 3. Dataset Explorer

Om de kwaliteit van het Lexis-systeem te kunnen meten, is een representatieve dataset van verwerkte e-mails en de verwachte bijhorende LLM-output onmisbaar. Het beheren, valideren en corrigeren van deze dataset vormt een arbeidsintensief proces. Een gespecialiseerde tool zou dit proces verlichten. In dit hoofdstuk wordt de ontwikkeling van de Dataset Explorer beschreven: een interne webapplicatie die het validatieproces ondersteunt. De opbouw van dit hoofdstuk volgt het traject van een probleem (3.1), de implementatie (3.2) tot de resulterende applicatie (3.3). Afgesloten met een reflectie op het ontwikkelproces (3.4).

### 3.1. Probleemstelling

Het valideren van LLM-uitvoer vereist een gestructureerde aanpak waarbij invoerdata en gegenereerde uitvoer systematisch met elkaar worden vergeleken. Zonder de juiste tooling leidt dit proces snel tot fouten en tijdverlies. Dit onderdeel beschrijft de context die aanleiding gaf tot de ontwikkeling van de Dataset Explorer en de functionele grenzen van de tool.

#### 3.1.1. Context

Bij de ontwikkeling van het Lexis-project worden inkomende e-mails verwerkt door een LLM. Om een representatieve en realistische dataset samen te stellen, werden echte e-mails opgevraagd bij klanten van CERM. Dit resulteerde in een dataset afkomstig van verschillende klanten, elk met hun eigen e-mailformaten, bijlagen en bestelconventies.

Elk e-mailrecord diende manueel verwerkt te worden: de developer maakte verbinding met de database van de betrokken klant, zocht de bijhorende bestelling op in de CERM-software, en extraheerde de relevante bestelgegevens. De resultaten werden opgeslagen als JSON-bestanden op het bestandssysteem. Dit was een arbeidsintensief proces met veel ruimte voor menselijke fouten, zeker naarmate de dataset in omvang toenam.

De doelstelling was het ontwikkelen van een dedicated tool waarmee een gebruiker efficiënt e-mails kon doorlopen, de verwachte LLM-uitvoer kon bewerken en valideren, en een labelstatus kon toekennen aan elk e-mailrecord.

#### 3.1.2. Scope

De Dataset Explorer moest een interne webapplicatie worden die het volledige validatieproces ondersteunt voor datasets van honderd of meer e-mails, verdeeld over meerdere klanten. De tool moest elke e-mail weergeven samen met de bijhorende bijlagen zodat de gebruiker alle nodige informatie op één scherm beschikbaar had.

Per e-mail moesten bewerkbare invoervelden beschikbaar zijn voor de verplichte bestelgegevens: klantnummer, ordernummer en een of meerdere orderregels met velden zoals productnummer, hoeveelheid, eenheidsprijs en leveringsgegevens. Op basis van de ingevulde gegevens moest de applicatie een gestructureerd JSON-bestand genereren en dit opslaan hostmachine.

Het moest ook mogelijk zijn om een reviewstatus toe te kennen aan elke e-mail. Zo kon een onderscheid worden gemaakt tussen gevalideerde records, records die nog nagelopen moesten worden, en records die om een andere reden niet verwerkt konden worden. Dit gaf het team een overzicht van de validatievoortgang en maakte gerichte opvolging mogelijk.

## 3.2. Implementatie

De Dataset Explorer werd volledig gerealiseerd via AI-ondersteunde ontwikkeling. Van de initiële architectuur tot de iteratieve uitbreidingen werd geen code handmatig geschreven. Dit onderdeel beschrijft de technologische keuzes, de ontwikkelingsaanpak, de datastructuur en de technische obstakels die tijdens het proces werden overwonnen.

### 3.2.1. Technologie

De AI-applicatie kreeg de instructies om de applicatie in Python te ontwikkelen. Het moest gebruikmaken van een objectgeoriënteerde structuur zodat de codebase overzichtelijk en uitbreidbaar bleef. Voor de gebruikersinterface werd gevraagd Plotly Dash te gebruiken, een Python-framework voor het bouwen van webapplicaties (Plotly Technologies Inc., 2024). De keuze voor Plotly Dash was genomen voor de praktische eigenschap dat zowel de gebruikersinterface als de serverlogica in één Python-programma kunnen worden geïmplementeerd, zonder nood aan een aparte frontend-omgeving. Omdat de testdata al in Python-scripts werd gegenereerd en verwerkt, paste Dash naadloos in het bestaande ecosysteem. Het framework laat ook het inlezen en wegschrijven van JSON-bestanden toe, wat essentieel was voor het validatieproces. De interface werd bewust eenvoudig gehouden als startpunt, waarna functionaliteit stapsgewijs werd toegevoegd.

### 3.2.2. Vibe-coding met Cursor

De Dataset Explorer werd ontwikkeld via een aanpak die omschreven wordt als *vibe coding*: een werkwijze waarbij de ontwikkelaar de gewenste functionaliteit beschrijft in natuurlijke taal en een AI-assistent de bijhorende code genereert (Karpathy, 2025). Als onderliggend model werd gebruikgemaakt van Claude Opus 4.6 (Anthropic, 2026b).

De initiële prompt was bewust eenvoudig gehouden:

#### Figuur 3.1

*Initiële prompt voor de Dataset Explorer-applicatie*

```
Generate a Plotly Dash application to explore the @data/ folder. The goal is to validate and edit output JSONs versus the input emails.
```

*Opmerking.* Eigen werk.

Voordat Cursor met de implementatie begon, werd automatisch een planningsfase doorlopen. In deze fase analyseerde het model de bestaande mappenstructuur en bestandsinhoud in de “data” folder om de datastructuur te begrijpen en een implementatieplan op te stellen. Deze werkwijze, waarbij het model eerst context vergaart via directe bestandstoegang alvorens code te genereren, sluit aan bij agentic LLM’s die eerder in deze bachelorproef werden besproken in sectie 1.4.

Vanuit die eerste versie werd de applicatie iteratief uitgebreid via een reeks gerichte vervolgprompts. Elke iteratie voegde een nieuwe feature of verbetering toe, van URL-routing tot validatielabels. De rol van de ontwikkelaar bestond erin de gewenste functionaliteit te omschrijven, de gegenereerde code te beoordelen, en eventuele problemen te rapporteren aan de AI-assistent.

### 3.2.3. Datastructuur

De testdata is per tenant georganiseerd in drie mappen, zoals weergegeven in :

- `raw_data/`: de originele e-mails als .msg-bestanden

- `preprocessed_data/`: JSON-bestanden met de geëxtraheerde e-mailmetadata, inclusief eventuele bijlagen
- `output/`: de door het LLM gegenereerde besteldata als JSON-bestanden

**Figuur 3.2***Folder structuur datasets*

```
dataset-<tenant>/
├── output/
│   ├── <id>.json
│   └── ...
├── preprocessed_data/
│   ├── attachments/
│   │   ├── <id>_<index>_<name>.pdf
│   │   └── ...
│   ├── <id>.json
│   └── ...
├── raw_data/
│   ├── <name>.msg
│   └── ...
└── validated.json
```

*Opmerking. Eigen werk.*

### 3.2.4. Obstakels

Tijdens de implementatie deed zich een technisch probleem voor met de autosave-functionaliteit. Dash maakt gebruik van pattern-matching callbacks. Deze callbacks reageren op een variabel aantal dynamisch gegenereerde componenten. Bij het initieel laden van de e-maillijst werden deze callbacks onverwacht getriggerd, waardoor de applicatie incorrecte data probeerde op te slaan voordat een e-mail geselecteerd was. Hoewel een tijdelijke oplossing werd gevonden via een expliciete controle die de callback vroegtijdig afbreekt, bleef het gedrag van de autosave onvoorspelbaar en onvoldoende betrouwbaar. Uiteindelijk werd de functionaliteit volledig verwijderd. Dit illustreert dat ook bij vibe coding technische kennis van het framework vereist blijft om problemen correct te kunnen diagnosticeren en een weloverwogen beslissing te nemen over de haalbaarheid van een feature.

## 3.3. Resulterende applicatie

De initiële versie van de applicatie was verrassend volledig voor de omvang en complexiteit van de gevraagde functionaliteit. De Dataset Explorer is een webapplicatie die lokaal wordt gehost en toegankelijk is via de browser. De interface is opgebouwd uit een zijbalk en een hoofdpaneel dat de details van de geselecteerde e-mail weergeeft.

**Figuur 3.3***Dataset explorer overview pagina*

The screenshot shows the 'Email Orders' overview page. On the left, there's a sidebar with a search bar and a list of email subjects with status badges (OK, REV, HOLD, REJ). The main area is titled 'Overview' and contains a 'Status Filter' dropdown set to 'All', an 'Export Visible Emails (97 items)' button, and a table of email details. The table has columns: Status, Subject, Lines, Attachments, Flags, Comment, and Actions. The 'Status' column uses colored badges: green for OK, orange for HOLD, blue for REV, and grey for OK. The 'Flags' column contains error messages like 'missing\_reference (1)', 'missing\_expected\_date (1)', 'bad\_customerid', and 'missing\_unitprice\_or\_zero (1)'. The 'Actions' column has 'Open' buttons and document icons.

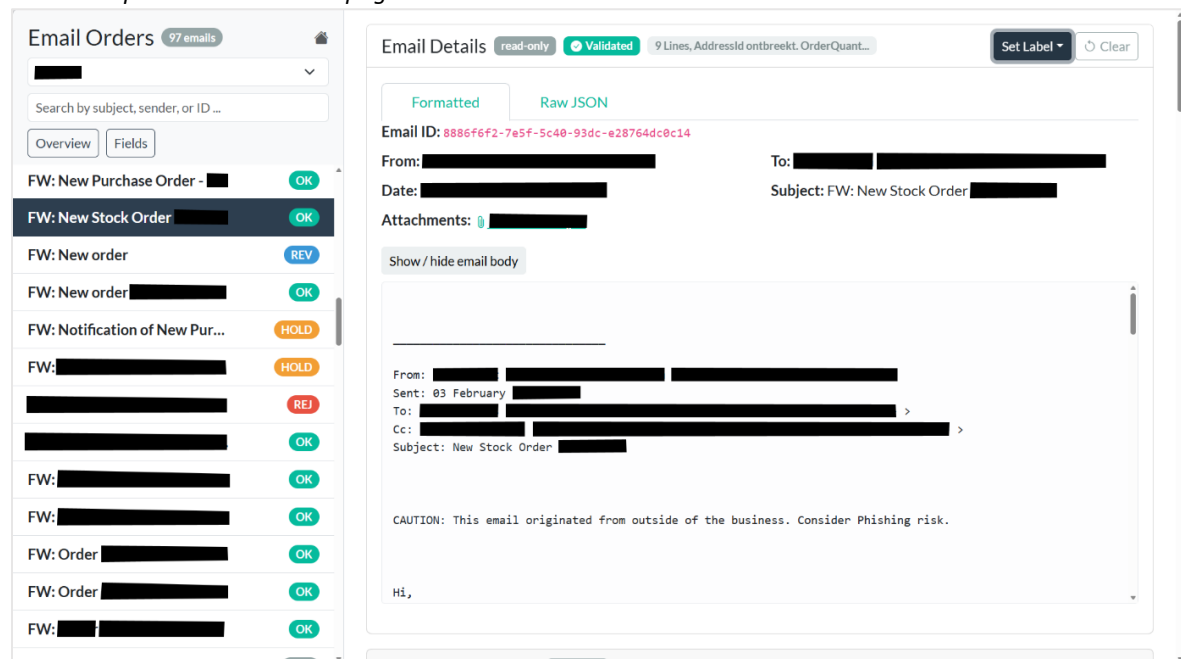
| Status | Subject        | Lines | Attachments | Flags                                                                | Comment                | Actions |
|--------|----------------|-------|-------------|----------------------------------------------------------------------|------------------------|---------|
| OK     | [Redacted]     | 4     | 1           |                                                                      |                        | Open    |
| HOLD   | FW: [Redacted] | 1     | 0           | missing_reference (1)<br>missing_expected_date (1)<br>bad_customerid | [Redacted]             | Open    |
| OK     | [Redacted]     | 1     | 2           | missing_unitprice_or_zero (1)                                        | Unit Price is 0 on PDF | Open    |
| OK     | [Redacted]     | 5     | 1           | missing_unitprice_or_zero (2)                                        |                        | Open    |
| OK     | [Redacted]     | 3     | 2           |                                                                      |                        | Open    |

*Opmerking. Eigen werk.*

De zijbalk in Figuur 3.3 toont een gesorteerde, doorzoekbare lijst van alle e-mails in de dataset. Elke rij toont het onderwerp en de afzender van de e-mail. Een zoekbalk bovenaan filtert de lijst op basis van onderwerp, afzender of e-mail-ID. Aan elke e-mail kan een labelstatus worden toegekend die als gekleurde badge naast de e-mail verschijnt: *Validated* (groen), *Rejected* (rood), *Needs Review* (blauw), *On Hold* (geel) en *Skipped* (grijs). Dit biedt een visueel overzicht van de voortgang van het validatieproces.

### 3.3.1. Functionaliteiten

Het e-maildetailpanel in Figuur 3.4 toont de metadata van de geselecteerde e-mail: afzender, ontvanger, datum, onderwerp en e-mail-ID. De volledige e-mailtekst is in een inklapbare vorm weergegeven. Een tweede tabblad, die "Raw JSON" noemt, toont de preprocessed data als onbewerkte tekst, wat nuttig is bij het debuggen of manueel controleren van de extractie. Wanneer de e-mail bijlagen bevat, worden PDF-bijlagen inline getoond via een ingebouwde PDF-viewer.

**Figuur 3.4***Dataset explorer e-mail details pagina*

*Opmerking.* Eigen werk.

Via de knoppen in de koptekst van het panel kan een label worden toegekend aan de geselecteerde e-mail. Bij het selecteren van een label opent een popup waarin een optionele toelichting kan worden ingegeven.

De output-editor biedt twee weergaven via tabs. In de formulierweergave worden de bestelgegevens getoond als bewerkbare formulievelden, gegroepeerd per orderregel (zie Figuur 3.5). Bovenaan bevinden zich velden voor de klant- en orderidentificatie. Daaronder volgen de orderregels, elk weergegeven als een uitklapbare kaart met velden voor productinformatie, hoeveelheid, prijs en leveringsgegevens. Orderregels kunnen worden toegevoegd of verwijderd. Voor het leveringsdatumveld wordt een datumkiezer gebruikt die de datum automatisch opmaakt in het vereiste formaat.

**Figuur 3.5**  
Output editor van een e-mail

The screenshot displays the 'Email Orders' interface on the left and the 'Output Editor' on the right. The 'Email Orders' panel lists 97 emails with a search bar and filters. The 'Output Editor' panel has two tabs: 'Form Editor' and 'JSON Editor'. Under 'Form Editor', there are input fields for 'CustomerId', 'PO Number', and 'Order Line 1'. 'Order Line 1' includes fields for 'ProductId', 'OrderQuantity' (set to 40000), and 'UnitPrice'. There are also 'Add Line', 'Save', and 'Delete' buttons. 'Order Line 2' is partially visible below.

*Opmerking.* Eigen werk.

Beide weergaven maken gebruik van een expliciete knop om op te slaan. Wijzigingen in de formulier-weergave worden pas weggeschreven naar het bestandssysteem na een bewuste bevestiging; in de JSON-editor wordt bovendien ongeldige invoer gedetecteerd en gemeld aan de gebruiker.

### 3.3.2. Navigatie en export

Elke e-mail heeft een unieke URL op basis van het e-mail-ID, bijvoorbeeld `'/<dataset>/email/a3f2...'`. Zo kan een specifieke e-mail eenvoudig worden gedeeld met collega's door de URL te kopiëren en door te sturen.

Via de overzichtspagina biedt de applicatie ook een exportfunctie (zie Figuur 3.3 op pagina 56). Met één klik genereert de tool een zip bestand van alle e-mails die op dat moment zichtbaar zijn in de lijst, rekening houdend met actieve filters of zoekopdrachten. Het archief bundelt per e-mail de ruwe data, de preprocessed data en de bijhorende output-JSON. Op die manier beschikt de gebruiker over een volledig, zelfstandig pakket met alle bestanden die nodig zijn voor verdere verwerking of archivering, zonder manueel door de mappenstructuur te navigeren.

### 3.3.3. Dashboard

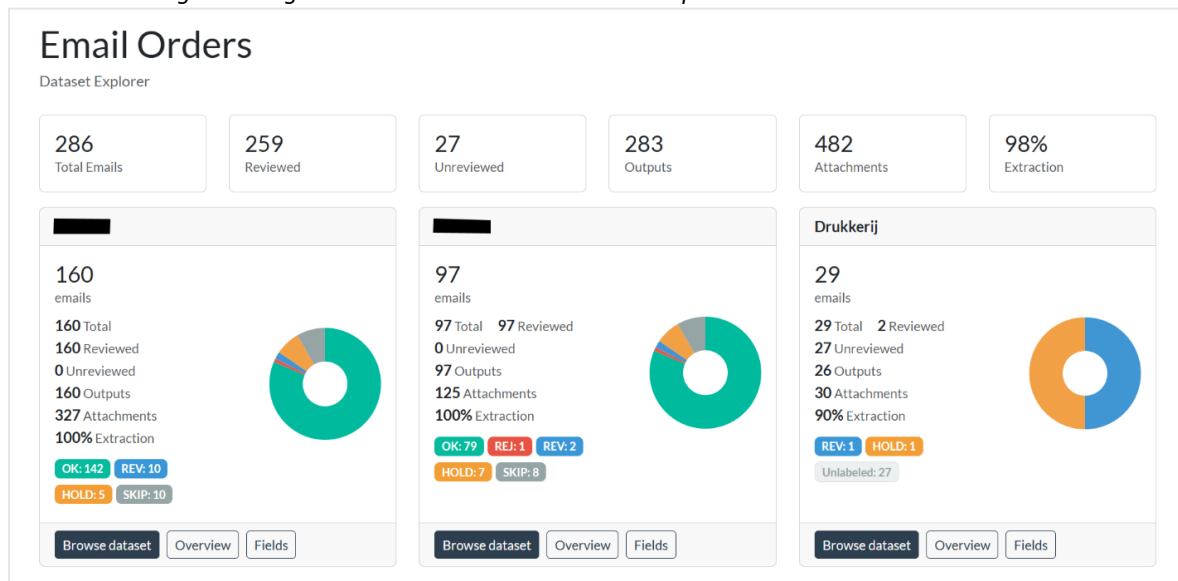
De dashboardweergave biedt een overzicht op twee niveaus (zie Figuur 3.6). Bovenaan worden globale statistieken over de volledige dataset getoond als metriekkaarten: het totale aantal e-mails, het aantal beoordeelde en onbeoordeelde records, het aantal gegenereerde outputs, het totale aantal bijlagen en het extractiepercentage van het LLM.

Daaronder volgt een opdeling per klant. Elke klant wordt weergegeven als een aparte kaart met dezelfde statistieken op klantniveau, aangevuld met een ringdiagram dat de verdeling van de label-statussen visueel weergeeft. Onderaan elke kaart worden de aantallen per label getoond als

gekleurde badges. Via de navigatieknoppen kan rechtstreeks naar de e-maillijst van die klant worden gesprongen.

**Figuur 3.6**

*Dashboardweergave met globale statistieken en een overzicht per klant*



*Opmerking.* Eigen werk.

## 3.4. Reflectie

De Dataset Explorer werd ontwikkeld als intern hulpmiddel en doorliep een volledig traject van probleemstelling tot werkende applicatie. Deze reflectie evalueert het resultaat, de gebruikte ontwikkelmethode, de beperkingen van de aanpak en de leermomenten die hieruit voortvloeien.

### 3.4.1. Resultaten

De Dataset Explorer werd ingezet om de volledige testdataset samen met collega's bij CERM te valideren. De tool maakte het mogelijk om e-mails systematisch door te nemen, de LLM-uitvoer te vergelijken met de originele e-mail, en fouten te corrigeren of te labelen voor verdere opvolging. Vergeleken met het handmatig openen van bestanden werd het validatieproces aanzienlijk sneller en gestructureerder.

### 3.4.2. Vibe coding als ontwikkelmethode

De voornaamste meerwaarde van vibe coding bij de ontwikkeling van de Dataset Explorer lag in de snelheid en de lage instapdrempel. Een werkende, professioneel ogende webapplicatie werd opgeleverd zonder dat de ontwikkelaar de onderliggende code schreef of doornam. De tijdsinvestering bleef beperkt tot het formuleren van instructies, het beoordelen van het visuele resultaat en het rapporteren van afwijkingen aan de AI-assistent.

De kwaliteit van het eindresultaat hing daarbij sterk af van de precisie van de prompts. Dit werd concreet zichtbaar bij de autosave-functionaliteit: een vaag geformuleerde opdracht leidde tot een implementatie met onvoorspelbaar gedrag, die uiteindelijk volledig verwijderd moest worden. Meer gedetailleerde instructies, met expliciete vermelding van het gewenste gedrag, de datastructuur en de randgevallen, leidden tot aanzienlijk betere resultaten. Dit sluit aan bij de

principes van prompt engineering die eerder in deze bachelorproef werden besproken (zie sectie 1.1).

Een kritisch aandachtspunt is dat de afwezigheid van inzicht in de gegenereerde code een risico vormt bij het diagnosticeren van fouten. Bij het autosave-probleem kon het onderliggende oorzaak-gevolgmechanisme worden herleid op basis van het zichtbare gedrag, maar een diepere technische analyse van de code was niet haalbaar. Dit beperkt de controle over het eindproduct en maakt de aanpak minder geschikt voor situaties waarbij betrouwbaarheid of veiligheid een hogere prioriteit heeft.

### 3.4.3. Beperkingen

Omdat de applicatie volledig door een AI-assistent werd gegenereerd zonder manueel inzicht in de werking ervan, is de onderhoudbaarheid beperkt. Het debuggen van onverwacht gedrag of het uitbreiden van de codebase vereist meer inspanning dan wanneer de code van begin tot eind zelf geschreven was. Dit sluit aan bij de risico's van AI-ondersteunde ontwikkeling die eerder in deze bachelorproef werden besproken, in het bijzonder het verlies van domein- en codekennis en de te grote afhankelijkheid van AI (zie secties 1.6.1; 1.6.4).

Daarnaast is de applicatie niet geoptimaliseerd voor prestaties. Het is ontwikkeld voor een beperkt aantal interne gebruikers, daarom is dit geen bezwaar. Voor een productieomgeving of een grotere schaal zou een meer doordachte architectuur noodzakelijk zijn.

### 3.4.4. Leermomenten en aanbevelingen

Plotly Dash bleek een geschikte keuze voor dit type interne validatietool: snel op te zetten, uitbreidbaar, en zonder nood aan een aparte frontend-infrastructuur. Voor grotere applicaties waarbij meerdere gebruikers gelijktijdig werken of waarbij real-time synchronisatie vereist is, zou een traditionele client-serverarchitectuur aangewezen zijn.

## 4. Bruno

Wanneer een tool waarop een team dagelijks vertrouwt zijn gratis aanbod inperkt, ontstaat de noodzaak om alternatieven te evalueren. Voor het Lexis-project was dit het geval met Postman: een licentiewijziging maakte teamsamenwerking betalend, en de cloudgebonden opslag van collecties bleek niet te voldoen met de vereisten voor versiebeheer en CI/CD-integratie. Dit hoofdstuk beschrijft hoe die situatie aanleiding gaf tot de migratie naar Bruno als API-client. Eerst wordt de problematiek rond Postman toegelicht. Vervolgens wordt de keuze voor Bruno onderbouwd via een vergelijkende evaluatie, waarna de implementatie stap voor stap wordt beschreven. Het hoofdstuk sluit af met de behaalde resultaten en een reflectie op de aanpak.

### 4.1. Probleem

Bij de ontwikkeling van Lexis was er nood aan een tool om HTTP-requests te beheren, te documenteren en geautomatiseerd uit te voeren. Het meest bekende hulpmiddel hiervoor is Postman. Postman heeft echter zijn gratis aanbod in de loop der jaren sterk ingeperkt. Waar teamfunctionaliteit vroeger kosteloos beschikbaar was voor meerdere gebruikers, is het gratis abonnement inmiddels beperkt tot één gebruiker. Voor een team dat gezamenlijk een API-collectie wil bijhouden, is een betalend teamplan vereist (Postman Inc., 2026). De kostprijs hiervan woog niet op tegen de voordelen van het gebruik binnen CERM.

Naast de licentieproblematiek slaat Postman collecties op in een eigen cloudomgeving, los van het versiebeheersysteem van het project (Postman Inc., 2026). Requests worden niet als bestanden op het bestandssysteem bewaard, waardoor het onmogelijk is om een API-collectie samen met de broncode in Git op te slaan en te beheren via pull requests. Wijzigingen aan requests zijn zo moeilijk traceerbaar en niet gekoppeld aan de onderliggende codewijzigingen.

Dit heeft ook gevolgen voor CI/CD-integratie. Omdat collecties niet als lokale bestanden beschikbaar zijn, vereist het uitvoeren ervan in een pipeline een verbinding met de cloudomgeving van Postman via een API-sleutel. Voor een project waarbij API-testen deel moesten uitmaken van een Azure DevOps-pipeline introduceert dit een externe cloudafhankelijkheid die vermijdbaar is.

Op basis van deze analyse werden drie concrete vereisten opgesteld waaraan een geschikt alternatief moest voldoen: de tool moet kosteloos bruikbaar zijn voor een team, collecties moeten worden opgeslagen als bestanden in een Git-repository, en de tool moet zonder externe cloudafhankelijkheid uitvoerbaar zijn in een CI/CD-pipeline.

### 4.2. Keuze Bruno

Bij het zoeken naar een alternatief voor Postman kwamen twee tools in aanmerking: Insomnia en Bruno. Beide tools werden geëvalueerd op de drie vereisten uit sectie 4.1.

Insomnia biedt een open-sourcevariant en ondersteunt bestandsgebaseerde collecties (Kong Inc., 2026), waarmee het voldoet aan de Git- en CI/CD-vereisten. Op het vlak van licenties stelt Insomnia beperkingen aan geavanceerde functies in de gratis versie, zoals synchronisatie en uitgebreide samenwerkingsfuncties. Om deze reden voldoet Insomnia niet volledig aan de vereisten.

Bruno voldoet aan alle drie de vereisten. De tool is volledig gratis en open source, zonder beperkingen op teamniveau (Bruno Software Inc., 2026). Alle requests worden opgeslagen als YAML-bestanden op het bestandssysteem, waardoor collecties in dezelfde Git-repository als de broncode kunnen worden bijgehouden. Omdat de collectie bestaat uit gewone bestanden, kan de

Bruno Command Line Interface (CLI) deze rechtstreeks vanuit de repository uitvoeren in een Azure DevOps-pipeline, zonder externe cloudafhankelijkheid.

Bijkomend had een teamlid bij aanvang van het project al ervaring met Bruno vanuit een eerdere professionele omgeving. Dit verlaagde de instapdrempel en bevestigde dat de tool bruikbaar is in een productiecontext. Op basis van de objectieve evaluatie én deze praktijkervaring werd Bruno gekozen als het API-testplatform voor het Lexis-project.

## 4.3. Implementatie

De implementatie van Bruno binnen het Lexis-project verliep in twee fasen. Eerst werd de tool verkend en werd de bestaande API-documentatie omgezet naar een bruikbare Bruno-collectie. Vervolgens werd de collectie operationeel gemaakt door environments, testen en pipeline-integratie toe te voegen. De onderstaande secties beschrijven elk onderdeel van deze opzet.

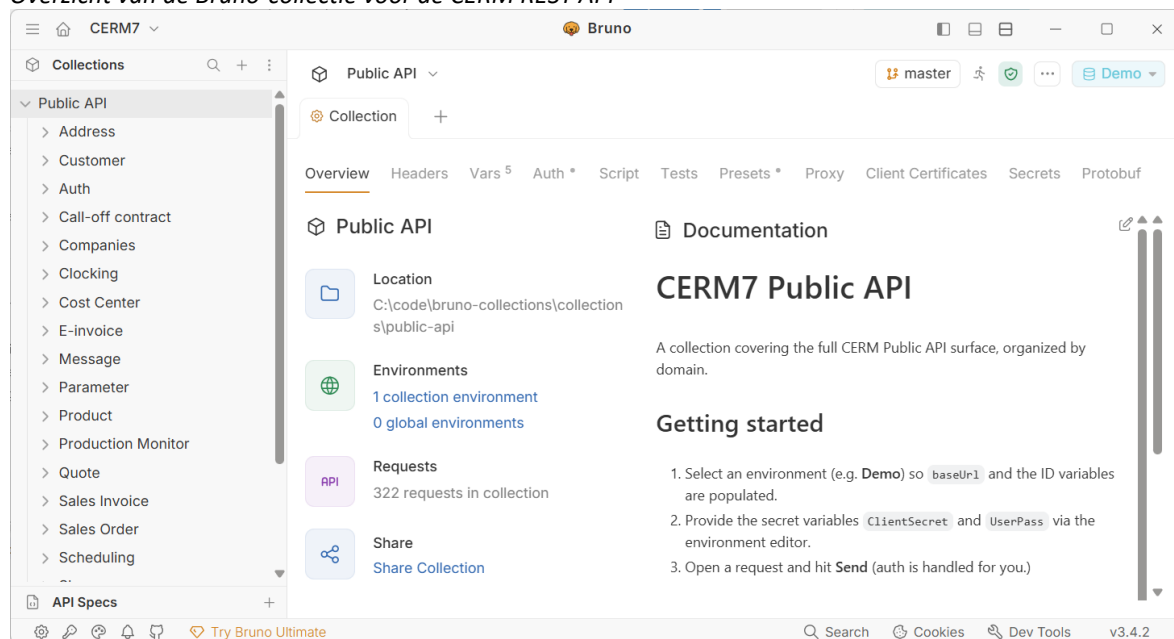
### 4.3.1. Opzet van de collectie

Bij aanvang van de implementatiefase werden van een collega 24 OpenAPI YAML Ain't Markup Language (YAML)-bestanden aangeleverd die de CERM Representational State Transfer (REST) API documenteren. OpenAPI is een open standaard voor het beschrijven van HTTP-gebaseerde API's in een machine-leesbaar formaat (the Linux Foundation, 2025).

Bruno ondersteunt het importeren van OpenAPI-specificaties, maar laat niet toe om meerdere bestanden tegelijk in één collectie te importeren (Bruno Software Inc., 2026). Elke specificatie werd daarom afzonderlijk geïmporteerd. Daarna werden de afzonderlijke collecties samengevoegd tot één geheel, waarbij een AI-assistent werd ingezet om dit proces te stroomlijnen. Dit wordt verder toegelicht in sectie 4.4. Het samenvoegen verliep grotendeels vlot, maar vereiste nadien nog aanzienlijk handmatig opkuiswerk om inconsistenties te verwijderen en de mappenstructuur te verfijnen.

**Figuur 4.1**

*Overzicht van de Bruno-collectie voor de CERM REST API*



*Opmerking.* Schermafbeelding van de Bruno interface.

In Bruno stemt een workspace overeen met een Git-repository. De volledige Bruno-collectie werd ondergebracht in een aparte, toegewijde Git-repository, los van de Lexis-broncode. Dit maakt versiebeheer op de collectie mogelijk en laat toe om wijzigingen te beheren via pull requests, net zoals bij reguliere broncode.

### 4.3.2. Environments

Bruno ondersteunt het werken met meerdere environments, vergelijkbaar met Postman (Bruno Software Inc., 2026; Postman Inc., 2026). Het verschil is dat environments in Bruno worden opgeslagen als YAML-bestanden, waardoor ze kunnen worden opgenomen in de Git-repository en gedeeld met het hele team. Via een eenvoudige omschakeling tussen environments kan worden gekozen tussen de verschillende omgevingen waarop requests worden gestuurd, zoals een lokale ontwikkelomgeving of de testomgeving op Azure.

### 4.3.3. Testen

Per request kunnen testen worden gedefinieerd die na uitvoering automatisch worden geverifieerd (Bruno Software Inc., 2026). Voor eenvoudige gevallen volstaan assertions: korte declaratieve checks die bijvoorbeeld de HTTP-statuscode of de aanwezigheid van een bepaald veld in de response valideren. Deze aanpak maakt het mogelijk om snel de basiscorrectheid van een endpoint te verifiëren zonder uitgebreide testscripts te schrijven. Het schrijven van assertions werd deels geautomatiseerd met behulp van een AI-assistent, zoals beschreven in sectie 4.4.

### 4.3.4. CLI en pipeline-integratie

Bruno beschikt over een CLI-tool waarmee collecties geautomatiseerd kunnen worden uitgevoerd (Bruno Software Inc., 2026). Met het commando `bru run` kan een volledige collectie, een subfolder of een enkel requestbestand worden uitgevoerd. Omdat de collectie bestaat uit gewone bestanden in een Git-repository, kan de pipeline deze rechtstreeks uitchecken en uitvoeren zonder afhankelijkheid van een externe clouddienst.

Deze CLI werd geïntegreerd in de Azure DevOps CI/CD-pipelines van het Lexis-project. Na elke deployment worden de Bruno-testen automatisch uitgevoerd als validatiestap. De resultaten worden gegenereerd in drie formaten: HTML voor menselijk leesbare rapportage, JUnit XML voor integratie met de testrapporten van Azure DevOps, en JavaScript Object Notation (JSON) voor verdere verwerking. JUnit XML is het standaard rapportageformaat dat door het JUnit-testframework wordt gebruikt en dat door Azure DevOps wordt herkend voor het weergeven van testresultaten.

### 4.3.5. Toekomstige uitbreidingen

Twee functionaliteiten van Bruno werden geëvalueerd maar konden in de huidige projectfase nog niet worden uitgerold.

Bruno biedt een ingebouwde secret manager voor het veilig beheren van gevoelige waarden zoals API-sleutels en tokens (Bruno Software Inc., 2026). Deze functionaliteit vereist echter een betaalde licentie. Omdat CERM heeft besloten deze licentie voorlopig niet aan te schaffen, valt de secret manager buiten scope voor de huidige implementatie. Gevoelige waarden worden in de tussentijd

beheerd via de environment-bestanden, waarbij geheimen niet worden opgenomen in de Git-repository.

Bruno ondersteunt een OpenAPI Sync-functionaliteit waarmee een collectie automatisch in sync kan worden gehouden met een OpenAPI-specificatiebestand (Bruno Software Inc., 2026). Wanneer de API-definitie wijzigt, worden de bijhorende requests in de collectie automatisch bijgewerkt. Deze feature werd getest en biedt potentieel om de collectie altijd actueel te houden ten opzichte van de CERM API-specificatie. De uitrol is echter nog niet mogelijk omdat de benodigde infrastructuur bij CERM hiervoor nog niet beschikbaar is. Beide functionaliteiten worden beschouwd als toekomstige uitbreidingen van de implementatie.

## 4.4. Gebruik van AI

Een belangrijk voordeel van Bruno's bestandsgebaseerde aanpak is de compatibiliteit met AI-assistenten. Omdat alle requests, environments en testen zijn opgeslagen als gewone YAML-bestanden in een Git-repository, kan een AI-editor zoals Cursor de bestanden rechtstreeks lezen, analyseren en aanpassen. Dit onderscheidt Bruno fundamenteel van Postman, waar collecties enkel via een grafische interface of een cloud-API toegankelijk zijn.

Dit voordeel werd doorheen de implementatie op verschillende manieren benut. Bij het samenvoegen van de afzonderlijk geïmporteerde collecties tot één geheel stelde Cursor een aanpak voor en voerde die iteratief uit over de bestanden. Het handmatig uitvoeren van die stap over een groot aantal request-bestanden zou aanzienlijk meer tijd hebben gevraagd.

Ook het schrijven van assertions werd grotendeels door AI ondersteund. Cursor kon per request de structuur van de response afleiden uit de YAML-definitie en op basis daarvan automatisch basis-assertions genereren. Dit maakte het mogelijk om in korte tijd een volledige testdekking op te bouwen over de hele collectie (Anysphere, 2026; the Linux Foundation, 2025).

Daarnaast werd Cursor ingezet voor de configuratie van de Azure DevOps pipeline. Het YAML-formaat van de pipelinedefinitie sluit goed aan op wat AI-modellen kunnen genereren, en de gegenereerde configuratie vereiste slechts minimale aanpassingen.

Ten slotte werd AI gebruikt om de correctheid van de migratie te verifiëren. Door de originele Postman-export en de OpenAPI YAML-specificaties te vergelijken met de gegenereerde Bruno-bestanden kon Cursor inconsistenties en ontbrekende requests opsporen die bij een manuele controle eenvoudig over het hoofd gezien zouden worden.

## 4.5. Resultaten

De implementatie van Bruno binnen het Lexis-project heeft een werkende API-testinfrastructuur opgeleverd die het team actief gebruikt. De volledige CERM REST API-collectie is overgezet naar Bruno en omvat meer dan 350 requests, verdeeld over meer dan 400 subfolders die overeenkomen met de verschillende modules van de CERM API. Voor elk request is minimaal één assertion gedefinieerd die verifieert of het endpoint correct reageert, goed voor meer dan 1.300 assertions in totaal.

De Azure DevOps-pipeline voert de testen automatisch uit na elke deployment en genereert een rapport in HTML-, JUnit XML- en JSON-formaat. Dit geeft het team directe feedback over de toestand van de API na een wijziging, zonder dat hiervoor manuele inspanning vereist is.

Ten opzichte van de situatie vóór de implementatie zijn drie concrete verbeteringen merkbaar. Ten eerste kunnen meerdere teamleden gratis en gelijktijdig aan de collectie werken, omdat ze in Git is opgeslagen. Wijzigingen zijn traceerbaar en kunnen via pull requests worden beoordeeld. Ten tweede is de pipeline-integratie volledig onafhankelijk van externe clouddiensten: de collectie wordt rechtstreeks vanuit de repository uitgevoerd. Ten derde biedt het bestandsgebaseerde formaat compatibiliteit met AI-assistenten, wat tijdens de implementatie werd ingezet om repetitieve taken te automatiseren. Dit wordt verder toegelicht in sectie 4.4.

De collectie is operationeel voor het Lexis-team en vormt een stabiele basis voor verdere uitbreiding. In een volgende fase kunnen de secret manager en de OpenAPI Sync worden geïntegreerd om het beheer van gevoelige waarden te centraliseren en de collectie automatisch gesynchroniseerd te houden met de evoluerende CERM REST API-specificaties.

## 4.6. Reflectie

Bruno werd ingezet als vervanging voor Postman binnen het Lexis-project en heeft de drie vooropgestelde vereisten volledig ingelost: gratis teamsamenwerking, versiebeheer via Git en cloudvrije pipeline-integratie. Deze reflectie evalueert de gehanteerde aanpak, de beperkingen en de leermomenten die hieruit voortvloeien.

### 4.6.1. Bestandsgebaseerde API-collecties als aanpak

De keuze voor een bestandsgebaseerde tool bleek niet alleen een oplossing voor de licentie- en Git-problematiek, maar had ook een onverwacht positief neveneffect: de compatibiliteit met AI-assistenten. Doordat alle requests en testen als YAML-bestanden in de repository staan, kon Cursor de collectie rechtstreeks lezen en aanpassen. Dit maakte het mogelijk om repetitieve taken, zoals het samenvoegen van collecties en het genereren van assertions, te automatiseren op een schaal die manueel niet haalbaar zou zijn geweest. De bestandsgebaseerde aanpak is dan ook niet enkel een technische keuze, maar hetgeen dat AI-ondersteuning toelaat.

### 4.6.2. Beperkingen

De implementatie kent een aantal beperkingen die in een volgende fase aandacht verdienen. De ingebouwde secret manager van Bruno vereist een betaalde licentie, waardoor gevoelige waarden voorlopig buiten de Git-repository worden beheerd via environment-bestanden. Dit is functioneel, maar minder gecentraliseerd en vatbaar voor menselijke fouten. Daarnaast kon de OpenAPI Sync-functionaliteit nog niet worden uitgerold door het ontbreken van de nodige infrastructuur bij CERM. Naarmate de CERM REST API verder evolueert, zal de collectie manueel bijgehouden moeten worden totdat deze synchronisatie operationeel is.

### 4.6.3. Leermomenten en aanbevelingen

Bruno bleek een geschikte keuze voor een team dat API-collecties wil beheren als onderdeel van een softwareontwikkelproces met versiebeheer en CI/CD. De instapdrempel is laag, de tooling is stabiel en het ecosysteem groeit actief. Een belangrijke les uit dit project is dat de keuze voor een bestandsgebaseerde tool vroeg in het project moet worden gemaakt: hoe groter een bestaande Postman-collectie, hoe omvangrijker de migratie. Voor teams die opnieuw beginnen, is de aanbeveling om van bij de start te kiezen voor een bestand gebaseerd alternatief en de collectie samen met de broncode in versiebeheer op te nemen.



## Algemeen besluit

Dit onderzoek vertrok vanuit de vraag hoe AI-ondersteuning het ontwikkelproces van een cloudgebaseerde applicatie kan versnellen en verbeteren, en welke voordelen en risico's daarmee gepaard gaan. Aan de hand van het Lexis-project bij CERM werd deze vraag op twee sporen onderzocht: een theoretisch overzicht van de bouwstenen waaruit moderne AI bestaat (prompt engineering, het Model Context Protocol, skills en agents), en drie concrete implementaties binnen Lexis. Daarbij bleek een duidelijk onderscheid: MCP-servers zorgen voor de informatie die de agent nodig heeft, terwijl skills het gedrag sturen waarmee de agent die informatie verwerkt. Zo kon AI niet alleen theoretisch bekeken worden, maar ook in de praktijk worden getest.

AI kan op verschillende manieren worden ingezet om het ontwikkelproces te versnellen en te verbeteren, op voorwaarde dat de ontwikkelaar de controle behoudt. In het Lexis-project bleek AI het krachtigst wanneer ze fungeert als een verlengstuk van de ontwikkelaar: de ontwikkelaar formuleert het eindresultaat, het model bepaalt de weg ernaartoe en neemt repetitieve taken over. Daarmee verschuift de rol van de ontwikkelaar van het zelf produceren van elke regel code naar het sturen, beoordelen en valideren van wat de assistent voorstelt.

De drie praktijkimplementaties tonen elk een ander gezicht van de samenwerking met AI. Bij de OAuth2-implementatie diende AI bovendien als leermiddel: ze leerde de ontwikkelaar nieuwe concepten aan en bracht een gevoelig stuk authenticatielogica succesvol naar productie. De architectuurkeuzes daarbij waren doordacht en het resultaat draait vandaag bij echte klanten. Bij de Dataset Explorer kwam vooral de snelheidswinst aan bod: een interne toepassing die normaal veel meer ontwikkeltijd vraagt, werd op korte termijn opgeleverd en doet betrouwbaar dienst voor intern gebruik. Bij de migratie van Postman naar Bruno hielp AI ten slotte om in korte tijd de volledige API-collectie te controleren, aan te vullen en uit te breiden met geautomatiseerde testen, wat de testinfrastructuur binnen CERM merkbaar verbeterde.

Tegenover deze voordelen staan duidelijke risico's en beperkingen. De Dataset Explorer maakte duidelijk dat AI zonder gerichte sturing wel werkende, maar geen productieklare code oplevert: de toepassing functioneert intern, maar is in haar huidige vorm niet leverbaar aan klanten. In het algemeen neemt AI de verantwoordelijkheid voor architectuur, beveiliging en correctheid niet weg: de ontwikkelaar blijft verantwoordelijk en moet de output kritisch beoordelen.

Voor vervolgwerk liggen op verschillende vlakken kansen open. Lexis wordt door CERM verder ontwikkeld en uitgerold naar bijkomende klanten, waardoor de impact van de bestaande implementaties op grotere schaal zichtbaar zal worden.

Het Lexis-project illustreert dat AI vandaag al een volwaardige plaats kan innemen in een professioneel softwareontwikkelproces, op voorwaarde dat ze wordt ingezet in duidelijke processen, met heldere instructies en met een ontwikkelaar die de eindbeoordeling blijft maken. Niet AI als vervanging, maar als verlengstuk van de ontwikkelaar. Dat is de werkwijze die dit project heeft bewezen.



## Bibliografie

- Aikido Security BV. (2026). *Unified Security Platform from Code to Runtime*. [Software]. Geraadpleegd op 12 mei 2026, van <https://www.aikido.dev/>
- Anthropic. (2024, 25 november). *Introducing the Model Context Protocol*. Geraadpleegd op 9 april 2026, van <https://www.anthropic.com/news/model-context-protocol>
- Anthropic. (2025, 5 december). *Donating the Model Context Protocol and establishing the Agentic AI Foundation*. Geraadpleegd op 10 april 2026, van <https://www.anthropic.com/news/donating-the-model-context-protocol-and-establishing-of-the-agentic-ai-foundation>
- Anthropic. (2026a). *Claude Code*. (Versie 2.1.156) [Software]. Geraadpleegd op 2026, van <https://claude.com/product/claude-code>
- Anthropic. (2026b). *Claude Opus*. (versie 4.6) [Large language model]. <https://claude.ai/>
- Anthropic. (2026c). *Orchestrate teams of Claude Code sessions*. Claude Code Docs. Geraadpleegd op 15 mei 2026, van <https://code.claude.com/docs/en/agent-teams>
- Anthropic. (z.d.a). *Extend Claude with skills*. Claude Code Docs. Geraadpleegd op 17 april 2026, van <https://code.claude.com/docs/en/skills>
- Anthropic. (z.d.b). *Extended thinking*. Claude API Docs. Geraadpleegd op 22 mei 2026, van <https://docs.anthropic.com/en/docs/build-with-claude/extended-thinking>
- Anthropic. (z.d.c). *Prompt engineering for AI guide*. Geraadpleegd op 7 april 2026, van <https://platform.claude.com/docs/en/build-with-claude/prompt-engineering/overview>
- AnySphere. (2026, 13 april). *Cursor*. (Versie 3.1) [Software]. Geraadpleegd op 13 april 2026, van <https://www.cursor.com/>
- Atlassian. (2025). *Extend Atlassian into any AI assistant using MCP*. Geraadpleegd op 10 april 2026, van <https://www.atlassian.com/platform/remote-mcp-server>
- Becker, J., Rush, N., Barnes, E., & Rein, D. (2025, 12 juli). Measuring the impact of early-2025 AI on experienced Open-Source developer productivity. *arXiv (Cornell University)*. <https://doi.org/10.48550/arXiv.2507.09089>
- Bruno Software Inc. (2026, 26 mei). *Bruno*. (Versie 3.4.2) [Software]. <https://www.usebruno.com>
- CERM NV. (2026). *Home [Afbeelding]*. CERM. <https://www.cerm.net/>
- Europees Parlement. (2023, 25 juli). *AI-verordening: eerste regels voor artificiële intelligentie*. Geraadpleegd op 5 mei 2026, van <https://www.europarl.europa.eu/topics/nl/article/20230601STO93804/ai-verordening-eerste-regels-voor-artificiele-intelligentie>
- Europees Parlement. (2026, 10 maart). Bescherming van auteursrechtelijk beschermd werk en creatieve sector van de EU in AI tijdperk. Geraadpleegd op 5 mei 2026, van <https://www.europarl.europa.eu/news/nl/press-room/20260306IPR37511/bescherming-van-auteursrechtelijk-beschermd-werk-en-de-creatieve-sector-van-eu>
- Europese Commissie. (2024). *AI Act*. Shaping Europe's digital future. Geraadpleegd op 5 mei 2026, van <https://digital-strategy.ec.europa.eu/en/policies/regulatory-framework-ai>

- Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., & Chen, J. (2025, 12 februari). Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study. *ACM Transactions On Software Engineering And Methodology*, 34(8), 1–34. <https://doi.org/10.1145/3716848>
- Hardt, D. (2012). *The OAuth 2.0 Authorization Framework*. IETF. <https://doi.org/10.17487/rfc6749>
- Hong, K., Troynikov, A., & Huber, J. (2025, 14 juli). *Context Rot: How Increasing Input Tokens Impacts LLM Performance*. Chroma. Geraadpleegd op 19 mei 2026, van <https://research.trychroma.com/context-rot>
- Karpathy, A. (2025, 2 februari). *There's a new kind of coding I call "vibe coding", where you fully give in to the vibes*. [Post]. X. Geraadpleegd op 18 mei 2026, van <https://x.com/karpathy/status/1886192184808149383>
- Kong Inc. (2026, 22 mei). *Insomnia*. (Versie 12.6.0) [Software]. <https://insomnia.rest/>
- Krishna, A., Galinkin, E., Derczynski, L., & Martin, J. (2025, 31 januari). Importing Phantoms: Measuring LLM package hallucination vulnerabilities. *arXiv (Cornell University)*. <https://doi.org/10.48550/arXiv.2501.19012>
- Lodderstedt, T., Bradley, J., Labunets, A., & Fett, D. (2025). *Best current practice for OAuth 2.0 Security*. IETF. <https://doi.org/10.17487/rfc9700>
- Lovable. (2026). *Lovable*. (Versie 2.0) [Software]. Geraadpleegd op 9 april 2026, van <https://lovable.dev/>
- Microsoft. (2025a, 15 mei). *Backends for Frontends pattern*. Microsoft Learn. Geraadpleegd op 23 maart 2026, van <https://learn.microsoft.com/azure/architecture/patterns/backends-for-frontends>
- Microsoft. (2025b, 12 december). *Enable AI assistance with the Azure DevOps MCP Server*. Geraadpleegd op 9 april 2026, van <https://learn.microsoft.com/en-us/azure/devops/mcp-server/mcp-server-overview>
- Model Context Protocol. (2025, 11 november). *Specification - Model Context Protocol*. Geraadpleegd op 10 april 2026, van <https://modelcontextprotocol.io/specification/2025-11-25>
- Model Context Protocol. (z.d.). *Architecture overview - Model Context Protocol*. Geraadpleegd op 10 april 2026, van <https://modelcontextprotocol.io/docs/learn/architecture>
- OpenAI. (z.d.). *Tokenizer*. OpenAI Platform. Geraadpleegd op 1 juni 2026, van <https://platform.openai.com/tokenizer>
- OWASP. (2025, 14 mei). *Cross Site Request Forgery (CSRF)*. OWASP. Geraadpleegd op 23 maart 2026, van <https://owasp.org/www-community/attacks/csrf>
- Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023, 13 februari). The impact of AI on developer productivity: Evidence from GitHub Copilot. *arXiv (Cornell University)*. <https://doi.org/10.48550/arXiv.2302.06590>
- Plotly Technologies Inc. (2024). *Dash: A framework for building analytical web applications*. (Versie 2.18) [Python-software]. Geraadpleegd op 3 maart 2026, van <https://dash.plotly.com>

- Postman Inc. (2026). *Postman*. [Software]. Geraadpleegd op 27 mei 2026, van <https://www.postman.com/>
- Pydantic Services Inc. (2026, 23 januari). *Pydantic AI*. (Versie 1.47.0) [Python-software]. Geraadpleegd op 9 april 2026, van <https://ai.pydantic.dev>
- Rodriguez, M. (2026, 25 maart). *Updates to GitHub Copilot interaction data usage policy*. Geraadpleegd op 5 mei 2026, van <https://github.blog/news-insights/company-news/updates-to-github-copilot-interaction-data-usage-policy/>
- Sakimura, N., Bradley, J., & Agarwal, N. (2015). *Proof Key for Code Exchange by OAuth Public Clients*. IETF. <https://doi.org/10.17487/rfc7636>
- Scheire, L., Helsmoortel, H., Baert, J., Aerts, E., Kerkhofs, N., Berx, P., Van Peer, B., Beheydt, K., & Verhelst, M. (2026, 3 mei). Is Claude Code intussen beter dan OpenClaw? [Podcastsegment, 00:55:21]. In *Nerland maandoverzicht: Mei 2026*. [Podcastaflevering]. Geraadpleegd op 6 mei 2026, van <https://podcast.nerland.be/nerland-maandoverzicht-mei-2026/>
- Schulhoff, S., Ilie, M., Balepur, N., Kahadze, K., Liu, A., Si, C., Li, Y., Gupta, A., Han, H., Schulhoff, S., Dulepet, P., Vidyadhara, S., Ki, D., Agrawal, S., Pham, C., Kroiz, G., Li, F., Tao, H., Srivastava, A., Hevander, D., . . . Resnik, P. (2024, 6 juni). The Prompt Report: A systematic survey of prompt engineering techniques. *arXiv (Cornell University)*. <https://doi.org/10.48550/arXiv.2406.06608>
- Shen, J. H., & Tamkin, A. (2026, 28 januari). How AI impacts skill Formation. *arXiv (Cornell University)*. <https://doi.org/10.48550/arXiv.2601.20245>
- Shukla, S., Joshi, H., & Syed, R. (2025, 26 september). Security degradation in iterative AI code Generation: A Systematic analysis of the paradox. *arXiv (Cornell University)*, 1–8. <https://doi.org/10.48550/arXiv.2506.11022>
- the Linux Foundation. (2025, 19 september). *OpenAPI Specification*. (Versie 3.2.0). (H. Andrews, J. Whitlock, K. Etheridge, L. Mitchell, M. Gardiner, M. Quintero, M. Kistler, R. Handl, & V. Biret, Reds.) Geraadpleegd op 2 juni 2026, van <https://spec.openapis.org/oas/v3.2.0.html>
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2023, 10 januari). *Chain-of-Thought prompting elicits reasoning in large language models*. *arXiv.org*. <https://doi.org/10.48550/arXiv.2201.11903>





**campus Brugge Xaverianenstraat**

Xaverianenstraat 10  
050 30 51 00  
campus.brugge@vives.be

—

**campus Brugge Station**

Spoorwegstraat 12  
050 40 59 00  
campus.brugge@vives.be

—

**campus Kortrijk**

Doorniksesteenweg 145  
056 26 41 60  
campus.kortrijk@vives.be

—

**campus Oostende Station**

Lijndraaiersstraat 60  
059 56 90 00  
campus.oostende@vives.be

—

**campus Oostende VLOC**

Nieuwpoortsesteenweg 945c  
059 30 81 50  
campus.oostende@vives.be

—

**campus Roeselare**

Wilgenstraat 32  
051 23 23 30  
campus.roeselare@vives.be

—

**campus Torhout**

Sint-Jozefstraat 1  
050 23 10 30  
campus.torhout@vives.be

**www.vives.be**

