



UHASSELT

KNOWLEDGE IN ACTION



Maastricht University

Faculteit Wetenschappen **School voor Informatietechnologie**

master in de informatica

Masterthesis

GenAI ter ondersteuning van coderen en debuggen

Ceel Beekers

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Frank NEVEN

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.



UHASSELT

KNOWLEDGE IN ACTION

www.uhasselt.be

Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

2024
2025



Maastricht University

Faculteit Wetenschappen ***School voor Informatietechnologie***

master in de informatica

Masterthesis

GenAI ter ondersteuning van coderen en debuggen

Ceel Beckers

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Frank NEVEN

UNIVERSITEIT HASSELT

MASTERPROEF VOORGEDRAGEN TOT HET BEHALEN VAN DE
GRAAD VAN MASTER IN DE INFORMATICA

GenAI ter ondersteuning van coderen en debuggen

Auteur:

Beckers Ceel

Promotor:

Prof. dr. Frank Neven

Co-promotor:

Prof. dr. Frank Neven

Begeleider:

Prof. dr. Frank Neven

Academiejaar 2024-2025



Dankwoord

Allereerst wil ik mijn diepste dank uitspreken aan mijn promotor en begeleider, prof. dr. Frank Neven. Ik ben oprecht dankbaar voor zijn geduld en begrip wanneer dingen niet volgens plan gingen, evenals voor zijn uitgebreide expertise in het vakgebied. Daarnaast waardeer ik zijn niet-aflatende inzet voor het verbeteren van het informaticaonderwijs, waarbij hij zich inzet om zowel de kwaliteit als de toegankelijkheid ervan te verhogen.

Ik wil ook mijn collega, Robin Weynjes, bedanken voor zijn steun, professionaliteit en inzichtelijke opmerkingen tijdens onze bijeenkomsten met prof. dr. Frank Neven, die we samen hebben bijgewoond.

Samenvatting

Deze thesis richt zich op studenten die het opleidingsonderdeel *Imperatief Programmeren* volgen, het eerste programmeervak binnen de Academische Bachelor Informatica aan de Universiteit Hasselt. Het vak introduceert programmeren in Python en gaat uit van een volledige afwezigheid van voorkennis. *Imperatief Programmeren* is een essentieel fundament voor verdere opleidingsonderdelen.

Het onderzoek bestaat uit twee grote delen. Het eerste deel onderzoekt hoe Generative Artificial Intelligence (GenAI) kan ondersteunen bij het iteratief oplossen van oefeningen. Het tweede deel richt zich op debugging.

Voorafgaand aan het eigenlijke onderzoek volgt een toelichting over Large Language Models (LLM's), een vorm van kunstmatige intelligentie (AI) die natuurlijke taal begrijpt en genereert. Hoewel LLM's over krachtige capaciteiten beschikken, kunnen ze ook vertekeningen bevatten of foutieve informatie opleveren. GenAI, waar LLM's een onderdeel van vormen, genereert nieuwe inhoud en omvat een breder technologisch spectrum dan LLM's alleen.

Tijdens het onderzoek worden meerdere LLM's ingezet. In het eerste deel van het onderzoek wordt gebruik gemaakt van ChatGPT (specifiek het GPT-4o model), Microsoft Copilot (eveneens gebaseerd op GPT-4o) en Google Gemini. Initieel gebeurt dit met Gemini 1.5. Later, bij de lancering van Gemini 2.0 Flash (Experimental), wordt overgeschakeld naar Gemini 2.0. De experimenten met Gemini 1.5 worden vervolgens herhaald met Gemini 2.0 om te controleren of de oorspronkelijke bevindingen overeind blijven. In het tweede deel ligt de focus op GitHub Copilot dat ondersteuning biedt voor meerdere modellen. Bij aanvang van het onderzoek is echter enkel het GPT-4o model beschikbaar buiten de previewfase. Inmiddels zijn meerdere modellen buiten preview toegankelijk. Voor alle interacties met GitHub Copilot wordt gebruik gemaakt van de chatfunctionaliteit.

Deel 1

Het eerste deel van het onderzoek begint met een literatuurstudie. Daaruit blijkt onder meer dat correcte begeleiding essentieel is om het gebruik van GenAI in het onderwijs optimaal te laten renderen. Binnen deze context worden verschillende frameworks voor prompt engineering besproken. Het meest relevante daarvan is het *CLEAR*-framework [24], dat bestaat uit vijf componenten: *concise*, *logical*, *explicit*, *adaptive* en *reflective*. Elk van deze elementen draagt bij aan het opstellen van effectieve prompts. *Concise* richt zich op het verwijderen van overbodige informatie, terwijl *logical* staat voor een gestructureerde en logische opbouw van de prompt. *Explicit* benadrukt het belang van duidelijke en nauwkeurige instructies. *Adaptive* verwijst naar de mogelijkheid om de prompt aan te passen wanneer de situatie daarom vraagt, en *reflective* stimuleert het gebruik van feedback om de kwaliteit van prompts voortdurend te verbeteren. Het CLEAR-framework blijkt bijzonder waardevol tijdens de prompt engineering en wordt in beide onderzoeksdelen toegepast. Naast dit framework biedt ook Microsoft aanvullende richtlijnen ter ondersteuning van effectief promptgebruik.

Vervolgonderzoek toont aan dat ChatGPT in staat is om zinvolle feedback te geven die het

leerproces ondersteunt. Deze formatieve feedback helpt studenten om hun begrip te verdiepen, fouten te herkennen en hun oplossingen te verbeteren tijdens het oefenen. Daarentegen blijkt ChatGPT minder betrouwbaar wanneer het gaat om feedback die bedoeld is om een eindoplossing te beoordelen, ook wel summatieve feedback genoemd. Deze vorm van feedback dient om een definitief oordeel te vellen over de kwaliteit van een oplossing. Studenten ervaren de gegenereerde feedback van ChatGPT als waardevol voor het leren, ondanks dat die soms onjuistheden bevat. Een afzonderlijk onderzoek naar een zelfgetraind AI-model laat vergelijkbare resultaten zien: ook dit model helpt bij het leerproces, maar blijkt niet altijd accuraat genoeg om als beoordelingsinstrument te dienen. Daarom blijft het van belang dat studenten de gegenereerde feedback kritisch interpreteren en toetsen aan eigen inzicht.

Verder blijkt uit onderzoek dat GenAI-gebaseerde tutors het leerproces op verschillende manieren kunnen ondersteunen. Ze dragen bij aan betere academische prestaties, verhogen de motivatie en vergroten de betrokkenheid tijdens de les. Bovendien helpen ze studenten om het studiemateriaal beter te begrijpen. Daarnaast is een begeleidingsmodel ontwikkeld door *Seppe Kimps* aan de *Universiteit Hasselt*, met als doel studenten te ondersteunen in het effectief inzetten van GenAI. Een aangepaste versie van dit stappenplan is opgenomen in de studieleidraad van het vak *Imperatief Programmeren* en vormt het uitgangspunt voor het eerste deel van dit onderzoek.

Binnen dit vak maken studenten gebruik van het online leer- en programmeerplatform *Dodona* dat oefeningen aanbiedt en automatische feedback geeft op ingediende code. De datasets die voor dit onderzoek worden gebruikt, zijn gebaseerd op programmeeroefeningen uit Dodona. Het aangepaste stappenplan van Seppe Kimps, opgenomen in de studieleidraad, wordt weergegeven in Bijlage A. Daarin is stap C onderverdeeld in drie deelstappen: *Implementeer een basisoplossing*, *Test met eenvoudige voorbeelden* en *Itereer en breid uit*. Deze structuur vormt het uitgangspunt van het eerste onderzoeksdeel. Binnen dit kader worden twee fasen onderscheiden in het gebruik van prompts. In de eerste fase wordt de LLM gevraagd om de opdracht op te splitsen in kleinere deelopgaven. In de tweede fase genereert de LLM testgevallen voor deze deelopgaven, waarmee studenten hun oplossingen kunnen verfijnen en valideren.

De volgende onderzoeksvragen worden geformuleerd:

RQ1: Kan GenAI een oefening vereenvoudigen, zodat de student deze stapsgewijs kan oplossen?

RQ2: Kan GenAI testfuncties genereren voor deze vereenvoudigde opgaven?

RQ3: Kan GenAI tests genereren voor deze vereenvoudigde opgaven in het geval er geen functies aanwezig zijn?

Het eigenlijke onderzoek begint met prompt engineering. Voor het opstellen van het stappenplan is één specifieke programmeeropgave geselecteerd, namelijk een opgave die meetelt voor punten. In Dodona zijn er per hoofdstuk meerdere opgaven beschikbaar, waarvan er vier punten opleveren. Voor dit onderzoek is de derde van deze vier gekozen als basis voor het prompt engineering-gedeelte.

In de eerste fase wordt onderzocht hoe studenten deze opgave opdelen in deelstappen. Vervolgens wordt nagegaan hoe de verschillende LLM's — ChatGPT, Gemini en Copilot — dit aanpakken. Opvallend is dat elk model één oplossingsstrategie voorstelt die niet expliciet door studenten wordt toegepast. Tegelijkertijd herkennen de LLM's — samen — alle strategieën die wél door studenten gebruikt worden. ChatGPT en Copilot genereren vrijwel identieke strategieën, wat te verklaren is door hun gedeelde onderliggende GPT-architectuur. Gemini komt in dit geval met een aanvullende strategie die niet door de andere modellen wordt voorgesteld. Aangezien niet met zekerheid kan worden vastgesteld welke strategieën daadwerkelijk door studenten zijn gevolgd, leveren deze observaties vooral inzichten op, zonder dat er harde conclusies kunnen worden getrokken.

Ook het genereren van tests wordt onderzocht aan de hand van een realistische situatie. Eerst worden alle mogelijke randgevallen voor de gekozen opgave geïdentificeerd, in totaal 25. Ver-

volgens wordt nagegaan hoeveel van deze gevallen gedetecteerd worden door de bestaande testgevallen van Dodona en door de verschillende LLM's. Dodona dekt 14 randgevallen, ChatGPT 15, Copilot 13 en Gemini 18. Zowel Dodona als de LLM's ontdekken dus testgevallen die door de andere niet worden opgemerkt. Wanneer de testgevallen van Dodona aan de LLM's worden aangeboden, breidt elk model de dekking verder uit: ChatGPT detecteert dan 20 randgevallen, Copilot 17 en Gemini 18. Geen enkel model weet echter alle 25 randgevallen te identificeren. Deze resultaten suggereren dat LLM's waardevolle aanvullingen kunnen bieden op bestaande testsets, maar dat geen enkel systeem volledige dekking garandeert.

De prompt engineering voor het genereren van tests maakt onderscheid tussen opgaven mét en zonder functies. Bij aanwezigheid van een functie worden deze opgeroepen in de tests; bij afwezigheid wordt gekozen voor input/output-paren. Voor Gemini wordt een licht afwijkende prompt gebruikt ten opzichte van ChatGPT en Copilot. In gevallen zonder functie wordt een gemeenschappelijke prompt toegepast.

Voor de evaluatie van zowel het genereren van stappen als het genereren van testcases is een puntsgewijs beoordelingssysteem ontwikkeld. Beide fases zijn beoordeeld op basis van zes hoofdcategorieën: *correctheid*, *granulariteit*, *kwaliteit*, *efficiëntie*, *flexibiliteit* en *integriteit*. Deze categorieën zijn verder onderverdeeld in specifieke beoordelingsonderdelen, waarbij elk onderdeel een maximale score kan behalen, afhankelijk van het relatieve belang binnen de evaluatie.

Bij het genereren van stappen bedraagt de totaalscore maximaal 75 punten. De categorie *correctheid* (maximaal 20 punten) omvat de onderdelen completeness en soliditeit, waarbij respectievelijk de aanwezigheid en relevantie van noodzakelijke stappen worden beoordeeld. *Granulariteit* (15 punten) bekijkt of er sprake is van overbodige of juist ontbrekende details. *Kwaliteit* (10 punten) richt zich op de begrijpelijkheid, duidelijkheid en complexiteit van de geformuleerde stappen. De categorieën *efficiëntie* en *flexibiliteit* worden elk beoordeeld op één criterium, met een maximum van 5 punten per onderdeel, en meten respectievelijk de optimaliteit en de aanpasbaarheid van het stappenplan. Tot slot wordt *integriteit* (20 punten) beoordeeld op de mate waarin het model antwoorden direct aan de student voorzegt, hetgeen als ongewenst wordt beschouwd.

Voor het genereren van testcases is de beoordeling uitgebreider, met een maximale totaalscore van 150 punten. De categorie *correctheid* (45 punten) bestaat uit de onderdelen foutloze output, juiste oplossingen en consistentie, waarbij respectievelijk de afwezigheid van syntactische en logische fouten, de juistheid van de gegenereerde uitkomsten, en de deterministische uitvoer worden geëvalueerd. *Granulariteit* (25 punten) beoordeelt het detailniveau van de testcases, de variatie in moeilijkheidsgraad, en de aanwezigheid van zowel minimale als extra edge cases. De categorie *kwaliteit* (15 punten) omvat de leesbaarheid, relevantie en documentatie van testcases. *Efficiëntie* (10 punten) wordt gemeten aan de hand van de tijdsduur en het resourcerendement van de uitvoering. *Flexibiliteit* (15 punten) betreft de mate waarin testcases eenvoudig aanpasbaar, uitbreidbaar en compatibel zijn met verschillende contexten en systemen. De categorie *integriteit*, met een relatief hoog gewicht van 40 punten, beoordeelt opnieuw of het model testcases aanbiedt die neigen naar directe beantwoording van de opgave, wat als ondermijnend wordt gezien voor het leerproces.

De kwantitatieve analyse van zowel de gegenereerde stappen als de testcases toont aan dat de prestaties van de drie geteste large language models (LLM's) – Copilot, ChatGPT en Gemini – relatief dicht bij elkaar liggen, met ChatGPT als algemeen best presterende model. Bij de evaluatie van het genereren van stappen behaalde ChatGPT de hoogste totaalscore (74,0/75), gevolgd door Gemini (72,3/75) en Copilot (70,3/75). De verschillen zijn het meest uitgesproken binnen de categorie *correctheid*, waar Copilot lager scoorde dan de andere modellen, vooral op het onderdeel soliditeit. Binnen *granulariteit* viel op dat Copilot soms te weinig stappen genereerde, terwijl ChatGPT en Gemini vaker extra stappen toevoegden. Op het vlak van *kwaliteit*, *efficiëntie*, *flexibiliteit* en *integriteit* bleven de scores onderling zeer vergelijkbaar, met voor alle drie de modellen een perfecte score op zowel *integriteit* als *flexibiliteit*.

Ook bij de beoordeling van het genereren van testcases behaalde ChatGPT de hoogste totaal-

score (136,0/150), gevolgd door Gemini (132,4/150) en Copilot (128,7/150). In de categorie *correctheid* scoorden alle modellen goed; opvallend was dat Copilot iets beter presteerde op het onderdeel: juiste oplossingen, terwijl Gemini daar wat achterbleef. Binnen de categorie *granulariteit* behaalden ChatGPT en Gemini hogere scores dan Copilot. Gemini genereerde over het algemeen de meeste en meest gevarieerde testgevallen, terwijl ChatGPT consistent en vollediger scoorde over de verschillende beoordelingscriteria heen. Wat *kwaliteit* betreft had Gemini een lichte voorsprong (14,9/15), voornamelijk dankzij een hogere leesbaarheidsscore. ChatGPT onderscheidde zich in de categorie *integriteit* met een perfecte score (40/40), terwijl Copilot daar de laagste score behaalde (34,3/40).

Deze scores bevestigen de subjectieve indrukken die tijdens het evaluatieproces ontstonden: ChatGPT werd intuïtief als het krachtigste model ervaren, met Gemini als nauwe tweede en Copilot als derde. Hoewel de stilistische output van Copilot vaak sterk leek op die van ChatGPT, lagen de prestaties inhoudelijk dicht bij die van Gemini.

Vervolgens vindt een kwalitatieve analyse plaats op een oefening met tests en een oefening met input/output-paren. De eindconclusie van dit eerste deel van het onderzoek luidt dat GenAI een oefening kan vereenvoudigen, zodat studenten deze stapsgewijs oplossen (RQ1). De effectiviteit verschilt echter per oefening, en controle blijft noodzakelijk. In sommige gevallen maakt een LLM eenvoudige opgavenodeloos complex. Daarnaast blijkt GenAI in staat testfuncties te genereren voor de vereenvoudigde oefeningen (RQ2). De gegenereerde oplossingen vereisen echter grondige controle. Foutieve output komt frequent voor, en niet alle randgevallen worden automatisch meegenomen. Over het algemeen volstaan de gegenereerde tests om de deelopgaven te valideren. Ook zonder functies kan GenAI tests genereren voor deze vereenvoudigde opgaven (RQ3), mits de gegenereerde testgevallen en oplossingen zorgvuldig gecontroleerd worden.

Samenvattend laat dit eerste deel van het onderzoek zien dat GenAI een nuttig hulpmiddel kan zijn bij het begeleiden van studenten in het oplossen van programmeeroefeningen, mits de gegenereerde output zorgvuldig wordt gecontroleerd. De technologie biedt potentieel voor ondersteuning bij zowel het vereenvoudigen van opgaven als het genereren van testgevallen, maar vereist nog menselijke validatie om betrouwbaar ingezet te worden in een onderwijscontext.

Deel 2

Het tweede deel van het onderzoek richt zich op debuggen. Eerst volgt een literatuurstudie. Hieruit blijkt dat ontwikkelaars tijdens het debuggen vrijwel uitsluitend gebruikmaken van de debugtools in de IDE (Integrated Development Environment). Deze bevinding draagt bij aan de keuze voor GitHub Copilot in Visual Studio Code (VS Code). Stap voor stap door de code gaan, blijkt daarbij de meest behulpzame functionaliteit volgens eerder onderzoek. Ook blijkt dat programmeurs regelmatig foutmeldingen kopiëren en plakken in een zoekmachine om snel tot een oplossing te komen of de fout beter te begrijpen. Verder toont de literatuurstudie aan dat LLM's beter presteren bij het debuggen wanneer ze toegang hebben tot de volledige broncode. Daarom valt de keuze op GitHub Copilot in plaats van bijvoorbeeld ChatGPT. De literatuurstudie levert meerdere debugstrategieën op. Slechts voor zes daarvan zijn geschikte scenario's te formuleren: *hypothesetesten*, *voorwaarts redeneren*, *achterwaarts redeneren*, *raadplegen van externe bronnen*, *analyseren van foutmeldingen* en *simplificatie*. Daarom richt het verkennend onderzoek zich uitsluitend op deze zes strategieën.

Bij de strategie *hypothesetesten* formuleert de student een hypothese over de oorzaak van een bug en legt deze voor aan GitHub Copilot. Copilot bevestigt of verwerpt de hypothese, of helpt bij het testen ervan. *Voorwaarts redeneren* houdt in dat Copilot suggesties doet voor het plaatsen van breakpoints, conditionele breakpoints of printstatements. Hierdoor kan een specifieke variabele onderzocht worden. Bij *achterwaarts redeneren* stelt Copilot, op basis van een foutmelding, geschikte breakpoints voor. In het geval van *foutmeldingen* wordt een

foutmelding aan Copilot doorgegeven met het verzoek om uitleg. Bij het *raadplegen van externe bronnen* fungeert Copilot zelf als hulpbron. Zo kan bijvoorbeeld gevraagd worden hoe de `len()`-functie in Python werkt of er een functie bestaat die de lengte van een string bepaalt. Copilot verwijst dan naar de juiste functie. Bij *simplificatie* verwijdert Copilot overbodige code, zodat de student zich beter kan richten op de kern van het probleem of eenvoudiger online naar oplossingen kan zoeken.

Voor het verkennend onderzoek worden eerst enkele datasets opgebouwd. De eerste dataset richt zich op syntaxfouten en is gebaseerd op de derde opdracht van de vier binnen dit vak. Dit betreft dezelfde opdracht als in het eerste deel van het onderzoek, aangezien deze het meest representatief lijkt voor het niveau van de cursus. De tweede dataset is op vergelijkbare wijze samengesteld, maar focust op logische fouten. De derde dataset dient uitsluitend voor het analyseren van externe resources. Deze dataset bevat alle onderdelen die vermeld staan in het boek *De Programmeursleerling* [39]. Voor het verkennend onderzoek wordt hieruit een selectie gemaakt van tien onderdelen (functies, methoden en constructies).

Elke strategie in het verkennend onderzoek start met een eenvoudige maatstaf. Deze wordt op basis van de bevindingen verfijnd voor het diepergaand onderzoek. Tijdens de verkennende fase komen meerdere inzichten naar voren. Zo blijkt het bijzonder moeilijk om een geschikte dataset te vinden of te ontwikkelen voor de strategie gericht op codeversimpeling. Daarnaast vormt het opstellen van een betrouwbare meetmethode een aanzienlijke uitdaging. Het risico op onvoldoende onderbouwde resultaten wordt hierdoor te groot, wat leidt tot het schrappen van deze strategie in het vervolgonderzoek. Op deze manier komt er meer tijd en aandacht vrij voor de overblijvende strategieën.

Uit de resultaten blijkt tevens dat de strategie rond achterwaarts redeneren minder effectief is dan verwacht. Aangezien studenten meestal de exacte locatie van een fout kunnen afleiden uit de foutmelding, voegt herhaling hiervan door een LLM weinig waarde toe. Daarom wordt ook deze strategie uitgesloten van het verdere onderzoek.

De overige strategieën tonen daarentegen wel potentieel. Op basis van de verkregen inzichten worden de volgende onderzoeksvragen geformuleerd:

RQ4: Kan een LLM worden ingezet om foutboodschappen uit te leggen?

RQ5a: Kan een LLM worden gebruikt om een *functie*, *methode* of *constructie* toe te lichten?

RQ5b: Kan een LLM worden gebruikt om een *functie*, *methode* of *constructie* aan te bevelen op basis van een beschrijving?

RQ6: Kan een LLM worden ingezet om een hypothese over niet-werkende code te valideren?

RQ7: Kan een LLM suggesties doen voor relevante breakpoints?

Voor elke strategie wordt een specifieke puntenverdeling opgesteld. Afhankelijk van de strategie bestaat deze verdeling uit vier of vijf hoofdcriteria: *accuraatheid*, *volledigheid*, *duidelijkheid*, *structuur* en indien van toepassing: *integriteit*. De weging van deze criteria verschilt per strategie. De puntenverdeling dient uitsluitend om de prestaties van GitHub Copilot per strategie te evalueren, niet om te bepalen welke strategie het beste resultaat oplevert.

Accuraatheid beoordeelt in welke mate feitelijke fouten of misinformatie worden vermeden. *Duidelijkheid* slaat op de helderheid van de gegeven uitleg. *Volledigheid* bekijkt in hoeverre de uitleg alle noodzakelijke elementen bevat. *Structuur* evalueert hoe logisch en overzichtelijk de uitleg is opgebouwd. Bij het criterium *integriteit* wordt nagegaan of GitHub Copilot niet te veel prijsgeeft en dus de oplossing onbedoeld verklapt. *Integriteit* is niet van toepassing op *foutboodschappen*, aangezien hier geen code beschikbaar is en er dus ook geen oplossing kan worden voorspeld.

Bij de categorie *externe bronnen* worden twee richtingen onderzocht: van functie naar uitleg (RQ5a) en van uitleg naar functie (RQ5b). Voor het *testen van hypothesen* worden zowel correcte als foute hypothesen geanalyseerd. Correcte hypothesen moeten herkend en bevestigd

worden, terwijl foute hypothesen als dusdanig moeten worden geïdentificeerd. In de verkennende studie bedraagt de verdeling tussen juiste en foute hypothesen respectievelijk 25% en 75%. In het diepgaandere onderzoek wijzigt deze verhouding: foutieve hypothesen tellen nu voor 55% mee, wat beter aansluit bij de gebruikte puntenverdeling. Juist en fout worden bovendien afzonderlijk beschouwd, zodat een sterke prestatie in de ene categorie eventuele tekortkomingen in de andere niet verbergt.

Ook in dit onderzoek worden verschillende datasets gebruikt. De dataset voor syntaxfouten bestaat uit dataset 1 en wordt aangevuld met één oefening per hoofdstuk uit Dodona. Analooeg bestaat de dataset voor logische fouten uit dataset 2, eveneens aangevuld met oefeningen uit de hoofdstukken. Voor de analyse van externe bronnen wordt een derde dataset gebruikt, samengesteld uit alle onderdelen vermeld in het boek *De Programmeursleerling* [39].

GitHub Copilot behaalt bij de interpretatie van *foutboodschappen* opvallend sterke resultaten. In alle gevallen wordt de maximale score toegekend voor *accuraatheid*, terwijl *duidelijkheid* en *volledigheid* telkens zeer hoog scoren (beide 19,6 op 20). Ook *structuur* wordt consistent goed beoordeeld (9,8 op 10). De score voor *integriteit* toont iets meer spreiding: in dataset 1 bedraagt het gemiddelde 22,222 op 25, en in de hoofdstukken 21,875 op 25. Bij het analyseren van *externe bronnen* blijft het beeld positief. Voor het uitleggen van functies worden maximale scores behaald voor *duidelijkheid* en *structuur*. *Accuraatheid* bereikt 34,13 op 35, en *volledigheid* 23,87 op 25. Aanbevelingen van functies leveren nagenoeg perfecte scores op voor *duidelijkheid* (24,96 op 25) en *structuur* (15 op 15) met een licht lagere *accuraatheid* (34,7 op 35) en een uitstekende *volledigheid* (24,85 op 25). In deze context is *integriteit* niet beoordeeld.

Uit de *hypothesetests* blijkt duidelijk dat GitHub Copilot beter presteert bij syntaxfouten dan bij logische fouten. Voor syntaxfouten bedraagt de totaalscore 95,74 op 100, terwijl deze bij logische fouten daalt naar 89 op 100. In beide gevallen ontvangen *structuur* en *volledigheid* de maximumscore. Ook *duidelijkheid* krijgt bij logische fouten de hoogste score, terwijl dit bij syntaxfouten 19,63 op 20 bedraagt, wat eveneens zeer hoog is. Voor syntaxfouten ligt de *accuraatheid* op 22,78 op 25 en de *integriteit* op 23,33 op 25. Bij logische fouten dalen deze scores respectievelijk naar 20 op 25 en 19 op 25. In het geval van syntaxfouten wordt in minstens 75% van de gevallen de maximumscore toegekend voor alle categorieën, zowel bij correcte als incorrecte hypothesen. In het geval van logische fouten valt op dat — specifiek bij de foutieve hypothesen — bijna de helft niet de maximale score behaalt voor *accuraatheid*. Ook voor *integriteit* wordt slechts in 50% van de gevallen de hoogste score toegekend binnen de dataset gebaseerd op oefeningen uit de Dodona-hoofdstukken. In dataset 2 behaalt 30% van de foutieve hypothesen evenmin de maximale score voor *integriteit*.

Voorwaarts redeneren behaalt hoge scores, met een totaalscore van 96,5 op 100 en maximale scores voor *volledigheid*, *structuur* en *integriteit*. *Duidelijkheid* nadert het maximum, terwijl de *accuraatheid* iets lager ligt (21,83 op 25).

Op basis van deze resultaten blijkt dat een Large Language Model zoals GitHub Copilot doeltreffend inzetbaar is voor het verklaren van foutboodschappen (RQ4). Ook bij het toelichten en aanbevelen van functies, methoden of constructies levert Copilot sterke prestaties (RQ5a & RQ5b). Daarmee vormt het systeem een waardevolle aanvulling op het programmeeronderwijs, zolang studenten de leerstof eerst via de cursus verwerken. Voor het valideren van hypothesen over niet-werkende code blijft de inzetbaarheid echter beperkt (RQ6). Syntactische fouten worden deels herkend, maar het toetsen van hypothesen bij logische fouten blijkt momenteel niet haalbaar. Alternatieve prompts zouden hierin mogelijk verbetering brengen, al zijn die binnen dit onderzoek niet geïdentificeerd. De voorlopige conclusie luidt dan ook dat deze toepassing nog onvoldoende betrouwbaar is.

De resultaten met betrekking tot het doen van suggesties voor relevante breakpoints (RQ7) bij voorwaarts redeneren schetsen een gemengd, maar overwegend positief beeld. Copilot genereert regelmatig bruikbare voorstellen, maar slaagt er niet altijd in om effectieve breakpoints aan te wijzen die daadwerkelijk tot de bug leiden. Aangezien dit de student niet schaadt, maar ook niet actief ondersteunt, blijft gebruik voor dit doel verantwoord, mits duidelijk wordt gemaakt

dat de suggesties niet steeds effectief zijn.

Samenvattend laat dit tweede deel van het onderzoek zien dat GitHub Copilot een bruikbaar hulpmiddel is voor het programmeeronderwijs, maar met verschillende mate van succes afhankelijk van de taak. Het werkt goed voor het uitleggen van foutboodschappen en het aanbevelen en uitleggen van functies, mits studenten de leerstof eerst via de cursus verwerken. Voor het aanbevelen van breakpoints zijn de resultaten overwegend positief maar wisselend. Het systeem faalt echter bij het geven van feedback op hypothesen over foutieve code. De technologie toont dus een gedifferentieerd inzetpotentieel binnen het programmeeronderwijs.

Inhoudsopgave

1	Inleiding	12
2	Wat zijn LLM's	14
2.1	LLM's en GenAI	14
2.2	Ondersteunde modellen	14
I	Iteratieve aanpak oefeningen	16
3	Overzicht onderzoek naar gebruik van LLMs	17
3.1	Correct gebruik van LLM's	17
3.2	Prompts	18
3.3	Feedback	18
3.4	Tutor	19
3.5	Framework	19
4	Stapsgewijze en iteratieve aanpak	21
4.1	Achtergrond	21
4.2	Methode van aanpak	22
4.2.1	Prompts voor het genereren van deelstappen	22
4.2.2	Prompts voor het genereren van testen	25
4.2.3	Score	30
4.3	Resultaten	33
4.3.1	Kwantitatieve analyse	33
4.3.2	Resultaten kwantitatieve analyse	40
4.3.3	Kwalitatieve analyse	41
4.4	Conclusie	43
4.5	Toekomstig werk en mogelijke verbeteringen	44
II	Debuggen	45
5	Overzicht onderzoek naar debuggen	46
5.1	Debug tools	46
5.2	Debuggen met LLM's	46
5.3	Debugstrategieën	47
6	Debuggen met LLM's	50
6.1	Achtergrond	50
6.2	Methode van aanpak verkennende studie	50
6.2.1	Datasets	51
6.2.2	Verkennende studie foutboodschappen	52
6.2.3	Verkennende studie achterwaarts redeneren	53

6.2.4	Verkennde studie voorwaarts redeneren	53
6.2.5	Verkennde studie externe bronnen	53
6.2.6	Verkennde studie hypothesetest	54
6.2.7	Verkennde studie simplificatie	54
6.3	Resultaten verkennde studie	55
6.3.1	Verkennde studie foutboodschappen	55
6.3.2	Verkennde studie achterwaarts redeneren	56
6.3.3	Verkennde studie voorwaarts redeneren	57
6.3.4	Verkennde studie externe bronnen	57
6.3.5	Verkennde studie hypothesetest	58
6.4	Conclusie verkennde studie	58
6.4.1	Verkennde studie foutboodschappen	58
6.4.2	Verkennde studie achterwaarts redeneren	58
6.4.3	Verkennde studie voorwaarts redeneren	59
6.4.4	Verkennde studie externe bronnen	59
6.4.5	Verkennde studie hypothesetest	59
6.5	Methode van aanpak	59
6.5.1	Foutboodschappen	59
6.5.2	Externe bronnen	61
6.5.3	Hypothesetest	63
6.5.4	Voorwaarts redeneren	64
6.6	Resultaten	66
6.6.1	Foutboodschappen	67
6.6.2	Externe bronnen	68
6.6.3	Hypothesetest	70
6.6.4	Voorwaarts redeneren	72
6.7	Conclusie	72
6.8	Toekomstig werk en mogelijke verbeteringen	73
7	Conclusies	74
A	Stappenplan Seppe Kimps Leidraad	80
B	Opgave prompt engineering	82
C	Kwantitatieve analyse tests	84
C.1	Opgave kwantitatieve analyse test	84
C.2	Eerste prompt Gemini	84
C.3	Antwoord Gemini eerste prompt	85
C.4	Tweede prompt Gemini	86
C.5	Antwoord Gemini tweede prompt	86
C.6	Output test 1	88
C.7	Output test 2	88
C.8	Output test 3	89
D	Kwantitatieve analyse input/output-paren	90
D.1	Opgave kwantitatieve analyse input/output-paren	90
D.2	Eerste prompt ChatGPT	91
D.3	Antwoord ChatGPT eerste prompt	91
D.4	Tweede prompt ChatGPT	92
D.5	Antwoord ChatGPT tweede prompt	92

Hoofdstuk 1

Inleiding

Generative Artificial Intelligence (GenAI) is een technologie die de afgelopen jaren sterk in opkomst is. Tools zoals *ChatGPT*, *Microsoft Copilot* en *Google Gemini* tonen aan hoe krachtig en veelzijdig deze technologie is. Ze openen de deur naar nieuwe mogelijkheden, maar roepen tegelijkertijd vragen op over het verantwoord gebruik ervan. Zo is het belangrijk om te voorkomen dat studenten GenAI gebruiken om fraude te plegen. Tegelijkertijd zou het onwenselijk zijn om dergelijke tools volledig te verbieden, gezien het enorme potentieel om het leerproces te ondersteunen. Een interessant voorbeeld is het inzetten van GenAI als persoonlijke tutor, waarmee studenten gepersonaliseerde feedback kunnen ontvangen en hun leerervaring kunnen verrijken.

Deze thesis richt zich op de specifieke doelgroep van studenten die het opleidingsonderdeel *Imperatief Programmeren* volgen. Dit vak is het eerste programmeervak binnen de Academische Bachelor Informatica aan de Universiteit Hasselt. Het biedt een introductie tot programmeren in Python en gaat ervan uit dat studenten geen voorkennis hebben. Het vak speelt dus een cruciale rol in het leggen van de basis voor de verdere opleiding.

De centrale vraagstelling van deze thesis is: “*Hoe kunnen LLM’s¹ worden ingezet binnen het opleidingsonderdeel Imperatief programmeren ter ondersteuning van het leerproces? Wat zegt de wetenschappelijke literatuur hierover?*”. Het uiteindelijke doel is om het vak te verbeteren door de slaagkansen van studenten te vergroten, zonder de werklast van docenten en assistenten te verhogen.

Om deze doelstelling te verwezenlijken, is eerst een literatuurstudie uitgevoerd. Het doel hiervan was om inzicht te krijgen in eerdere onderzoeken en toepassingen van LLM’s in het onderwijs. Op basis van deze literatuur zijn de volgende onderzoeksvragen geformuleerd:

Onderzoeksvragen deel 1

- RQ1: Kan GenAI een oefening vereenvoudigen, zodat de student deze stapsgewijs kan oplossen?
- RQ2: Kan GenAI testfuncties genereren voor deze vereenvoudigde opgaven?
- RQ3: Kan GenAI tests genereren voor deze vereenvoudigde opgaven in het geval er geen functies aanwezig zijn?

Deze vragen sluiten nauw aan bij de praktijk. *Prof. dr. Frank Neven*, docent van het vak *Imperatief Programmeren*, signaleert dat studenten vaak moeite ondervinden met het starten van een oefening. Door LLM’s in te zetten om opgaven te vereenvoudigen, ontstaat een toegankelijker instap. Studenten krijgen eerst een vereenvoudigde versie aangeboden, die later kan worden uitgebreid naar de volledige oefening. Een geleidelijke opbouw bevordert inzicht en verlaagt de instapdrempel. Voor de hoofdopgave worden doorgaans specifieke testgevallen

¹Large Language Models

voorzien, die niet noodzakelijk geschikt zijn voor deelopgaven. Daardoor blijven ook RQ2 en RQ3 relevant.

In het tweede deel van dit onderzoek wordt opnieuw een literatuurstudie uitgevoerd. Deze studie heeft als doel inzicht te verkrijgen in bestaande kennis over debuggingstrategieën, in combinatie met de vraag: “*Op welke manier kunnen LLM’s het debuggingproces ondersteunen?*”.

Aansluitend wordt een verkennend onderzoek uitgevoerd om mogelijke toepassingen van LLM’s binnen het debuggingproces te identificeren. Dit onderzoek biedt inzicht in welke aspecten geschikt zijn voor verder onderzoek, welke minder relevant zijn, en welke buiten de reikwijdte van dit onderzoek vallen. Op basis van deze verkenning worden de onderzoeksvragen voor het tweede deel van de studie geformuleerd:

Onderzoeksvragen deel 2

RQ4: Kan een LLM worden ingezet om foutboodschappen uit te leggen?

RQ5a: Kan een LLM worden gebruikt om een *functie*, *methode* of *constructie* toe te lichten?

RQ5b: Kan een LLM worden gebruikt om een *functie*, *methode* of *constructie* aan te bevelen op basis van een beschrijving?

RQ6: Kan een LLM worden ingezet om een hypothese over niet-werkende code te valideren?

RQ7: Kan een LLM suggesties doen voor relevante breakpoints?

Ook binnen het domein van debugging signaleert de docent moeilijkheden bij studenten. Op basis van gesprekken met *prof. dr. Frank Neven* blijkt dat studenten regelmatig moeite hebben met het analyseren en oplossen van fouten in hun code. In het verkennend onderzoek wordt nagegaan welke toepassingen van LLM’s effectief blijken in het bieden van ondersteuning, en welke minder relevant zijn.

Bovendien blijkt uit observaties tijdens de opleiding dat studenten reeds spontaan gebruikmaken van LLM’s bij debuggingtaken, zoals het verklaren van foutmeldingen of het opzoeken van geschikte functies.² Deze praktijken worden vaak toegepast zonder duidelijk inzicht in de betrouwbaarheid ervan, wat het belang van verder onderzoek onderstreept.

²Dit komt uit persoonlijke ervaringen tijdens de opleiding.

Hoofdstuk 2

Wat zijn LLM's

2.1 LLM's en GenAI

Het begrip LLM staat voor “Large Language Model”. Een LLM is een geavanceerd systeem voor kunstmatige intelligentie (AI) dat in staat is natuurlijke taal te begrijpen en te genereren [4]. Voor het ontwikkelen van een LLM worden diverse machine learning-technieken toegepast, gecombineerd met deep learning-architecturen zoals transformers [4]. Voorbeelden van geavanceerde LLM's zijn Generative Pretrained Transformers (GPT) en Bidirectional Encoder Representations from Transformers (BERT) [4].

LLM's bieden diverse voordelen. Ze begrijpen genuanceerde relaties tussen woorden en kunnen natuurlijke taal genereren die contextueel relevant is in verschillende formaten [4]. Omdat LLM's zijn getraind op meertalige datasets, kunnen ze teksten vertalen met hoge nauwkeurigheid. Hierbij houden ze rekening met taalgebonden nuances zoals uitdrukkingen, spreekwoorden en andere taalspecifieke eigenschappen, waardoor vertalingen natuurlijk aanvoelen [4].

Toch kennen LLM's ook beperkingen. Voorbeelden hiervan zijn de hoge rekenkracht en energie die nodig zijn, ethische vraagstukken, en beperkingen in het begrijpen van complexe of nieuwe contexten [4]. Het trainen van een LLM heeft een aanzienlijke impact op het milieu. Hoe groter het model, hoe meer geheugen en elektriciteit er nodig is. De benodigde elektriciteit, de productie en het transport van de hardware genereren aanzienlijke CO₂-uitstoot.

Bovendien kunnen biases in de trainingsdata leiden tot versterking van vooroordelen tegen bepaalde groepen mensen [4]. Een LLM kan ook misleidende informatie genereren, die bijvoorbeeld kan worden gebruikt voor phishingaanvallen of het verspreiden van nepnieuws [4]. Daarnaast hebben LLM's vaak moeite met het begrijpen van nieuwe of onbekende contexten [4].

Generative AI (GenAI), waarvan LLM's een onderdeel zijn, is een vorm van AI die in staat is nieuwe inhoud te creëren, zoals tekst, afbeeldingen, video's of muziek [20]. Hoewel GenAI vaak synoniem wordt gebruikt met LLM's, omvat het een breder scala aan technologieën. LLM's worden doorgaans ingezet voor het genereren van tekst en vallen daarom onder de bredere categorie van GenAI.

2.2 Ondersteunde modellen

In deze thesis worden de volgende modellen geanalyseerd: ChatGPT (4o), Google Gemini 2.0 en Microsoft Copilot. ChatGPT wordt beschouwd vanwege de populariteit, met een marktaandeel van 54,96% in het LLM-gebruik [6]. De keuze voor versie 4o is gemaakt omdat deze op het moment van schrijven het meest geavanceerde gratis beschikbare model was.

Google Gemini en Microsoft Copilot worden opgenomen vanwege hun brede toegankelijkheid. Gemini is gekoppeld aan Googleaccounts, terwijl Copilot gebruikt kan worden zonder in te loggen (getest op 26 december 2024). Gebruikers van Windows hebben vaak al een Microsoft-account, aangezien dit een vereiste is voor het gebruik van Windows-systemen [27]. Voor het gebruik van Google Gemini is een Googleaccount nodig, wat voor veel mensen geen obstakel zal zijn.

Daarnaast richt het tweede deel van de thesis zich op GitHub Copilot omdat studenten gratis toegang hebben via het *GitHub Student Developer Pack*.

Eigenschappen ChatGPT 4o

ChatGPT 4o, ontwikkeld door OpenAI, is gebaseerd op het GPT-4o-model. Dit model kan in realtime redeneren over audio, visuele gegevens en tekst [30]. Een van de functies van ChatGPT is het genereren van code. Hiervoor beschikt het over een aparte interface, genaamd “Canvas”, die speciaal is ontworpen om codeproductie te optimaliseren [31]. In dit onderzoek wordt er echter geen gebruik gemaakt van deze speciale interface; alle gegenereerde code zal uitsluitend via de chatinterface worden aangemaakt.

Eigenschappen Gemini 2.0

Gemini 2.0 Flash Experimental, ontwikkeld door Google, presteert beter dan zijn voorganger Gemini 1.5 Pro [18]. Net als ChatGPT ondersteunt het invoer in de vorm van audio, afbeeldingen en tekst. Daarnaast kan Gemini worden ingezet als zoekmachine, voor het uitvoeren van code, en voor het testen van functies van derden [18]. Opmerking: aangezien Gemini versie 2.0 bij aanvang van het onderzoek nog niet beschikbaar was, werd het onderzoek aanvankelijk uitgevoerd met Gemini 1.5. De uiteindelijke resultaten zijn echter wel getest met Gemini 2.0.

Eigenschappen Microsoft Copilot

Microsoft Copilot, net als ChatGPT gebaseerd op het GPT-4o-model, is ontworpen voor toepassingen in zakelijke en educatieve omgevingen [22]. Het ondersteunt invoer in de vorm van audio, afbeeldingen en tekst, waardoor gebruikers informatie in verschillende formats kunnen verwerken [21]. Een opvallend kenmerk van Copilot is dat het vaak verwijzingen toevoegt aan zijn antwoorden, zodat gebruikers de bron van de informatie kunnen controleren [22]. Hoewel Gemini en ChatGPT dit ook doen, lijkt Copilot dit frequenter te doen. Dit is nuttig in situaties waar transparantie belangrijk is. Daarnaast kan Copilot code interpreteren, maar deze functionaliteit is expliciet beperkt tot Python-code [21].

Eigenschappen GitHub Copilot

GitHub Copilot [12] biedt de mogelijkheid om codevoorstellen te doen als vervanging van *IntelliSense*. Daarnaast biedt het een chatfunctie. Verder toont GitHub Copilot in PyCharm verschillende suggesties in een lijst, waaruit de gebruiker kan kiezen. Hoewel deze functie niet in Visual Studio Code beschikbaar is, biedt VS Code andere opties. Zo kan GitHub Copilot een foutmelding uitleggen met één klik en deze ook oplossen. Daarnaast maakt het in VS Code inline vragen mogelijk.

Op het moment van schrijven van deze thesis zijn meerdere modellen beschikbaar voor GitHub Copilot. Dit onderzoek maakt gebruik van het GPT-4o-model, aangezien dit als enige model niet in preview beschikbaar is bij aanvang van het onderzoek.

Deel I

Iteratieve aanpak oefeningen

Hoofdstuk 3

Overzicht onderzoek naar gebruik van LLMs

3.1 Correct gebruik van LLM's

In de paper *Enhancing Programming Education with ChatGPT: A Case Study on Student Perceptions and Interactions in a Python Course* [25] hebben onderzoekers de percepties van studenten onderzocht na het gebruik van ChatGPT gedurende een acht weken durende Python-cursus. In deze cursus werden studenten aangemoedigd om ChatGPT te gebruiken voor verschillende programmeerdoeleinden. Zo konden ze tijdens opdrachten gebruikmaken van de tool voor het debuggen van code, het genereren van voorbeeldcode, het uitleggen van code, en het verkrijgen van tips voor code-optimalisatie. Daarnaast werd ChatGPT ingezet bij huiswerk-opdrachten, waarbij studenten werden aangemoedigd om de tool te gebruiken om onduidelijke codefragmenten uit de colleges beter te begrijpen. Tijdens individuele miniprojecten werd ChatGPT eveneens toegepast, bijvoorbeeld voor het debuggen, brainstormen, optimaliseren van code, verduidelijken van de requirements en het verkrijgen van een dieper inzicht in de technische aspecten van het project.

Uit de studie bleek dat studenten ChatGPT vooral nuttig vonden voor het verduidelijken van problemen, wat hen hielp de benodigde tijd om oplossingen te vinden te verkorten. De tool werd gewaardeerd vanwege zijn vermogen om fouten te identificeren, verbeteringen te suggereren en studenten te begeleiden bij het verbeteren van hun code. Niettemin uitte een student bezorgdheid over een mogelijke afhankelijkheid van ChatGPT, wat zijn vermogen tot zelfstandig denken en leren zou kunnen beperken. Daarnaast werd opgemerkt dat ChatGPT soms incorrecte oplossingen bood. Een aantal studenten stelde voor dat ChatGPT behulpzamer zou zijn als het stap-voor-stap hints gaf in plaats van direct een volledige oplossing te presenteren.

In de studie *“ChatGPT Is Here to Help, Not to Replace Anybody” - An Evaluation of Students’ Opinions On Integrating ChatGPT In CS Courses* [8] concludeerden onderzoekers dat studenten over het algemeen baat hadden bij het gebruik van ChatGPT. Tegelijkertijd erkenden zij dat sommige studenten moeite hadden om de tool effectief in te zetten. Het geven van specifieke lessen en oefeningen over het gebruik van ChatGPT bleek voordelig te zijn voor het leerproces van de studenten.

Deze studies benadrukken het belang van het correct leren omgaan met GenAI-tools. Ze tonen aan dat effectieve begeleiding essentieel is om de voordelen van deze technologie in een onderwijscontext te maximaliseren.

3.2 Prompts

Om GenAI efficiënt te gebruiken, is het essentieel om prompts zorgvuldig op te stellen. Een nuttig hulpmiddel hiervoor is het *CLEAR*-framework [24], dat bestaat uit vijf componenten: **concise**, **logical**, **explicit**, **adaptive** en **reflective**. De eerste drie componenten zijn met name nuttig voor eenmalige prompts, terwijl de laatste twee meer geschikt zijn voor het ontwikkelen van aanpasbare prompts of prompttemplates. Met andere woorden: studenten zouden zich voornamelijk moeten richten op de eerste drie componenten, terwijl docenten die een API-gedreven bot voor studenten willen ontwerpen baat hebben bij het toepassen van alle vijf de componenten.

Het eerste component, **concise**, benadrukt het belang van het verwijderen van overbodige informatie om de prompt zo duidelijk mogelijk te maken. Het tweede component, **logical**, draait om een samenhangende en logische opbouw van informatie binnen de prompt. Het derde component, **explicit**, verwijst naar het geven van duidelijke en precieze instructies om het gewenste resultaat te verkrijgen. Dit omvat richtlijnen over het formaat, de context en de scope. Bijvoorbeeld: als een antwoord op een codeervraag geen code mag bevatten, moet dit expliciet worden vermeld.

Het vierde component, **adaptive**, suggereert dat prompts flexibel moeten zijn, zodat ze indien nodig kunnen worden aangepast. Bijvoorbeeld: als een prompt te generiek of te specifiek is, kan het gebruik van synoniemen zoals *leg uit* in plaats van *bespreek* het antwoord verbeteren. Het vijfde en laatste component, **reflective**, betreft het gebruik van gebruikersfeedback om de kwaliteit van de prompt continu te verbeteren.

Naast het *CLEAR*-framework heeft Microsoft, de ontwikkelaar van Copilot, ook richtlijnen gegeven voor het maken van effectieve prompts. In hun artikel *Meet your AI assistant for education: Microsoft Copilot* [41], delen ze vijf tips om prompts te optimaliseren.

De eerste tip, **define clear objectives**, benadrukt het belang van het vaststellen van een specifiek doel zoals het creëren van een syllabus. De tweede tip, **be specific**, adviseert duidelijke en gedetailleerde instructies te geven, wat aansluit bij het **explicit**-component van het *CLEAR*-framework. De derde tip, **structure the prompt**, raadt aan om complexe taken op te splitsen in kleinere stappen. Bijvoorbeeld: vraag eerst om een schets van een outline en richt je daarna op specifieke secties.

De vierde tip, **iterate and refine**, moedigt gebruikers aan om prompts te herschrijven en aan te passen op basis van de initiële outputs. Als alternatief kunnen vervolgvragen worden gesteld om de gewenste resultaten te verfijnen. De vijfde tip, **combine experience**, stelt voor om persoonlijke kennis en expertise te benutten bij het verbeteren van prompts.

Deze richtlijnen en frameworks bieden waardevolle handvatten om prompts effectief te formuleren en zo het maximale uit GenAI-tools te halen.

3.3 Feedback

Feedback kan worden onderverdeeld in twee categorieën: **summatieve feedback** en **formatieve feedback**. Automatische beoordelingssystemen zijn voornamelijk ontworpen om summatieve feedback te leveren, zoals scores of foutmeldingen. Echter, ze schieten vaak tekort als het gaat om het bieden van effectieve formatieve feedback, die bedoeld is om het leerproces te ondersteunen en te verbeteren [16].

In de paper *Students' Perceptions and Preferences of Generative Artificial Intelligence Feedback for Programming* [45] ontdekten de onderzoekers dat ChatGPT goed aansluit bij de richtlijnen voor formatieve feedback. Studenten beoordeelden de feedback van ChatGPT als nuttiger dan de feedback die zij doorgaans ontvingen van het automatische beoordelingssysteem dat gekoppeld was aan hun GitHub-repository.

Hoewel studenten de feedback van ChatGPT waardevol vonden, wezen zij ook op enkele beperkingen. Ze merkten op dat de feedback niet volledig foutloos was en minder geschikt bleek voor het genereren van summatieve feedback. Desondanks benadrukten zij de unieke voordelen van ChatGPT voor het ondersteunen van hun leerproces.

Nadat dit deel van het onderzoek was afgerond, werd er een nieuwe paper gepubliceerd [32]. Hoewel het onderzoek in die paper een andere focus had, vonden de auteurs tussendoor vergelijkbare resultaten. Zij trainden een eigen LLM en merkten op dat ook hun model incorrecte verwachte antwoorden genereerde. Daarnaast vergeleken ze hun LLM met onder andere ChatGPT-4o. Hoewel ze hun scores op een andere manier berekenden en met andere opgaven werkten, stelden ze een lage dekkingsgraad en middelmatige accuraatheid vast. Dit suggereert dat de LLM mogelijk nuttig is in tussentijdse iteraties, maar waarschijnlijk niet betrouwbaar genoeg is voor de uiteindelijke uitkomst. Het benadrukt bovendien het belang voor studenten om de gegenereerde verwachte uitkomsten kritisch te controleren in plaats van ze klakkeloos te accepteren.

3.4 Tutor

Intelligente tutoringsystemen (ITS) hebben bewezen vergelijkbare effecten te kunnen bereiken als menselijke tutors [19]. In de paper *AI-Tutoring in Software Engineering Education* [11] implementeerden onderzoekers een intelligent tutoringssysteem gebaseerd op OpenAI's API. De auteurs suggereerden echter een verschuiving naar een chatbot-gebaseerd systeem, zodat studenten follow-up vragen konden stellen en een interactievere leerervaring kregen.

De paper *Intelligent Tutor: Leveraging ChatGPT and Microsoft Copilot Studio to Deliver a Generative AI Student Support and Feedback System within Teams* [7] biedt interessante inzichten in het gebruik van generatieve AI (GenAI) als persoonlijke tutor. Het implementeren van GenAI in deze rol heeft verschillende voordelen. Het kan bijvoorbeeld academische prestaties verbeteren, de motivatie van studenten verhogen en hun betrokkenheid bij de lessen stimuleren. Bovendien helpt het studenten om de studiestof beter te begrijpen.

Opvallend is dat de studie geen statistisch significant verschil aantoonde in de algehele tevredenheid van studenten over GenAI als tutoringssysteem. Hoewel het geslacht van de studenten geen significante invloed had, bleek de academische achtergrond wel een statistisch significante impact te hebben op hun ervaringen met dit systeem.

3.5 Framework

Seppe Kimps van de Universiteit Hasselt, ontwikkelde een framework om studenten te begeleiden bij het effectief gebruik van generatieve AI [36]. Dit framework beschrijft een gedetailleerd, acht-stappenplan voor het integreren van generatieve AI in programmeeroefeningen. Bij elke stap kunnen studenten feedback, suggesties of volledige oplossingen vragen aan de generatieve AI. In deze thesis wordt deze gestructureerde aanpak toegepast en verder uitgebouwd.

In de eerste stap introduceert Kimps de oefening, die wordt opgedeeld in drie delen: (1) het voorbereiden van een voorbeeld met beschrijvende of visuele elementen, (2) het opdelen van de oefening in kleinere onderdelen, en (3) het beschrijven van de volledige taak in overzichtelijke en uitvoerbare stappen. Kimps merkt op dat de eerste twee delen optioneel zijn. De tweede stap omvat het selecteren van programmeerconstructies en -principes, zoals het bepalen van het type lus (bijvoorbeeld een `for`- of `while`-lus) dat wordt gebruikt, met als doel het ontwikkelen van pseudocode.

De derde stap richt zich op het identificeren van een basisoplossing, waarbij studenten het kernprobleem oplossen dat later kan worden uitgebreid om de volledige taak te dekken. In de vierde stap implementeren studenten deze basisoplossing door hun pseudocode te vertalen naar een werkend programma. De vijfde stap omvat het identificeren en oplossen van eventuele

fouten of waarschuwingen die zich voordoen, waarbij studenten de nodige stappen ondernemen om deze problemen aan te pakken.

In de zesde stap maken studenten testcases om alle mogelijke invoer, inclusief randgevallen, te dekken en ervoor te zorgen dat de code naar behoren werkt. De zevende stap omvat het uitbreiden van de oplossing door stappen vier tot en met zes te herhalen voor elke extra functionaliteit of vereiste, wat uiteindelijk leidt tot een volledig programma. De achtste en laatste stap — uitgevoerd zonder hulp van AI — vraagt studenten om te reflecteren op de oefening. Kimps raadt aan om eventuele delen van de code die ze niet volledig begrijpen te identificeren of om secties te herschrijven voor betere efficiëntie.

Hoofdstuk 4

Stapsgewijze en iteratieve aanpak

4.1 Achtergrond

Dit onderzoek richt zich op studenten die de introductiecursus *Imperatief Programmeren* volgen. Aangezien het een inleidende cursus betreft, wordt ervan uitgegaan dat de studenten weinig tot geen programmeerervaring hebben. De professor die dit vak doceert, *prof. dr. Frank Neven*, merkte op dat sommige studenten vaak moeite hebben met het starten van een oefening.

Om deze drempel te verlagen, wordt overwogen om LLM's in te zetten om de oefeningen te vereenvoudigen. Het idee is om studenten eerst een vereenvoudigde versie van een oefening aan te bieden, die ze vervolgens kunnen uitbreiden naar de volledige opgave. Het wordt verwacht dat een eenvoudiger start studenten helpt om beter te begrijpen hoe ze de oefening moeten aanpakken. Zodra dit inzicht verworven is, zouden ze makkelijker aan de slag kunnen gaan.

Deze aanpak sluit aan bij het stappenplan van *Seppe Kimps*. In het bijzonder komt dit overeen met stap drie en vier in paragraaf 3.5. De LLM zou worden gebruikt om stap drie, het identificeren van een basisoplossing, te vereenvoudigen. De student moet vervolgens zelf de pseudocode opstellen en deze daarna implementeren.

Voor de initiële opgave zijn er tests voorzien op het platform Dodona. Indien een student ervoor kiest de oefening op te splitsen in kleinere deelstappen, kunnen deze standaardtests mogelijk niet geschikt zijn. Daarom wordt er met behulp van een LLM voorgesteld om testfuncties te genereren voor deze deelstappen. Aangezien de gekozen deelstappen per student kunnen verschillen, is het niet mogelijk om deze tests vooraf op te stellen. Dit sluit aan bij stap zes van het stappenplan van Seppe Kimps.

Het stappenplan is tevens opgenomen in de studieleidraad van het vak, zij het in een aangepaste vorm. De versie uit de studieleidraad is weergegeven in Bijlage A. Hierin zijn stap drie, vier en zes samengevoegd tot één enkele stap, aangeduid als stap C. Dit onderzoek is gericht op deze aangepaste versie van het stappenplan.

Stap C is onderverdeeld in drie deelstappen: *Implementeer een basisoplossing*, *Test met eenvoudige voorbeelden* en *Itereer en breid uit*. Binnen deze structuur worden twee fasen voor de prompts onderscheiden. In de eerste fase vraagt de student de LLM om de opdracht op te splitsen in verschillende deelopgaven. In de tweede fase vraagt de student de LLM om testcases te genereren voor deze deelopgaven.

4.2 Methode van aanpak

LLM's geven een niet-deterministisch antwoord, wat betekent dat ze voor dezelfde inputprompt telkens een verschillend antwoord kunnen genereren. Hierdoor is het onmogelijk om één optimale prompt te bepalen. Het doel van dit onderzoek was dan ook niet om de beste prompt te vinden, maar om te onderzoeken of het überhaupt mogelijk was om deze methode toe te passen. Indien dit het geval bleek, was het de bedoeling een prompt te ontwikkelen die in de meeste gevallen goed functioneerde.

Om deze prompt te vinden is gebruikgemaakt van *prompt engineering*. In deze fase zijn er geen objectieve scores toegekend om te bepalen welke prompt het beste was. In plaats daarvan zijn de gegenereerde antwoorden beoordeeld op hun kwaliteit en ingeschat of ze op het eerste gezicht correct leken. Voor meer informatie hierover, zie deelparagraaf 4.2.1 voor de fase waarin de opdracht wordt opgedeeld in stappen en deelparagraaf 4.2.2 voor de fase waarin tests worden gegenereerd.

Nadat geschikte prompts waren gevonden die over het algemeen bevredigende antwoorden opleverden, zijn deze voorzien van een score. De verdeling van deze scores is te zien in deelparagraaf 4.2.3. Tijdens de ontwikkeling zijn bewust geen scores toegekend, omdat de focus lag op het creëren van een bruikbare prompt in plaats van een prompt die uitsluitend goed presteert in benchmarks.

Benchmarks vormen immers geen perfecte representatie van het effectieve gebruik in realistische scenario's. Indien de prompts zouden afgestemd worden op hoge benchmarkscores, zou dit mogelijk leiden tot een vertekend beeld van de werkelijke prestaties. Daarom is tijdens het *prompt engineering*-proces steeds het CLEAR-framework [24] als leidraad genomen.

4.2.1 Prompts voor het genereren van deeltappen

Deze fase richt zich op het genereren van deeltappen met behulp van LLM's. Voor de ontwikkeling is er aanvankelijk gefocust op een representatieve opgave, namelijk de opdracht *Eenvoudige Rekenmachine* [28], zoals weergegeven in Bijlage B. Binnen het vak zijn er vier opdrachten op punten; deze specifieke opdracht is de derde. Om die reden werd dit als een representatieve oefening beschouwd.

De bedoeling van deze opdracht was dat studenten een rekenmachine implementeerden. Deze rekenmachine hoefde de wiskundige volgorde van bewerkingen niet te respecteren, maar moest de berekening van links naar rechts uitvoeren. De implementatie moest ondersteuning bieden voor de operatoren: `+`, `-`, `*`, `/` en `^`. Indien er een fout optrad, moest de functie "NaN" retourneren. De implementatie moest aanroepbaar zijn via de functie `evaluate()`, die één parameter ontving: een string die de invoer representeerde. Deze string kon spaties bevatten. Studenten mochten typecasts en de `range`-functie gebruiken, maar geen andere hulpfuncties. Indien ze extra functionaliteit nodig hadden, moesten ze deze zelf implementeren.

Analyse van studentenoplossingen Als eerste stap zijn de oplossingen van studenten bestudeerd om de mogelijke deeltappen te achterhalen. Enkel de uiteindelijke oplossingen van 48 studenten stonden ter beschikking, maar niet hun tussenstappen. Hierdoor kon enkel een inschatting gemaakt worden van de strategieën die ze mogelijk hebben gevolgd.

Bij de analyse is er gelet op verschillende aspecten:

- Was er een functie aanwezig die twee getallen en een operator verwerkt?
- Hoe werden loops gebruikt? Werd er één loop gebruikt voor meerdere operatoren, en was deze aangepast vanuit een implementatie voor één operator?
- Werden er meerdere loops gebruikt? Bijvoorbeeld één voor de bewerking met meerdere operatoren en een aparte voor de laatste twee operaties? Was dit een copy-paste van een eerdere loop?

- Welke variabelen en commentaren werden gebruikt? Bijvoorbeeld: `getal_1` en `getal_2` versus `totaal` en `getal`.

Op basis van deze kenmerken zijn de volgende strategieën geïdentificeerd:

- Alles tegelijk
- Eerst twee getallen met operator, daarna meerdere getallen, gevolgd door filtering en foutafhandeling
- Eerst twee getallen en operator (als drie strings), daarna meerdere operatoren, gevolgd door filtering en foutafhandeling
- Onmiddellijk meerdere operatoren, gevolgd door filtering en foutafhandeling
- Niet opgelost of onvoldoende informatie om een strategie te herkennen

Merk op dat met *filtering* wordt bedoeld: het controleren van de invoer en het retourneren van "NaN" bij syntactisch ongeldige input.

Het was niet mogelijk om te achterhalen of studenten eerst een implementatie voor gehele getallen maakten en deze daarna uitbreidden naar floats, of dat ze meteen met floats begonnen. Wel was te zien dat sommige studenten de invoer afzonderlijk inlazen (bijvoorbeeld alleen een `getal` of alleen een operator) door te controleren of hiervoor een aparte functie was voorzien. Dit werd beschouwd als een deelstap en niet als een vereenvoudigde opgave.

Na deze analyse begon het *prompt engineering*-proces. Zodra de resultaten van de LLM's voldeden, kregen zowel ChatGPT, Copilot als Gemini (Flash 1.5) de opdracht om de taak op te splitsen in deelopgaven. Dit proces werd driemaal herhaald.

De gevonden deelopgaven waren als volgt:

ChatGPT:

- Onmiddellijk meerdere operatoren, dan filteren en foutafhandeling.
- Eerst twee getallen met operator, dan filteren en foutafhandeling, dan meerdere getallen.
- Eerst twee getallen met operator, dan meerdere getallen, dan filteren en foutafhandeling.

Gemini:

- Eerst twee getallen en operator (als drie strings), dan meerdere operatoren, dan filteren en foutafhandeling.
- Eerst twee getallen met operator, dan meerdere getallen, dan filteren en foutafhandeling.
- Eerst twee getallen met operator, dan filteren en foutafhandeling, dan meerdere getallen.

Copilot:

- Onmiddellijk meerdere operatoren, dan filteren en foutafhandeling.
- Eerst twee getallen met operator, dan meerdere getallen, dan filteren en foutafhandeling.
- Eerst twee getallen met operator, dan filteren en foutafhandeling, dan meerdere getallen.

Wat opvalt, is dat elk van deze modellen een oplossingsmethode heeft gevonden die niet expliciet door de studenten werd toegepast. Tegelijkertijd werden alle door studenten gebruikte strategieën ook door de LLM's herkend.

Copilot en ChatGPT produceerden dezelfde deelopgaven, wat logisch is aangezien ze beide gebaseerd zijn op dezelfde GPT-versie. Gemini kwam echter met een alternatieve strategie.

Omdat de exacte strategieën van studenten niet met zekerheid kunnen worden bepaald, kunnen hier geen harde conclusies uit worden getrokken. Bovendien blijven LLM's zich ontwikkelen.

Zelfs tijdens het schrijven van deze thesis heeft Google een nieuwe versie van Gemini uitgebracht. Hierdoor kunnen de voorgestelde deelopgaven in de toekomst veranderen.

Dit wordt vermeld omdat de output van de LLM's werd gebruikt als indicatie om te beoordelen of de prompt bruikbaar was, en niet als definitief bewijs van effectiviteit.

Prompt engineering Als eerste stap in het prompt engineering-proces kreeg ChatGPT een prompt met enkel de opgave, waarna het resultaat werd geobserveerd. Direct viel op dat het model code genereerde. Aanvankelijk werd geprobeerd dit te verhelpen door de prompt te herschrijven, zodat duidelijk was dat alleen de deelopgaven gewenst waren. Toen dit niet het gewenste effect had, werd expliciet vermeld dat er geen code mocht worden gegenereerd. Dit leek in eerste instantie succesvol, maar later bleek dat LLM's deze instructie soms verkeerd interpreteerden.

Daarom werd onderzocht wat er gebeurde wanneer een voorbeeld werd gegeven van de gewenste antwoordstructuur. Het gebruikte voorbeeld is terug te vinden in Figuur 4.1. Copilot bleef echter code genereren. Vervolgens werd expliciet vermeld dat een nieuwe opgave gewenst was, maar dit had een negatief effect op het resultaat. Na verdere tests bleek dat zelfs met een voorbeeld de instructie nog steeds verkeerd geïnterpreteerd kon worden.

Een oefening om tekst in allemaal hoofdletters te zijn, terwijl er ook speciale tekens voorkomen. Wordt: Eerst enkel kleine letters uitwerken, dan uitbreiden met hoofdletters, en (dan uitbreiden) met speciale tekens.

Figuur 4.1: Het voorbeeld dat aan het model meegegeven werd.

Om te voorkomen dat er code werd gegenereerd, werd geprobeerd een reden te geven waarom code niet gewenst was. Dit zorgde er inderdaad voor dat Copilot stopte met het genereren van code, maar in plaats daarvan werd een algemeen stappenplan gegeven zonder concrete deelopgaven. Deze specifieke prompt¹ is te vinden in Figuur 4.2.

Dit is mijn opgave: <opgave>
Geef een vereenvoudigde versie van de opgave. Bijvoorbeeld: een oefening om tekst in allemaal hoofdletters te zetten, terwijl er ook speciale tekens voorkomen. Wordt: Eerst enkel kleine letters uitwerken, dan uitbreiden met hoofdletters, en (dan uitbreiden) met speciale tekens. Ik wil geen code zien, aangezien ik dit zelf moet implementeren.

Figuur 4.2: Deze prompt leidde tot een belangrijke doorbraak, omdat het ervoor zorgde dat de LLM's geen code meer genereerden. Dit is enkel de beschrijving van de prompt-structuur. Om de effectieve prompt te verkrijgen, moet <opgave> vervangen worden door de daadwerkelijke opgave.

Ik heb een opgave gekregen. Ik weet niet hoe ik moet beginnen. Kan je mij een eenvoudigere opgave geven, die ik later tot de echte opgave kan uitbreiden?

Figuur 4.3: Dit is de initiële prompt dat gebruikt werd om de opgave door te geven aan de LLM.

Daarnaast werd er geëxperimenteerd met het automatisch laten genereren van prompts door de LLM zelf. Dit werkte tot op zekere hoogte, maar de modellen konden hun eigen prompts niet volledig volgen. Daarom was dit geen haalbare optie.

¹Tijdens de ontwikkeling zijn veel prompts opgesteld. Daarom is het niet mogelijk om ze allemaal in dit thesisverslag op te nemen. Enkel de prompts die een significante doorbraak opleverden, worden besproken.

Verder werd onderzocht of het helpen zou om in plaats van de exacte opgave een beschrijving te geven. De gedachte hierachter was dat een kortere prompt mogelijk tot een beter resultaat zou leiden. Dit bleek echter niet het geval te zijn. Vervolgens werd een conversationele aanpak geprobeerd waarbij eerst de opgave werd gegeven, gewacht werd op een antwoord en daarna de prompt werd ingevoerd. In Figuur 4.3 is te zien dat hierbij niet expliciet werd vermeld dat er geen code mocht worden gegenereerd.

Uit de antwoorden van ChatGPT en Copilot bleek dat de instructie werd begrepen. Dit resulteerde in een vereenvoudigde opgave zonder implementatie. Gemini daarentegen interpreteerde de vraag niet correct. Hoewel de modellen de opdracht leken te begrijpen, genereerde ChatGPT bij de volgende stap alsnog code. Copilot volgde de instructie beter op, maar de output werd alsnog in codevorm gepresenteerd. Aangezien Gemini de vraag in eerste instantie al niet goed begreep, was dit geen verrassend resultaat.

Hoewel sommige modellen nog steeds code genereerden, werd besloten om alles in één uitgebreide prompt te combineren. Dit bleek de beste resultaten op te leveren.

Dit is mijn opgave: <opgave>
Geef een vereenvoudigde versie van de opgave. Die ik op een incrementele aanpak kan uitbreiden. Bijvoorbeeld: een oefening om tekst in allemaal hoofdletters te zetten, terwijl er ook speciale tekens voorkomen. Wordt: Eerst enkel kleine letters uitwerken, dan uitbreiden met hoofdletters, en vervolgens uitbreiden met speciale tekens. Ik wil geen code zien, aangezien ik dit zelf moet implementeren.

Figuur 4.4: Dit is de uiteindelijke prompt dat succesvol bleek voor het genereren van een vereenvoudigde opgave. Dit is enkel de beschrijving van de promptstructuur. Om de effectieve prompt te verkrijgen, moet <opgave> vervangen worden door de daadwerkelijke opgave.

Na verdere experimenten bleek dat expliciet vermelden dat de opgave op een incrementele manier moest worden uitgebreid (zie Figuur 4.4) ervoor zorgde dat de modellen deze instructie consequent opvolgden. Hoewel slechts om een vereenvoudigde versie van de opgave werd gevraagd, gaven Gemini 2.0, ChatGPT en Copilot systematisch meerdere stappen tegelijk.

Oorspronkelijk was het plan om de LLM eerst één stap te laten genereren, vervolgens de code in te voeren en daarna de volgende stap te vragen. Echter, omdat deze prompt ervoor zorgde dat de LLM's alle stappen ineens gaven — wat efficiënter bleek — werd besloten deze aanpak te hanteren.

4.2.2 Prompts voor het genereren van testen

Net zoals bij deelparagraaf 4.2.1 lag de focus in deze fase eerst op de rekenmachine. Allereerst werd onderzocht of de LLM's überhaupt in staat waren om tests te genereren. Dit gebeurde door de LLM's testgevallen te laten maken voor de initiële opgave en vervolgens de coverage te vergelijken met Dodona.

Wanneer simpelweg werd gevraagd om testfuncties te genereren, deden ChatGPT en Copilot dit zoals verwacht. Gemini 1.5 daarentegen gaf naast de tests ook een implementatie. Dit werd opgelost door expliciet te vragen geen implementatie te genereren. De prompt die werd gebruikt om te testen of de LLM's testgevallen konden genereren, is te zien in Figuur 4.5.

In Tabel 4.1 is te zien op welke criteria punten zijn toegekend om te bepalen welke LLM het best presteerde en hoe Dodona zich daartoe verhoudt. Alle modellen detecteerden correct de operatoren en de deling door 0. ChatGPT herkende echter niet de floatnotatie $x..$ Wat betreft spaties: Dodona, ChatGPT en Gemini vonden allemaal ontbrekende spaties, terwijl Copilot dit niet detecteerde. Bovendien was ChatGPT de enige die ook overvloedige spaties herkende.

Dit is mijn opgave: <opgave>
 Genereer test cases voor mij in de vorm van functie aanroepen. Ik
 wil geen implementatie zien.

Figuur 4.5: Dit is de prompt dat gebruikt werd om de testcoverage van de LLM's te vergelijken met Dodona. Dit is enkel de structuur van de prompt; om de effectieve prompt te krijgen, moet <opgave> vervangen worden door de daadwerkelijke opgave.

Scoreverdeling	Dodona	ChatGPT	Copilot	Gemini
Testen van operatoren: +, -, *, /, ^	5/5	5/5	5/5	5/5
Delen door 0	1/1	1/1	1/1	1/1
Floats (x.x, x., .x, .)	3/4	2/4	3/4	3/4
Spaties (geen/te veel)	1/2	2/2	0/2	1/2
Operatoren op verkeerde plaats/ Operatoren achter elkaar	1/2	2/2	1/2	2/2
Verkeerde tekens (haakjes)	2/2	0/2	0/2	0/2
Lege input/spatie-input	0/2	0/2	1/2	2/2
Enkel floats, geen operator	0/1	0/1	0/1	1/1
Volgorde van bewerkingen	1/1	1/1	1/1	1/1
Enkel een operator	0/1	0/1	0/1	1/1
Meerdere punten in dezelfde float (x.x/x.x.x)	0/2	2/2	1/2	1/2
Meerdere leidende nullen	0/1	0/1	0/1	0/1
Witruimte binnen getallen	0/1	0/1	0/1	0/1
Totaal	14/25	15/25	13/25	18/25

Tabel 4.1: scores van de testcoverage per LLM in vergelijking met Dodona

Bij operatoren op een verkeerde plaats: Dodona vond alleen een minteken aan het begin, terwijl Copilot enkel meerdere achter elkaar vond. De andere LLM's detecteerden beide gevallen. Voor floats met meerdere punten vond Gemini `x.x.x`, Copilot vond `x.x`, en ChatGPT vond beide, terwijl Dodona er geen vond. Copilot detecteerde een lege string als input, terwijl Gemini zowel een lege string als een spatie-input herkende. Alleen Gemini herkende invoer met enkel floats zonder operator. Alle modellen testten de volgorde van bewerkingen. Enkel Gemini controleerde op invoer die alleen uit een operator bestond. Geen van de modellen testte op invoer met alleen een punt (`.`), meerdere leidende nullen of witruimte binnen getallen.

Daarna werd aan de LLM's gevraagd of hun tests volledig waren en of er nog gevallen ontbraken. Dit werd gedaan met de prompt:

Waarom zijn je tests volledig? Of ontbreken er nog?

Gemini gaf aan dat er altijd nog randgevallen mogelijk waren, maar gaf geen extra testcases. Copilot en ChatGPT vulden daarentegen hun tests aan. ChatGPT voegde tests toe voor ongeldige tekens en zorgde ervoor dat alle mogelijke float notaties (`x.x`, `x.`, `.x`, `.`) gedekt waren. Daarnaast voegde het tests toe voor lege input of invoer die uitsluitend uit spaties bestond. Copilot voegde tests toe die controleerden of een operator aan het begin van de invoer stond, naast reeds bestaande tests voor meerdere opeenvolgende operatoren. Hij testte ook wat er gebeurde als een float zonder operator werd ingevoerd en wat er gebeurde bij invoer met alleen een operator.

Daarna werd onderzocht wat er zou gebeuren als de LLM's de testcases van Dodona kregen en werd gekeken welke extra tests ze zelf nog zouden vinden. De gebruikte prompt is te zien in Figuur 4.6. Uit Tabel 4.2 blijkt dat alle LLM's — ChatGPT, Copilot en Gemini 1.5 — extra testcases genereerden. Geen van hen wist echter alle mogelijke tests te vinden. Nadat was vastgesteld dat de LLM's in staat waren om testcases te genereren, werd onderzocht hoe tests konden worden gegenereerd voor een vereenvoudigde versie van de rekenmachinefunctie

Dit is mijn opgave: <opgave>
 Dit zijn mijn test cases: <test cases>
 Zijn deze volledig?

Figuur 4.6: Dit is de prompt dat gebruikt werd om te controleren welke aanvullingen de LLM's gaven wanneer ze de tests van Dodona kregen. Dit is enkel de structuur van de prompt; om de effectieve prompt te krijgen, moet <opgave> vervangen worden door de daadwerkelijke opgave en <test cases> door de testcases van Dodona.

Scoreverdeling	Dodona	ChatGPT	Copilot	Gemini
Testen van operatoren: +, -, *, /, ^	5/5	5/5	5/5	5/5
Delen door 0	1/1	1/1	1/1	1/1
Floats (x.x, x., .x, .)	3/4	4/4	3/4	3/4
Spaties (geen/te veel)	1/2	2/2	2/2	2/2
Operatoren op verkeerde plaats/				
Operatoren achter elkaar	1/2	2/2	2/2	2/2
Verkeerde tekens (haakjes)	2/2	2/2	2/2	2/2
Lege input/spatie-input	0/2	0/2	1/2	0/2
Enkel floats, geen operator	0/1	1/1	0/1	0/1
Volgorde van bewerkingen	1/1	1/1	1/1	1/1
Enkel een operator	0/1	1/1	0/1	0/1
Meerdere punten in een float	0/2	1/2	0/2	0/2
Meerdere leidende nullen	0/1	0/1	0/1	1/1
Witruimte binnen getallen	0/1	0/1	0/1	1/1
Totaal	14/25	20/25	17/25	18/25

Tabel 4.2: scores van de testcoverage per LLM na het invoeren van de Dodona-tests

met minder functionaliteit.

Prompt engineering met functies voor deelstappen Aangezien ChatGPT, Gemini en Copilot chat-gebaseerde LLM's zijn, was het mogelijk om door te vragen op de gegenereerde deelopgaven en hiervoor testfuncties te laten maken. De eerste prompt die hiervoor werd gebruikt, werkte direct. In deze prompt werd gevraagd:

“Maak voor elk van deze stappen een testfunctie. Deze testfuncties moet ik in de main kunnen aanroepen. Ik wil dat ze de vorm hebben van `print(<arg>, <functie(arg)>, <verwacht resultaat>).`”

ChatGPT en Copilot gaven een antwoord zoals verwacht, maar Gemini 1.5 genereerde slechts testcases voor sommige deelopgaven en niet voor alle. Uiteraard was deze prompt nog niet optimaal. Naast het verbeteren van de formulering, moest ook het aanroepen van de testfuncties eenvoudiger worden gemaakt.

Eerst lag de focus op het ervoor zorgen dat de testfuncties direct aanroepbaar waren in de `main`, zodat studenten ze eenvoudig konden kopiëren. Vervolgens werd geprobeerd dit werkend te krijgen op Gemini 1.5, maar zonder succes. Omdat dit niet lukte op Gemini Flash 1.5, werd de aanpak getest op Gemini Pro 1.5. Dit gaf betere resultaten, maar nog steeds niet voldoende. Ook werd geprobeerd om zowel de deelopgaven als de testen tegelijkertijd te laten genereren, maar dit werkte nog minder goed.

Met de release van Gemini Flash 2.0 (Experimental) werd het wel mogelijk om testfuncties te genereren. Daarom werd vanaf dat punt het onderzoek verder toegespitst op deze krachtigere versie van Gemini, die bovendien gratis beschikbaar was. Hoewel dit een ander model is, wordt de aanname gehanteerd dat LLM's steeds meer kunnen. Bovendien blijven de LLM-modellen getraind worden op nieuwe data.

Later werd de layout van de output aangepast, zodat de input, het verwachte resultaat en het resultaat elk op een aparte regel werden geprint. Daarnaast moest het resultaat en het verwachte resultaat op dezelfde plaats beginnen, zodat studenten eenvoudig afwijkingen konden zien. In eerste instantie werd dit geprobeerd met tabs, maar het aantal spaties per tab verschilt per terminal, zelfs binnen hetzelfde besturingssysteem. In Windows Terminal gebruikt Python bijvoorbeeld 8 spaties per tab, terwijl PyCharm er slechts 4 gebruikt. Hierdoor bleken tabs geen betrouwbare optie. Vervolgens werd overgestapt op de string-operatie `rjust(20)`, waarmee tekst rechts wordt uitgelijnd op kolom 20.

Met deze aanpak gaven de LLM's nog steeds slechts één test per deelfunctie. Dit probleem werd uiteindelijk opgelost door een regel met `'...'` toe te voegen. Deze prompt werkte goed voor de rekenmachine, maar was nog niet voldoende om ook correcte layout-generatie te garanderen bij andere soorten uitvoer, zoals uitvoer over meerdere regels, bijvoorbeeld bij een schaakbord. Dit werd opgelost door de tekst:

```
"Ik verwacht dat je de vorm aanpast aan de opgave. Dit wil zeggen dat
bij een resultaat dat op meerdere lijnen uitgeprint hoort te worden, je
dit ook op een nieuwe regel uitprint."
```

toe te voegen aan de prompt.

Uiteindelijke prompts met functies In de uiteindelijke prompt was het niet mogelijk om één enkel prompt te gebruiken voor zowel ChatGPT, Copilot als Gemini. Daarom werd dit opgesplitst in twee aparte prompts. De eerste prompt (Figuur 4.7) wordt gebruikt door ChatGPT en Copilot, aangezien ze hiermee de beste resultaten geven.

Wanneer deze prompt echter aan Gemini werd gegeven, leek het aanvankelijk goed te werken. Bij het testen met meerdere opgaven bleek echter dat Gemini soms antwoorden verklapte. Om deze reden werd voor Gemini (Flash 2.0 Experimental) een alternatief prompt opgesteld (Figuur 4.8).

Indien deze alternatieve prompt werd gebruikt voor ChatGPT of Copilot, verminderde de kwaliteit van het antwoord aanzienlijk. ChatGPT stopte met het genereren van testfuncties en gaf in plaats daarvan pseudo-code voor deze functies. Copilot bleef testfuncties genereren in de correcte vorm, maar genereerde er minder dan met de oorspronkelijke prompt.

Aangezien Copilot op dezelfde GPT-versie gebaseerd is als ChatGPT, werd ervoor gekozen om Copilot dezelfde prompt te laten gebruiken als ChatGPT, en niet die van Gemini.

```
Maak voor elk van deze stappen een testfunctie. Deze testfuncties
moeten in de main aanroepen kunnen worden. Ik wil dat deze functies
verschillende print statements hebben met de vorm:
print("parameter:".rjust(20), <arg>)
print("resultaat:".rjust(20), <functie(arg)>)
print("verwacht resultaat:".rjust(20), <verwacht resultaat>)
Print()
...
Print bij iedere functie ook uit wat er getest wordt. Ik verwacht
dat je de vorm aanpast aan de opgave, dit wil zeggen dat bij een
resultaat dat op meerdere lijnen uitgeprint hoort te worden, je dit
ook op nieuwe regel uitprint.
```

Figuur 4.7: Dit is de uiteindelijke prompt om testfuncties te genereren voor ChatGPT en Copilot. In deze prompt hoeven de parameters tussen `'<'` en `'>'` niet vervangen te worden door de student. Dit is namelijk de taak van de LLM. Merk op dat in deze prompt “aanroepen” staat, en niet “aangeroeppen” zoals je zou verwachten. Daarnaast begint één “print” willekeurig met hoofdletter terwijl de rest dat niet heeft.

Maak voor elk van deze stappen een testfunctie. Deze testfuncties moeten in de main aanroepen kunnen worden. Ik wil dat deze functies verschillende print statements hebben met de vorm:

```
print("parameter:".rjust(20), <arg>)
print("resultaat:".rjust(20), <functie(arg>)>)
print("verwacht resultaat:".rjust(20), <verwacht resultaat>)
Print()
...
```

Print bij iedere functie ook uit wat er getest wordt. Ik verwacht dat je de vorm aanpast aan de opgave, dit wil zeggen dat bij een resultaat dat op meerdere lijnen uitgeprint hoort te worden, je dit ook op nieuwe regel uitprint.

Geef me geen implementatie (code) van de opgave.

Figuur 4.8: Dit is de uiteindelijke prompt om testfuncties te genereren bij Gemini 2.0. In deze prompt hoeven de parameters tussen '<' en '>' niet vervangen te worden door de student. Dit is namelijk de taak van de LLM. Merk op dat in deze prompt “aanroepen” staat, en niet “aangeropen” zoals je zou verwachten. Daarnaast begint één “print” willekeurig met hoofdletter terwijl de rest dat niet heeft.

Dit is mijn opgave: <opgave>

Ik vroeg je: Geef een vereenvoudigde versie van de opgave. Die ik op een incrementele aanpak kan uitbreiden. Bijvoorbeeld: een oefening om tekst in allemaal hoofdletters te zetten, terwijl er ook speciale tekens voorkomen. Wordt: Eerst enkel kleine letters uitwerken, dan uitbreiden met hoofdletters, en vervolgens uitbreiden met speciale tekens. Ik wil geen code zien, aangezien ik dit zelf moet implementeren.

Jij antwoordde: <antwoord>

Dit is mijn implementatie voor stap <X>:

<implementatie>

Is dit correct, of is dit fout? Waar moet ik aandacht aan besteden?

Ik wil niet dat je me code geeft, dit moet ik zelf doen.

Figuur 4.9: Prompt die het beste werkte voor het verkrijgen van feedback op de eigen implementatie.

Prompt engineering zonder functies voor deelstappen De prompts uit paragraaf 4.2.2 werken enkel in combinatie met functies. Echter, aan het begin van de cursus hebben studenten nog geen kennism gemaakt met functies. Daarom is er ervoor gekozen om input/output-paren te genereren met behulp van LLM’s.

In eerste instantie werd geprobeerd de LLM formatieve feedback te laten geven op code. De gebruikte prompt hiervoor is te zien in Figuur 4.9. Dit werd echter losgelaten, omdat het vereiste dat studenten hun code naar de LLM stuurden. Dit zou ertoe kunnen leiden dat de LLM leert van de ingezonden code en deze vervolgens aan andere studenten voorstelt. Dit kan problemen veroorzaken bij plagiaatcontrole. Hoewel dit minder problematisch is bij oefeningen, vormt het bij opdrachten die meetellen voor punten een groot nadeel. Om deze reden werd ervoor gekozen om over te schakelen naar input/output-paren.

Uit eerdere experimenten met prompt engineering bleek dat LLM’s beter luisteren wanneer een reden wordt opgegeven. Daarom werd dit principe direct toegepast. De eerste prompt werkte al redelijk goed, maar genereerde slechts één input/output paar per stap. Dit probleem werd opgelost door het woord *verschillende* toe te voegen, wat direct effect had.

Bovendien gaven de LLM’s de input en output consequent weer in codeblocks, waardoor studenten deze gemakkelijk konden kopiëren. De uiteindelijke prompt is te zien in Figuur 4.10.

Maak voor elk van deze stappen verschillende input-output paren. Dit zodat ik kan testen of dat mijn implementatie correct is. Ik wil dat ik de input makkelijk kan kopiëren.

Figuur 4.10: prompt die het beste werkte voor het genereren van input/output-paren

4.2.3 Score

Voor zowel de fase van het genereren van stappen als die van het genereren van testcases is een scoreverdeling opgesteld. Beide fases zijn beoordeeld aan de hand van zes categorieën: *correctheid*, *granulariteit*, *kwaliteit*, *efficiëntie*, *flexibiliteit* en *integriteit*. Deze categorieën zijn verder opgesplitst in specifieke onderdelen. Voor iedere categorie wordt een subtotaal weergegeven, en uiteindelijk wordt een totaalscore berekend. Deze scores worden afzonderlijk berekend voor elk model en voor elke oefening.

Evaluatie criteria voor het genereren van stappen De categorie *correctheid* is opgedeeld in twee onderdelen: *completeness* (compleetheid) en *soliditeit*. Voor het onderdeel *completeness* wordt verwacht dat alle noodzakelijke stappen aanwezig zijn en correct zijn geformuleerd. Als een belangrijke stap ontbreekt, wordt dit genoteerd. De score wordt als volgt bepaald: als alle noodzakelijke stappen aanwezig zijn, wordt 100% toegekend; ontbreekt één noodzakelijke stap, dan wordt de score 50%; ontbreken meerdere noodzakelijke stappen, dan wordt 0% toegekend. Voor het onderdeel *soliditeit* wordt verwacht dat elke opgenomen stap noodzakelijk en relevant is voor het behalen van de doelstelling. Overbodige of irrelevante stappen worden vermeld. De score wordt op dezelfde manier als bij *completeness* berekend, met een aftrek van 50% per irrelevante stap.

De categorie *granulariteit* is eveneens opgedeeld in twee onderdelen: *te veel stappen* en *te weinig stappen*. Als het plan onnodig gedetailleerd is, wat kan leiden tot inefficiëntie of verwarring, bevat het stappenplan overbodige stappen. Dit geldt ook voor redundante of gesplitste stappen zonder duidelijke meerwaarde. In dat geval wordt 50% afgetrokken per overbodige stap. Als het plan daarentegen belangrijke details mist, waardoor het moeilijk wordt om het algoritme te implementeren, bevat het stappenplan te weinig stappen. Dit geldt ook als complexe stappen niet voldoende worden opgesplitst om praktisch uitvoerbaar te zijn. Ook hier wordt een aftrek van 50% per fout toegepast.

Om verwarring te voorkomen kan een stap niet tegelijkertijd vallen onder zowel “*correctheid*, *soliditeit*” als “*granulariteit*, *te veel stappen*”. Een stap behoort slechts tot één van beide categorieën. Als de stap inhoudelijk fout is, valt deze onder *correctheid*. Als de stap te klein is maar inhoudelijk correct, valt deze onder *granulariteit*. Op dezelfde manier kan een stap niet tegelijkertijd worden beoordeeld als “*correctheid*, *completeness*” en “*granulariteit*, *te weinig stappen*”. Een stap behoort ook hier slechts tot één categorie. Als een noodzakelijke stap voor het oplossen van de oefening ontbreekt, waardoor functionaliteit ontbreekt, valt deze onder *correctheid*. Als twee opeenvolgende stappen te groot zijn (te veel tegelijk), valt dit onder *granulariteit*.

De categorie *kwaliteit* is, net als de voorgaande categorieën, opgedeeld in twee onderdelen: *begrijpbaarheid* en *duidelijkheid*, en *complexiteit*. Bij *begrijpbaarheid* en *duidelijkheid* wordt verwacht dat de stappen helder en ondubbelzinnig zijn beschreven. Het taalgebruik en de terminologie moeten consistent en geschikt zijn voor de doelgroep. De score wordt eerst booleaans berekend voor elke stap afzonderlijk. Dit betekent dat een stap ofwel 100% van de score krijgt, ofwel 0%. Vervolgens wordt het gemiddelde van alle stappen berekend. Voor het onderdeel *complexiteit* wordt verwacht dat moeilijke of complexe stappen voldoende worden toegelicht en, indien nodig, worden ondersteund door hulpmiddelen of vereisten. De score wordt op dezelfde manier berekend als bij *begrijpbaarheid* en *duidelijkheid*.

De categorie *efficiëntie* bestaat uit één onderdeel. Hier wordt verwacht dat het stappenplan tijd- en resourcegebruik bij de implementatie minimaliseert. Dit houdt in dat het systeem geen

Categorie	Onderdeel	Punten
Correctheid	Completeness	/10
Correctheid	Soliditeit	/10
Correctheid	Totaal	/20
Granulariteit	Te veel stappen	/5
Granulariteit	Te weinig stappen	/10
Granulariteit	Totaal	/15
Kwaliteit	Begrijpbaarheid en Duidelijkheid	/5
Kwaliteit	Complexiteit	/5
Kwaliteit	Totaal	/10
Efficiëntie	Totaal	/5
Flexibiliteit	Totaal	/5
Integriteit	Totaal	/20
Totaal	Totaal	/75

Tabel 4.3: Dit is de overzichtstabel die de puntenverdeling voor elke categorie en elk onderdeel weergeeft.

onnodige redundantie in de logica of structuur bevat die de uitvoering vertraagt. De score wordt, net als bij de categorie *kwaliteit*, eerst per stap booleaans berekend. Vervolgens wordt hiervan het gemiddelde berekend.

De categorie *flexibiliteit* bestaat eveneens uit één onderdeel. Hierbij wordt beoordeeld of het stappenplan eenvoudig kan worden aangepast aan variaties in de input en andere contextuele eisen. Er wordt ook gekeken of het stappenplan voldoende ruimte biedt voor iteratieve verbetering. Ook hier wordt de score per stap booleaans berekend, waarna het gemiddelde wordt genomen.

Ook binnen de categorie *integriteit* is er, net als bij *efficiëntie* en *flexibiliteit*, slechts één beoordelingscriterium. Hierbij wordt beoordeeld of de LLM antwoorden direct voorzegt aan de student. Indien dit voorkomt, wordt dit genoteerd. De score voor dit onderdeel wordt per stap als een booleaanse waarde bepaald. Vervolgens wordt ook hiervan het gemiddelde berekend.

Voor elk onderdeel binnen *correctheid* en *granulariteit*, met uitzondering van *te veel stappen*, kunnen maximaal 10 punten worden behaald. Voor *integriteit* bedraagt de maximale score 20 punten. Alle overige onderdelen, waaronder *te veel stappen*, kennen een maximale score van slechts 5 punten. Dit komt doordat een overschot aan stappen als minder problematisch wordt beschouwd dan een tekort. De verdeling van de scores wordt samengevat in Tabel 4.3.

Evaluatie criteria De categorie *correctheid* is onderverdeeld in drie onderdelen: *foutloze output*, *juiste oplossingen* en *consistentie*. Bij het onderdeel *foutloze output* wordt verwacht dat er geen syntactische of logische fouten in de testcases voorkomen. Als dit wel het geval is, wordt dit genoteerd en wordt de score met 50% verminderd per foute stap. Indien er slechts één stap is, gaat de score direct naar nul. Voor het onderdeel *juiste oplossingen* wordt geëist dat de testcases de juiste verwachte output genereren voor een gegeven input. Bij afwijkingen in de verwachte uitkomsten wordt dit eveneens genoteerd. De score wordt berekend op basis van het aantal stappen. Stel dat er n stappen zijn, dan wordt er $\frac{2}{n}$ punten afgetrokken per fout, ten opzichte van een totaalscore van $\frac{n}{n}$. Met andere woorden, als 50% van de stappen fouten bevat, is de score 0. Bevat 25% van de stappen fouten, dan bedraagt de score 50%. Vervolgens wordt deze score omgerekend naar het correcte aantal punten. Het onderdeel *consistentie* vereist dat de output identiek moet zijn bij meerdere uitvoeringen met dezelfde inputs. Dit houdt in dat de testcases deterministisch moeten zijn. De score wordt binair bepaald: 100% of 0%.

De categorie *granulariteit* bevat vier onderdelen: *detailniveau*, *variatie in complexiteit*, *minimale edge cases* en *extra edge cases*. Het onderdeel *detailniveau* verwacht dat de testcases voldoende gedetailleerd zijn om specifieke aspecten van de deelopgave te testen. Indien dit niet het geval is, wordt dit genoteerd. Er wordt op dezelfde manier als in de categorie *correctheid* 5 strafpunten per fout aangerekend. Het onderdeel *variatie in complexiteit* vereist dat de testcases zowel eenvoudige als complexe scenario's dekken. Indien dit niet het geval is, wordt dit eveneens genoteerd. Voor het onderdeel *minimale edge cases* wordt verwacht dat de testcases de meest voor de hand liggende randgevallen bevatten. Indien dit niet het geval is, wordt ook dit genoteerd. De score voor zowel *variatie in complexiteit* wordt berekend op basis van het aantal stappen, als *minimale edge cases* op basis van het aantal edge cases. Stel dat er n stappen zijn, dan wordt er $\frac{2}{n}$ punten afgetrokken per stap die niet voldoet, ten opzichte van een totaalscore van $\frac{n}{n}$. Met andere woorden, als 50% van de stappen niet voldoet, is de score 0. Voldoen 25% van de stappen niet, dan bedraagt de score 50%. Vervolgens wordt deze score omgerekend naar het correcte aantal punten. Het onderdeel *extra edge cases* begint met een score van 0, waarbij voor elke toegevoegde extra edge case één punt wordt toegekend. Een randgeval dat minder voor de hand ligt, wordt beschouwd als een extra edge case. Zodra zo'n geval wordt gevonden, wordt dit genoteerd.

Kwaliteit bestaat uit drie onderdelen: *leesbaarheid*, *relevantie* en *documentatie*. Het onderdeel *leesbaarheid* beoordeelt of de testcases goed gestructureerd en leesbaar zijn. Als dit niet het geval is, wordt dit genoteerd. De score wordt eerst per stap afzonderlijk berekend en daarna wordt het gemiddelde genomen. Elke stap kan drie scores krijgen: 100% voor perfecte uitvoering, 50% voor een kleine schoonheidsfout (bijvoorbeeld een newline bij output op meerdere regels), en 0% voor onbruikbare output. Het onderdeel *relevantie* kijkt of de testcases relevant zijn voor de deelopgave en geen overbodige of irrelevante testen bevatten. Indien dit wel het geval is, wordt dit genoteerd. De score wordt per stap booleaans (0% of 100%) berekend en het gemiddelde van alle stappen wordt genomen. Onder *documentatie* wordt verwacht dat er waar nodig duidelijke uitleg en annotaties worden toegevoegd. Als dit niet het geval is, wordt dit genoteerd. De score wordt, net als bij *relevantie*, booleaans per stap berekend. Let op: als een stap geen documentatie heeft, maar dit geen meerwaarde zou geven, wordt toch 100% toegekend.

De categorie *efficiëntie* bestaat uit twee onderdelen: *tijdsduur* en *prestatie*. Het onderdeel *tijdsduur* verwacht dat de testcases binnen een redelijke tijdsduur uitgevoerd kunnen worden. Indien dit niet het geval is, wordt dit genoteerd. Het onderdeel *prestatie* verwacht dat de testcases efficiënt omgaan met bronnen en niet onnodig veel rekenkracht vereisen. Indien dit niet het geval is, wordt ook dit genoteerd. De score wordt voor beide onderdelen booleaans berekend per stap, en vervolgens wordt hier het gemiddelde van genomen, gelijkaardig aan de categorie *kwaliteit*.

Vervolgens bestaat de categorie *flexibiliteit* uit drie onderdelen: *aanpasbaarheid*, *uitbreidbaarheid* en *compatibiliteit*. Het onderdeel *aanpasbaarheid* verwacht dat de testcases eenvoudig aan te passen zijn mocht een verwacht resultaat niet kloppen. Indien dit niet het geval is, wordt dit genoteerd. Het onderdeel *uitbreidbaarheid* verwacht dat er eenvoudig gelijkaardige testcases toegevoegd kunnen worden. Indien dit niet het geval is, wordt dit ook hier genoteerd. Vervolgens verwacht het onderdeel *compatibiliteit* dat de testcases compatibel zijn met verschillende opgaven en systemen waar mogelijk. Testcases mogen bijvoorbeeld niet enkel op Windows werken. Indien dit niet het geval is, wordt dit genoteerd. De scores van deze onderdelen worden gelijkaardig als bij de categorieën *kwaliteit* en *efficiëntie* berekend, eerst booleaans per stap, vervolgens wordt er het gemiddelde van genomen.

De categorie *integriteit* bestaat uit één beoordelingscriterium, waarbij wordt geëvalueerd of de LLM antwoorden direct voorzegt aan de student. Indien dit voorkomt, wordt dit genoteerd. De score voor dit onderdeel wordt per stap als een booleaanse waarde bepaald. Vervolgens wordt hiervan het gemiddelde berekend.

Met uitzondering van *integriteit*, *foutloze output* en *juiste oplossing*, worden per onderdeel,

Categorie	Onderdeel	Punten
Correctheid	Foutloze output	/20
Correctheid	Juiste oplossingen	/20
Correctheid	Consistentie	/5
Correctheid	Totaal	/45
Granulariteit	Detailniveau	/10
Granulariteit	Variatie in complexiteit	/5
Granulariteit	Minimale Edge cases	/5
Granulariteit	Extra Edge cases	/5
Granulariteit	Totaal	/25
Kwaliteit	Leesbaarheid	/5
Kwaliteit	Relevantie	/5
Kwaliteit	Documentatie	/5
Kwaliteit	Totaal	/15
Efficiëntie	Tijdsduur	/5
Efficiëntie	Prestatie	/5
Efficiëntie	Totaal	/10
Flexibiliteit	Aanpasbaarheid	/5
Flexibiliteit	Uitbreidbaarheid	/5
Flexibiliteit	Compatibiliteit	/5
Flexibiliteit	Totaal	/15
Integriteit	Totaal	/40
Totaal	Totaal	/150

Tabel 4.4: Dit is de overzichtstabel die de puntenverdeling voor elke categorie en elk onderdeel weergeeft.

afhankelijk van het belang, maximaal 10 of 5 punten toegekend. Voor zowel *foutloze output* als *juiste oplossing* is de maximale score 20 punten, terwijl voor *integriteit* maximaal 40 punten worden toegekend. Een overzicht van deze verdeling is te vinden in Tabel 4.4.

4.3 Resultaten

Er wordt een kwantitatieve en een kwalitatieve analyse uitgevoerd, die de basis vormen voor de conclusies. De kwalitatieve analyse richt zich op één oefening met input/output-paren en één oefening met testfuncties. In de kwantitatieve analyse wordt van elke les van het opleidings-ondereel Imperatief Programmeren I één oefening onderzocht. De moeilijkheidsgraad van de oefeningen verschilt echter. Bovendien moet een oefening opgesplitst kunnen worden. Om te bepalen of dit mogelijk is, wordt gekeken of de oefening atomaire kenmerken heeft. Een atomaire oefening is een oefening die geen tests, zoals een if-statement, en heeft geen loops zoals een while- of for-loop bevat. Atomaire oefeningen worden uitgesloten van de analyse. Enkel oefeningen met minstens één test, één loop, of beide, worden meegenomen. Alle resultaten zijn afgerond tot op één decimaal.

4.3.1 Kwantitatieve analyse

DPL Hoofdstuk 8: functies - Gemeenschappelijke karakters Bij de eerste opgave dient de student een functie `gemeenschappelijke_karakters` te implementeren. Deze functie retourneert het aantal gemeenschappelijke karakters tussen twee strings. Het is cruciaal dat dubbele karakters slechts één keer worden geteld. Bijvoorbeeld, indien de input `'een'` en `'twee'` is, dient de functie 1 als resultaat te geven.

In deze eerste opgave blijkt uit de analyse van Copilot dat dit systeem in de eerste stap alleen kleine letters beschouwt, in de tweede stap ook hoofdletters en in de derde stap speciale tekens. Gemini geeft een gelijkaardig antwoord. ChatGPT voegt echter twee extra stappen toe: één voor cijfers en één voor spaties en lege invoer. Hoewel Dodona enkel op de eerste opgave test, is het in de praktijk de bedoeling om meer te testen. Voor de *kwaliteit: begrijpbaarheid en duidelijkheid* moeten bij Copilot en Gemini punten worden afgetrokken. Dit omdat beide systemen een voorbeeld geven dat niet overeenkomt met de opgave voor die stap. Copilot beschouwt hoofdletters als hetzelfde symbool als kleine letters, terwijl Gemini hoofdletters als een verschillend symbool behandelt. ChatGPT laat dit onduidelijk, wat leidt tot een aftrek van een punt voor *begrijpbaarheid en duidelijkheid*. Dit resulteert in een score van 3,3/5 voor zowel Copilot als Gemini en 4/5 voor ChatGPT. De opgave zelf bevatte echter enige ambiguïteit, waardoor er geen punten worden afgetrokken voor Gemini of Copilot om deze reden. Verder zijn er geen andere bemerkingen. Dit resulteert in een score van 74/75 voor ChatGPT en 73,3/75 voor Gemini en Copilot.

Wanneer er naar de tests gekeken wordt, heeft Copilot zijn aanpak aangepast en beschouwt hij hoofdletters nu als een ander karakter. Dit leidt tot een foutief verwachte uitkomst in de tweede stap. In de derde stap is er opnieuw een verkeerd verwacht resultaat, wat resulteert in een score van 0/20 voor *correctheid: juiste oplossing*. Copilot test ook geen extra edge cases, wat de score voor deze categorie verlaagt tot 125/150. ChatGPT levert een verkeerd verwacht resultaat in de tweede stap door hoofdletters mee te tellen. Hoewel het verwachte resultaat dan drie zou zijn, klopt dit niet, omdat ChatGPT in de code een commentaar gaf waarin hij bepaalde karakters benoemde. In dit geval telde ChatGPT het karakter dat als kleine letter in de ene string en als grote letter in de andere voorkwam wel mee, maar vergat hij een ander karakter. Bij de vierde stap geeft ChatGPT eveneens een verkeerd verwachte uitkomst. Dit resulteert in een score van 4/20 voor *correctheid: juiste oplossing*. ChatGPT test wel extra edge cases, waaronder invoer die enkel bestaat uit spaties. Dit wordt als een extra edge case beschouwd, omdat dit geen lege invoer is en ook geen geldige invoer met spaties. Dit levert ChatGPT een score van 1/5 voor *granulariteit: extra edge cases* op. Verder zijn er geen opmerkingen, wat resulteert in een score van 130/150 voor ChatGPT. Gemini heeft in 10 van de 22 gevallen een verkeerd verwacht resultaat. Dit resulteert in een score van 1,8/20 voor *correctheid: juiste oplossing*. Gemini test echter wel het extra edge case waarin zowel hoofdletters als kleine letters in een bepaalde string voorkomen, vergeleken met een string waarin alleen de kleine letter voorkomt. Daarnaast test Gemini spaties, wat niet in zijn stappenplan stond. Dit leidt tot een score van 2/5 voor *extra edge cases*. Voor de overige categorieën zijn er geen verdere opmerkingen, wat resulteert in een score van 128,8/150 voor Gemini.

Assignment 3 - Eenvoudige Rekenmachine Bij deze opgaven moesten de studenten een rekenmachine implementeren die van links naar rechts evalueert. Bij de deelopgaven valt er niets op te merken over ChatGPT en Gemini, die beide een score van 75/75 behalen. Copilot mist echter de stap om meerdere operatoren toe te voegen, wat resulteert in een score van 5/10 voor *correctheid: completeness*, en daarmee een eindscore van 70/75. Copilot test wel alle minimale edge cases, zoals deling door 0 en meerdere operatoren, en controleert ook op ongeldige invoer. Dit leidt tot een score van 1/5 voor *extra edge cases*. Verder zijn er geen opmerkingen, wat Copilot een score van 146/150 oplevert.

ChatGPT test extra edge cases, zoals floats die beginnen of eindigen met een punt, en ongeldige invoer, zoals een operator zonder getal erna. Dit levert ChatGPT een score van 4/5 voor *extra edge cases*. ChatGPT test ook de minimale edge cases, zoals deling door nul en meerdere operatoren. Verder zijn er geen opmerkingen, wat resulteert in een score van 149/150 voor ChatGPT.

Gemini beschouwt in stap twee ook negatieve getallen. Omdat in de opgave expliciet staat dat het om positieve kommagetallen gaat, wordt dit als een fout gerekend. Hetzelfde geldt voor stap vier. In stap vijf geeft Gemini ook een verkeerd verwacht resultaat. In totaal zijn er vier van de 29 verwachte resultaten fout, wat resulteert in een score van 14,5/20 voor *correctheid*:

juiste oplossingen. Gemini test naast de minimale edge cases ook ongeldige invoer, een operator op de verkeerde plaats, meerdere operatoren achter elkaar, lege invoer, een float bestaande uit enkel een punt, en een float met meerdere punten achter elkaar. Dit levert Gemini een score van 5/5 voor *extra edge cases*, wat resulteert in een eindscore van 144,5/150.

DPL Hoofdstuk 10: Strings - Palindrome sentences In de oefening waarbij de studenten moesten bepalen of een opgave een palindroom was, behaalden alle LLM's 75/75 voor het genereren van stappen. Copilot en Gemini hanteerden een vergelijkbare aanpak, waarbij de student eerst een test uitvoerde op een palindroom met alleen kleine letters, daarna met hoofdletters en tenslotte met speciale tekens. ChatGPT volgde een andere methode, waarbij de student eerst drie stappen moest uitvoeren om de tekst correct te structureren, daarna moest testen of de tekst een palindroom was, en uiteindelijk alles samenvoegde. Hoewel deze aanpak juist is, kreeg de student pas bij de laatste stap een werkend resultaat.

Op het gebied van testen scoorde Copilot perfect, maar vanwege het ontbreken van extra edge cases behaalde het een score van 145/150. ChatGPT daarentegen kreeg slechts 4/20 voor de juistheid van de oplossingen. Dit kwam doordat bij twee van de vijf stappen foutieve voorbeeldoplossingen werden gegeven: bij stap 3 werd het verwachte resultaat van stap 2 getoond en bij stap 4 werd een string gepresenteerd die duidelijk geen palindroom was, maar als verwachte uitkomst voor de functie `is_palindrome()` wel “true” werd aangegeven. Daarnaast werden er geen extra edge cases getest, wat resulteerde in een score van 129/150 voor ChatGPT. Gemini scoorde 0/20 voor de juistheid van de oplossingen, omdat bij elke stap ten minste één testcase fout was. Gemini ontving echter extra punten voor het testen van input met alleen speciale tekens en alleen getallen. Dit leidde tot een score van 22/25 voor *granulariteit*. Op de overige punten viel niets op aan te merken, waardoor Gemini eindigde met een score van 127/150.

DPL Hoofdstukken 11 en 12: Tuples en lijsten (deel 1) - Clean Tuples In de opdracht waarbij de studenten duplicaten uit een tuple moesten verwijderen en enkel het eerste voorkomen van een item moesten behouden, scoorde Copilot voor het genereren van stappen 0/10 op *correctheid: soliditeit*. Dit kwam doordat Copilot eerst de oefening voorstelt met lijsten van integers, waarna het wordt uitgebreid naar lijsten in het algemeen. Vervolgens wordt alles verworpen en opnieuw gestart, maar nu voor tuples van integers, en later wordt dit verder uitgebreid. Dit resulteerde in onnodig dubbel werk. ChatGPT bevatte één stap te veel; het stelde voor om eerst van integers naar strings uit te breiden en daarna verder naar andere datatypes. Dit zou samengevoegd kunnen worden in één stap. Gemini begon met het retourneren van de input, wat overbodig was. Vermits dit juist was binnen het stappenplan valt dit onder granulariteit. Gemini stelde ook voor om eerst uit te breiden met een enkele string en daarna de rest, waarbij het expliciet over andere tuples sprak. Dit verhoogde de moeilijkheidsgraad. ChatGPT had een extra stap voor dit proces. Omdat Gemini hier tuples als elementen behandelde, werden er strafpunten toegekend. Zowel ChatGPT als Gemini scoorden voor *granulariteit* 12,5/15. Er viel verder niets op aan te merken, waardoor Copilot eindigde met 65/75, en Gemini en ChatGPT beide eindigden op 72,5/75.

Wat betreft de tests ontbrak er overal één minimale edge case, namelijk de lege tuple. Gemini testte integers, strings, floats en booleans, terwijl ChatGPT geen booleans testte. Copilot testte geen floats, wat resulteerde in scores van 0/5 voor Copilot, 1/5 voor ChatGPT, en 3/5 voor Gemini voor *minimale edge cases*. ChatGPT en Copilot testten slechts één tuple per geval, maar hier werd geen aftrek toegepast omdat dit voldoende was. Gemini testte wel verschillende tuples per test. Voor Extra Edge cases vonden zowel Gemini als ChatGPT twee gevallen, namelijk tuples als elementen en lijsten als elementen van de tuples, wat hen beide 2/5 opleverde voor *extra edge cases*. Dit leidde tot de eindscore van 145/150 voor Gemini, 140/150 voor Copilot en 143/150 voor ChatGPT.

DPL Hoofdstukken 11 en 12: Lijsten (Deel 2) - Schaakbord Bij de opdracht waarbij de studenten een schaakbord met symbolen '.' en '*' moesten genereren, stelde ChatGPT voor om eerst een r bij k grid van enkel punten te maken. Vervolgens werd een één bij k grid met afwisselende punten en asterisken voorgesteld, waarna dit werd uitgebreid naar een r bij k grid met een patroon van punten en asterisken. Later werd het schaakbord als geheel voorgesteld en ten slotte werd een dynamisch r bij k grid geïntroduceerd. De eerste stap is echter niet passend voor het stappenplan en de vijfde stap creëert verwarring omdat er niet expliciet werd vermeld dat r en k vaste waarden hadden. Deze stap wordt onder *granulariteit: soliditeit* geplaatst, wat de score voor deze categorie verlaagt naar 2,5/5. Hierdoor komt de score voor ChatGPT op 67,5/75.

Gemini stelde voor om eerst één regel met afwisselende punten en asterisken te implementeren, gevolgd door twee regels: de eerste met een punt en de tweede met een asterisk. De derde stap bestond uit een r bij k grid waarbij elke regel met een punt begon. Als vierde stap werd het volledige schaakbord voorgesteld. De tweede stap is echter niet relevant voor het stappenplan en valt onder *correctheid: soliditeit*, wat de score voor deze categorie op 5/10 brengt. Daardoor behaalt Gemini 70/75.

Copilot stelde voor om te beginnen met een r bij k grid van enkel punten, gevolgd door het volledige schaakbord. Daarna werd de opdracht opnieuw voorgesteld, maar dan met de mogelijkheid om het beginsymbool te kiezen, zoals een punt. Tenslotte werd opnieuw de volledige opdracht voorgesteld. Stappen drie en vier waren overbodig, wat leidde tot een score van 0/10 voor *correctheid: soliditeit*. Dit resulteerde in een score van 10/20 voor *correctheid*. Aangezien deze stappen redundante logica zouden kunnen introduceren, werd stap drie ook als inefficiënt beschouwd. Dit leidde tot een score van 3,7/5 voor efficiëntie. Copilot scoort in totaal 63,7/75.

Wat betreft de testcases, botste ChatGPT op een fout doordat zijn testcases veronderstelden dat de functie het resultaat zou teruggeven terwijl het werd uitgeprint. Dit veroorzaakte een crash, maar als het werd aangepast, gaf ChatGPT het juiste resultaat consistent weer. ChatGPT scoorde 0/20 voor *foutloze output* en behaalde daardoor 15/45 voor *correctheid*. De minimale edge cases, zoals even of oneven r en k , werden getest, maar niet de extra edge cases, zoals nul of negatieve waarden. Dit leidde tot een score van 125/150 voor ChatGPT.

Net als ChatGPT gaat Gemini ervan uit dat de student het resultaat uitprint en niet retourneert, wat leidt tot een crash van de code. Om deze reden krijgt hij voor *correctheid: foutloze output* een score van 0/20. De voorgestelde opgaven zijn echter wel correct en consistent, wat resulteert in een score van 25/45 voor *correctheid*. Extra edge cases worden niet getest en van de minimale edge cases wordt slechts 50% getest, namelijk alleen even schaakborden. De oneven waarden voor r en k worden niet meegenomen. Dit leidt tot een score van 15/25 voor *granulariteit*. Verder zijn er geen op- of aanmerkingen, waardoor Gemini een totaalscore van 120/150 behaalt.

In het geval van Copilot wordt dezelfde fout gemaakt met het printen van het resultaat, maar de code crasht niet. Het verwachte resultaat wordt correct als lijst uitgeprint, wat positief bijdraagt aan de kwaliteit. Omdat het slechts om een schoonheidsfoutje gaat, krijgt Copilot voor *kwaliteit: leesbaarheid* een score van 2,5/5, wat resulteert in 12,5/15 voor *kwaliteit*. Voor *correctheid* scoort Copilot 45/45, maar voor *granulariteit* is de score lager. De waarde van r wordt telkens als 3 en k als 4 getest, wat resulteert in een score van 0/5 voor *variatie in complexiteit*. De minimale edge cases, zoals even r en oneven k , worden niet getest, wat eveneens leidt tot een score van 0/5. Er worden geen extra edge cases getest. Dus ook hier is de score 0/5. In stap drie wordt nog steeds getest met het puntje in plaats van het sterretje, wat de score voor *detailniveau* beperkt tot 5/10, oftewel 5/25 voor *granulariteit*. Verder zijn er geen op- of aanmerkingen, waardoor Copilot een totaalscore van 127,5/150 behaalt.

DPL Hoofdstukken 13 en 14: Dictionaries en sets - Merging dictionaries Bij de oefening waarbij studenten dictionaries moesten samenvoegen, toont Copilot een gestructureerde aanpak. Echter gaat het verder dan de oorspronkelijke opdracht. Naast het samenvoegen van

twee dictionaries houdt het rekening met gedeelde keys en voegt het lijsten samen indien de keys overeenkomen. Hierdoor scoort Copilot slechts 50% op *efficiëntie*, wat resulteert in een score van 72,5/75. In de tweede fase, waarin testcases gegenereerd moeten worden, geeft Copilot alle oplossingen weg, wat niet de bedoeling is. Dit leidt tot een score van 0/40 op *integriteit* en een totaalscore van 105/150. ChatGPT slaagt niet voor de laatste stap, wat onder *granulariteit: detailniveau* valt, en test geen van de geïntroduceerde features. Dit resulteert in een score van 5/10 voor deze categorie en een totaalscore van 140/150.

Gemini hanteert een vijfstappenplan, waarbij de laatste stap als optioneel wordt aangegeven. Omdat deze expliciet als optioneel wordt vermeld, worden hier geen punten voor afgetrokken. De eerste twee stappen bevatten echter overbodige tussenstappen die niet bijdragen aan een efficiënte oplossing. Bovendien ontbreekt uitleg over waarom een derde, lege dictionary aangemaakt moet worden om wijzigingen buiten de scope te voorkomen. Dit valt onder *kwaliteit: complexiteit*, wat resulteert in puntenaftrek en een granulariteitsscore van 3/5. Hierdoor behaalt Gemini een score van 73/75. Hoewel er geen extra edge cases worden getest, voldoet de rest van de aanpak. Dit leidt tot een eindscore van 145/150.

Assignment 4 - Schaken Bij deze opdracht moesten studenten, gegeven een schaakbord, aangeven welke stukken elk schaakstuk kon slaan. Copilot houdt geen rekening met obstakels: in de eerste fase wordt aangenomen dat studenten dit zelf zien. In de tweede fase wordt het niet getest. Omdat dit een ontbrekende stap is, valt dit onder *correctheid: completeness*. Dit resulteert in een score van 5/10. Hierdoor behaalt Copilot 70/75 voor het genereren van stappen. Bij het testen worden slechts het schaakstuk zelf en vijandelijke stukken weergegeven, waardoor de student niet kan controleren of eigen stukken correct worden gefilterd. Dit leidt tot een score van 5/10 op *granulariteit: detailniveau*. Daarnaast worden er geen extra edge cases getest. In de tweede stap worden direct de eindposities weergegeven in plaats van de bewegingen (zoals “x omhoog, y omlaag”), wat pas in een latere fase had moeten gebeuren. Dit leidt tot een score van 12/20 op *correctheid: juiste oplossingen*, met een eindscore van 132/150.

ChatGPT kiest een andere aanpak, maar deze is volledig en verdient de maximale 75/75. Bij het testen worden echter de items van de sets geprint in plaats van de sets zelf, wat als een schoonheidsfoutje wordt gezien en resulteert in een score van 2,5/5 op *kwaliteit: leesbaarheid*. Daarnaast worden obstakels niet getest en bij de derde en vierde test wordt een foutief verwacht resultaat gegeven. De minimale edge cases, zoals stukken aan de rand van het bord, worden niet getest. Dit leidt tot een score van 0/5. Voor *extra edge cases* krijgt ChatGPT 1/5, omdat alleen een leeg bord wordt getest. Op *correctheid: juiste oplossingen* behaalt het model een 4/20, en op *granulariteit: detailniveau* een 5/10. Dit brengt de eindscore op 117,5/150.

Gemini maakt een fout in het voorbeeld van de eerste stap en houdt geen rekening met zetten die niet toegestaan zijn. Dit wordt gezien als een ontbrekende stap, wat resulteert in een score van 5/10 op *correctheid: completeness*. De fout in het voorbeeld valt onder *kwaliteit: complexiteit*, wat leidt tot een score van 3,7/5. Hierdoor scoort Gemini 68,7/75 voor het genereren van stappen. Bij de testgeneratie bevatten drie tests een foutief verwacht resultaat. Dit verlaagt de score naar 11/17, omgerekend 12,9/20. Eén extra edge case wordt getest (een bord met slechts één stuk), wat een score van 1/5 oplevert. Alle minimale edge cases, zoals stukken aan de rand van het bord, worden correct getest. Blokkades worden niet getest, maar omdat dit al in de eerste fase als fout werd meegeteld, wordt hier geen extra aftrek toegepast. Gemini genereert de meeste tests van alle modellen en eindigt met een score van 138,9/150.

Herhalingsoefeningen - Game of Life In de eerste fase behalen alle LLM's de maximale score van 75/75. Bij de testcases geeft Copilot één fout verwacht resultaat, wat leidt tot een score van 13,3/20 op *correctheid: juiste oplossingen*. In de stappen 1 tot en met 5 worden uitsluitend symmetrische games getest, waardoor studenten niet kunnen controleren of ze de assen correct hebben verwerkt. Dit resulteert in een 0/10 op *detailniveau*. Extra edge cases worden niet getest, al was dat niet vereist. Door schoonheidsfoutjes in 50% van de stappen scoort Copilot 3,7/5 op *kwaliteit: leesbaarheid*, wat de eindscore op 127/150 brengt. Chat-

GPT gebruikt symmetrische voorbeelden in stappen 2 tot en met 4, waardoor studenten niet kunnen testen of ze rijen en kolommen verwisseld hebben. Dit leidt eveneens tot een 0/10 op *detailniveau*. Er worden geen extra edge cases getest, maar verder zijn er geen fouten, waardoor ChatGPT een eindscore van 135/150 behaalt. Gemini geeft in de tweede fase één fout verwacht resultaat, wat resulteert in een score van 14,3/20 op *correctheid: juiste oplossingen*. De testcases voor fases twee en drie zijn volledig symmetrisch, waardoor studenten niet kunnen controleren of ze kolommen en randen correct hebben verwerkt. Hierdoor scoort Gemini een 0/10 op *detailniveau*. Daarnaast ontbreken alle testen voor stap 4, wat de score op 10/15 brengt voor *flexibiliteit* (3,75/5 op zowel *aanpasbaarheid* als *uitbreidbaarheid*). Dit resulteert in een eindscore van 124,3/150.

Extra oefeningen - Sneeuwvlok Bij de oefening voor het creëren en printen van een sneeuwvlok scoren alle LLM's de maximale 75/75, hoewel Gemini onnodig een voorbeeldfiguur toevoegt. Bij het genereren van testcases maakt Copilot in drie van de vier stappen schoonheidsfouten, wat leidt tot een score van 3,1/5 op *kwaliteit: leesbaarheid*. Daarnaast verklapt Copilot niet alleen de antwoorden, maar ook de oplossingsmethode, wat resulteert in een 0/40 op *integriteit*. Verder gebruikt Copilot een flexibele invoer in plaats van de gevraagde vaste invoer, wat als een syntactische of logische fout wordt beschouwd en leidt tot een 0/20 op *correctheid: foutloze output*. Hierdoor behaalt Copilot een 25/45 op *correctheid* en een eindscore van 83,1/150.

ChatGPT test geen extra edge cases, zoals een 0×0-matrix of negatieve dimensies. Daarnaast scoort het model 0/5 op *flexibiliteit uitbreidbaarheid*, omdat de antwoorden letterlijk zijn geco-deerd. Dit is minder problematisch dan de aanpak van Copilot, waarbij de student de oplossing direct kan overnemen. Aangezien er verder niets op aan te merken valt, behaalt ChatGPT een score van 140/150. Gemini maakt functies die de oplossing al voorzeggen, wat resulteert in een 0/40 op *integriteit*. Hoewel invoerwaarden correct worden gegenereerd, ontbreken extra edge cases. Verder zijn er geen grote fouten, waardoor Gemini een eindscore van 105/150 krijgt.

Opmerking Bij de volgende oefeningen wordt geen gebruik gemaakt van functies, maar van input/output-paren. Hierdoor zijn de aspecten *kwaliteit*, *leesbaarheid*, *efficiëntie* en *flexibiliteit* niet van toepassing. Deze punten worden dan ook niet meegenomen in de beoordeling, om te voorkomen dat ze het uiteindelijke beeld vertekenen.

DPL Hoofdstuk 6: Conditie - Sorteert 3 getallen Bij het veredelen in stappen valt er weinig op te merken bij de verschillende LLM's, waardoor ze allemaal een score van 75/75 krijgen. Wat betreft de input/output-paren scoort Copilot perfect op *correctheid*, maar test alleen het absolute minimum: geen negatieve getallen, geen kommagetallen, geen nul en geen getallen met meer dan één cijfer. ChatGPT test daarentegen wel al deze gevallen en test optioneel op ongeldige invoer. Omdat in de opgave expliciet vermeld werd dat dit niet zou voorkomen, krijgt ChatGPT geen extra punten hiervoor. Maar er worden ook geen punten afgetrokken, aangezien dit optioneel was. ChatGPT krijgt een score van 4/5 voor het testen van extra edge cases, omdat Dodona deze zelf niet test. Gemini test optioneel op gelijke getallen. Hierdoor worden er geen punten gegeven, aangezien dit expliciet werd uitgesloten. Gemini test niet op kommagetallen, maar wel op nul, negatieve getallen en getallen met meer dan één cijfer, waardoor het 3/5 scoort voor extra edge cases. Voor de rest zijn er geen significante opmerkingen. Hierdoor krijgt Copilot een score van 115/120, ChatGPT 119/120 en Gemini 118/120.

DPL Hoofdstuk 7: Iteraties (deel 1) - Rij van Fibonacci Voor de Fibonacci-reeks maakt Copilot twee fouten in de stappen, wat resulteert in een score van 0/10 voor *correctheid: soliditeit*. De eindscore voor Copilot op het verdelen in stappen komt daarmee uit op 65/75. Verder valt er bij ChatGPT niets op te merken, en voor Gemini zijn er ook geen grote tekort-komingen. Echter, bij het genereren van de input/output-paren voor de tests levert één van de

zeven inputs een fout antwoord op, omdat deze net niet ver genoeg gaat in de Fibonacci-reeks. Daarom krijgt Gemini een 14,3/20 voor *correctheid: juiste oplossingen*. Gemini test op negatieve getallen en ongeldige invoer, waardoor het 2/5 scoort voor *extra edge cases*. Dit brengt de uiteindelijke score op 111,3/120. ChatGPT scoort zonder verdere opmerkingen 115/120. Copilot test op negatieve en ongeldige invoer en krijgt daarom ook 2/5 voor *extra edge cases*, wat de uiteindelijke score voor Copilot op 117/120 brengt.

DPL Hoofdstuk 7: Iteraties (deel 2) - Vijf dobbelstenen Bij deze oefening moeten de studenten de kans berekenen dat, bij het gooien van verschillende dobbelstenen achter elkaar, elke worp minstens zo groot is als de vorige worp. De professor verwacht dat de studenten dit met de randomfunctie doen, wat betekent dat ze elke keer een andere uitkomst zullen krijgen. Bij het genereren van de stappen valt er bij Copilot niets op aan te merken, wat hem een score van 75/75 oplevert. Bij het genereren van de testen heeft Copilot echter een verkeerde invoer: bij het genereren van meerdere worpen verwacht hij geen invoer en bij de vierde stap verwacht hij het aantal simulaties in plaats van het aantal dobbelstenen. Hierdoor krijgt Copilot een score van 0/20 voor *correctheid: foutloze output*. Verder maakt Copilot duidelijk dat de output kan variëren bij dezelfde input, wat zijn eindscore op 95/120 brengt. Bij ChatGPT valt er niets op te merken bij het genereren van de stappen, wat hem 75/75 oplevert. Wat betreft de input/output-paren biedt ChatGPT zowel een praktische als een theoretische uitkomst, maar bij één van de zeven input/output-paren ligt de praktische uitkomst te ver van de theoretische uitkomst. Bovendien geeft hij bij dit paar geen theoretische uitkomst, wat resulteert in een score van 14,3/20 voor *correctheid: juiste oplossingen*. ChatGPT maakt echter duidelijk dat het antwoord kan afwijken, wat positief is. Aangezien er verder geen opmerkingen zijn, krijgt ChatGPT een eindscore van 109,3/120. Gemini maakt een vreemde keuze in stap drie, waarbij hij een joker voorstelt die eender welke waarde kan aannemen. Bij stap drie stelt hij voor om bijvoorbeeld te kijken of dat iedere worp twee hoger is dan de vorige. Dit leidt tot een score van 0/10 voor *correctheid: soliditeit*. Deze stappen voegen inefficiëntie toe, wat een score van 0/5 voor *efficiëntie* oplevert, waardoor de totaalscore voor de stappen op 60/75 komt. Bij de testen komen de stappen niet meer overeen met de gegeven stappen, omdat Gemini twee extra stappen heeft toegevoegd tussen stap twee en drie. Doordat hij bij stap twee drie verschillende testmogelijkheden gaf, is dit beschouwd als een fout in de nummering. Dit resulteert in een score van 3,3/5 voor *documentatie*. Daarnaast zijn 50% van de uitkomsten te ver verwijderd van het theoretische antwoord, waardoor Gemini een score van 0/20 voor *correctheid: juiste oplossingen* krijgt. Dit leidt tot een eindscore van 93,3/120 voor Gemini.

DPL Hoofdstuk 7: Iteraties (deel 3) - Gemiddelde Bij de volgende oefening berekent de student het gemiddelde van een reeks getallen, waarbij het aantal invoerwaarden dynamisch is en de reeks eindigt bij het teken 'q'. Copilot genereert de stappen zonder opmerkingen en behaalt daarmee een score van 75/75. Tijdens de testen scoort Copilot echter 0/5 voor *minimale edge cases*, doordat geen enkele test wordt uitgevoerd op lege input, gehele getallen, negatieve waarden of het getal 0; enkel positieve floats worden getest. Dit resulteert in 15/25 voor *granulariteit* en een totaalscore van 110/120. ChatGPT behaalt eveneens 75/75 voor de stappen en test alle gevallen behalve lege input. Dit leidt tot een score van 3/5 voor *minimale edge cases*. Daarnaast worden ongeldige invoer zoals speciale tekens meegenomen, wat 1/5 oplevert voor *extra edge cases*. De *granulariteit* wordt beoordeeld met 19/25 en de totaalscore komt uit op 114/120. Gemini levert correcte stappen aan en behaalt 75/75, maar maakt 5 van de 14 input/output-paren fout, goed voor 5,7/20 op *correctheid: juiste oplossingen*. Alle minimale edge cases worden getest, maar extra edge cases ontbreken. Dit leidt tot 20/25 voor *granulariteit* en een totaalscore van 100,7/120.

Assignment 2 - Printinterval In deze opdracht moesten studenten drie getallen invoeren: a , b en c . Vervolgens moesten voor elk getal i in het interval $[a, b]$ de machten met basis i en exponenten van 0 tot c worden weergegeven, met daarna de som j van deze machten. Bij het voorleggen van deze opgave aan verschillende LLM's traden telkens ernstige fouten op. Daarom

werd besloten om in plaats van een tekstuele kopie een schermafbeelding van de opgave mee te geven, maar zelfs dit leverde slechts matige resultaten op. Copilot start correct met het printen van de volledige opgave, maar toont in de tweede stap de som, terwijl die al in de eerste stap had moeten verschijnen. In de derde stap wordt voorgesteld extra testgevallen toe te voegen, wat op zich verdedigbaar is. De tweede stap is simpelweg incorrect. Hierdoor behaalt Copilot een score van 5/10 op *correctheid: soliditeit*. Daarnaast worden bij elke stap voorbeelden gegeven die niet overeenkomen met de uitleg, wat resulteert in een 0/5 op *kwaliteit: begrijpbaarheid en duidelijkheid*. Bovendien ontbreken enkele essentiële stappen, zoals het expliciet uitprinten van de getallen in het interval, gevolgd door de exponenten en tenslotte de som. Dit had op twee manieren correct kunnen gebeuren: (1) eerst alle getallen in het interval tonen, dan hun exponenten en tenslotte de sommen, of (2) per getal in het interval eerst de exponenten berekenen, dan de som en vervolgens doorgaan naar het volgende getal. Omdat de eindoplossing technisch gezien aanwezig is, bevat de *granulariteit* te weinig tussenstappen, wat resulteert in een 0/10. Copilot behaalt uiteindelijk een 55/75. Bij de tests scoort Copilot 0/20 op *correctheid: juiste oplossingen*, omdat het exponentiëren steeds start vanaf exponent 1 in plaats van 0. Daarnaast worden minimale edge cases, zoals $a = b$ en $c = 0$, niet getest, wat leidt tot een 0/5 op *granulariteit: minimale edge cases*. Dit resulteert in een uiteindelijke score van 90/120. ChatGPT volgt de stappen correct en behaalt een 75/75. Echter, bij het berekenen van de machten wordt een 1 vergeten bij de exponenten van 1, wat resulteert in een fout bij de som. Bij de som van de machten van 5 wordt bovendien een waarde verkeerd berekend. Daarnaast wordt in stap 4 nog steeds het getal i tussen a en b geprint voordat de exponenten beginnen, wat niet de bedoeling is. Doordat drie van de zeven tests fout zijn, krijgt ChatGPT slechts 2,9/20 op *correctheid: juiste oplossingen*. Ook hier ontbreken de minimale edge cases, wat leidt tot een 0/5. Dit resulteert in een score van 92,9/120. Gemini introduceert een stap die totaal niet relevant is, wat leidt tot een 5/10 op *correctheid: soliditeit*. De stappen twee tot en met vier richten zich telkens op een specifiek deel van de opgave, waarna in stap vijf alles wordt samengevoegd. Hoewel deze aanpak ongebruikelijk is, is deze niet fout en scoort Gemini 70/75. Bij de tests worden echter alleen input/output-paren gebruikt die Gemini in de eerste prompt als voorbeeld gaf. Edge cases worden niet getest en in stap vijf ontbreken alle testen. Hierdoor scoort Gemini 5/10 voor *granulariteit: detailniveau* en 10/25 voor *granulariteit* in totaal. De bestaande tests zijn wel correct, wat resulteert in een eindscore van 105/120.

4.3.2 Resultaten kwantitatieve analyse

De resultaten in Tabel 4.5 tonen aan dat de scores van de modellen dicht bij elkaar liggen. Over het algemeen behaalt ChatGPT de hoogste score, gevolgd door Gemini en vervolgens Copilot. Binnen de categorie *granulariteit* blijkt Copilot vaker te weinig stappen te genereren, terwijl ChatGPT en Gemini eerder geneigd zijn om extra stappen te voorzien. Het grootste verschil is zichtbaar binnen de categorie *correctheid*, voornamelijk door de lagere score van Copilot op *soliditeit*.

Vergelijking met subjectieve indrukken tijdens de testgeneratie bevestigt deze rangschikking. ChatGPT werd intuïtief als het best presterende model ervaren. Hierbij lagen Gemini en Copilot dicht bij elkaar, met Gemini net iets beter. Dit is opmerkelijk, aangezien de antwoorden van Copilot en ChatGPT stilistisch veel gelijkenissen vertoonden, terwijl de prestaties van Copilot meer overeenkwamen met die van Gemini.

De resultaten in Tabel 4.6 tonen drie totalen: één voor testgeneratie (**Tests**), één voor input/output-paren (**Input/output**) en een totaal over beide (**Totaal**). Sommige categorieën waren niet van toepassing op input/output-paren en kregen daardoor geen score.

Opvallend is dat ChatGPT in alle totalen de hoogste score behaalt, gevolgd door Gemini en Copilot. Over het algemeen genereerde Gemini de meeste testgevallen, met ChatGPT op de tweede plaats en Copilot als laatste. Dit wijst op een tendens waarbij Gemini een grotere variëteit aan testgevallen produceert, terwijl ChatGPT en Copilot andere sterktes en zwaktes vertonen afhankelijk van de beoordelingscategorie.

Categorie	Onderdeel	Copilot	ChatGPT	Gemini
Correctheid	Completeness	9,3/10	10/10	9,6/10
Correctheid	Soliditeit	7,5/10	9,6/10	8,6/10
Correctheid	Totaal	16,8/20	19,6/20	18,2/20
Granulariteit	Te veel stappen	5/5	4,6/5	4,8/5
Granulariteit	Te weinig stappen	9,3/10	10/10	10/10
Granulariteit	Totaal	14,3/15	14,6/15	14,8/15
Kwaliteit	Begrijpbaarheid en Duidelijkheid	4,5/5	4,9/5	4,9/5
Kwaliteit	Complexiteit	5/5	5/5	4,9/5
Kwaliteit	Totaal	9,5/10	9,9/10	9,7/10
Efficiëntie	Totaal	4,7/5	4,8/5	4,6/5
Flexibiliteit	Totaal	5/5	5/5	5/5
Integriteit	Totaal	20/20	20/20	20/20
Totaal	Totaal	70,3/75	74,0/75	72,3/75

Tabel 4.5: Dit is het overzicht van de scores voor het genereren van stappen: verdeeld per categorie en onderdeel voor elk van de LLM's. De resultaten zijn afgerond op één decimaal.

4.3.3 Kwalitatieve analyse

Voor de kwalitatieve analyse zijn twee representatieve oefeningen geselecteerd die nog niet eerder behandeld zijn. Deze oefeningen zijn terug te vinden in Bijlage C (paragraaf C.1) en Bijlage D (paragraaf D.1). In dit deel worden de uitwerkingen systematisch geanalyseerd. Voor de eerste oefening is Gemini gebruikt, terwijl voor de tweede oefening ChatGPT is ingezet. De eerste oefening richt zich op het genereren van testen, terwijl de tweede oefening de focus legt op het genereren van input/output-paren.

Tests De eerste oefening start met een prompt (Bijlage C, paragraaf C.2) die een stappenplan oplevert (Bijlage C, paragraaf C.3). Vervolgens genereert Gemini testgevallen op basis van een tweede prompt (Bijlage C, paragraaf C.4) waarvan de resultaten weergegeven zijn in Bijlage C (paragraaf C.5).

Gemini verdeelt de opgave in drie stappen. In de eerste stap wordt voorgesteld om enkel kleine letters te versleutelen, zodat de focus niet op het decoderen ligt. Voordat de oefening gestart wordt, dient het stappenplan gecontroleerd te worden. Bij een gebrekkige uitwerking kan een alternatieve versie gegenereerd worden of een andere LLM geraadpleegd worden. In Copilot moeten beide prompts vooraf gegenereerd worden, aangezien deze tool op het moment van schrijven geen aanpassingen aan eerder ingevoerde prompts toelaat. Indien het genereren van de tweede prompt bij Copilot mislukt, moet het stappenplan opnieuw opgesteld worden.

Na een snelle controle van de antwoorden kan de eerste stap geïmplementeerd worden. Uit de testresultaten blijkt dat één van de vier testen faalt (Bijlage C, paragraaf C.6). De oorzaak ligt bij Gemini, die hoofdletters toch omzet en een extra karakter toevoegt. Deze fout is reeds aanwezig in het antwoord op de eerste prompt. Dit benadrukt het belang van een grondige controle van de gegenereerde resultaten. Een voordeel is dat de testgevallen niet handmatig opgesteld en geïmplementeerd hoeven te worden, hoewel extra testgevallen indien gewenst eenvoudig kunnen worden toegevoegd. Indien de testen handmatig geschreven zouden worden, zou eveneens een controle nodig zijn.

De tweede stap omvat de implementatie van zowel kleine als hoofdletters. Opnieuw faalt één van de vier testen (Bijlage C, paragraaf C.7). De oorzaak is wederom een extra karakter in de uitvoer. Bovendien blijkt de beschrijving **Hoofdletters en speciale tekens onveranderd**

Categorie	Onderdeel	Copilot	ChatGPT	Gemini
Correctheid	Foutloze output	17,1/20	18,6/20	18,6/20
Correctheid	Juiste oplossingen	16,1/20	14,9/20	13,1/20
Correctheid	Consistentie	5/5	5/5	5/5
Correctheid	Totaal	38,2/45	38,5/45	36,7/45
Granulariteit	Detailniveau	8,6/10	8,6/10	8,9/10
Granulariteit	Variatie in complexiteit	4,6/5	5/5	5/5
Granulariteit	Minimale Edge cases	3,6/5	3,9/5	3,9/5
Granulariteit	Extra Edge cases	0/5	0,9/5	1,1/5
Granulariteit	Totaal	16,8/25	18,4/25	18,9/25
Kwaliteit	Leesbaarheid	4,3/5	4,7/5	5/5
Kwaliteit	Relevantie	5/5	5/5	5/5
Kwaliteit	Documentatie	5/5	5/5	4,9/5
Kwaliteit	Totaal	14,4/15	14,7/15	14,9/15
Efficiëntie	Tijdsduur	5/5	5/5	5/5
Efficiëntie	Prestatie	5/5	5/5	5/5
Efficiëntie	Totaal	10/10	10/10	10/10
Flexibiliteit	Aanpasbaarheid	5/5	5/5	4,9/5
Flexibiliteit	Uitbreidbaarheid	5/5	4,4/5	4,9/5
Flexibiliteit	Compatibiliteit	5/5	5/5	5/5
Flexibiliteit	Totaal	15/15	14,4/15	14,7/15
Integriteit	Totaal	34,3/40	40/40	37,1/40
Totaal	Tests	125,6/150	134,3/150	130,9/150
Totaal	Input/output-paren	105,4/120	110,0/120	105,7/120
Totaal	Totaal	128,7/150	136,0/150	132,4/150

Tabel 4.6: Dit is het overzicht van de scores per categorie en onderdeel voor elk van de LLM's bij testgeneratie. De resultaten zijn afgerond op één decimaal.

incorrect binnen deze stap. Hoewel Dodona in veel gevallen testgevallen voorziet, zijn deze vaak pas beschikbaar in de laatste stap van een oefening. De gegenereerde testgevallen bieden daarom een praktische aanvulling.

De derde en laatste stap betreft de implementatie van de decodeerfunctie, waarbij enkel n wordt afgetrokken in plaats van opgeteld. Ook hier faalt één van de vier testen (Bijlage C, paragraaf C.8) om dezelfde reden als in de voorgaande stappen.

Over het geheel genomen biedt het gebruik van LLM's zowel voordelen als nadelen. Een belangrijk voordeel is dat geen vereenvoudigde versie van de opgave bedacht hoeft te worden, wat vooral nuttig is voor studenten die moeite hebben met de start van een oefening. Voor gevorderde studenten is dit mogelijk minder relevant, maar voor beginners in een introductiecursus programmeren kan het een waardevolle ondersteuning bieden. Een potentieel nadeel is de noodzaak om te controleren of het stappenplan correct en volledig is. Dit nadeel weegt echter minder zwaar, aangezien een handmatig opgesteld stappenplan eveneens gecontroleerd zou moeten worden.

Een ander voordeel is de automatische generatie van testgevallen, waardoor deze niet handmatig bedacht of geïmplementeerd hoeven te worden. Wel blijft het noodzakelijk de testgevallen te controleren op fouten. Net zoals bij handmatig geschreven testen is deze controle onmisbaar, waardoor het nadeel verwaarloosbaar is ten opzichte van het voordeel.

Een mogelijke zorg is dat studenten minder probleemoplossende vaardigheden ontwikkelen, doordat de opgave niet zelf vereenvoudigd hoeft te worden. Daarom is het aan te raden eerst

zelf een poging te wagen voordat een LLM wordt geraadpleegd. In situaties zoals bij UHasselt, waar probleemoplossende vaardigheden in een apart vak ontwikkeld worden, biedt deze aanpak echter een bijkomend voordeel. Studenten die moeite hebben met probleemoplossing kunnen de programmeeroefening toch maken en blijven bij in de cursus, terwijl zij in een ander vak hun probleemoplossende vaardigheden verder ontwikkelen.

Input/output-paren Bij de oefening gericht op het genereren van input/output-paren genereert ChatGPT een stappenplan (Bijlage D, paragraaf D.2), met bijbehorend antwoord (Bijlage D, paragraaf D.3). Vervolgens wordt een tweede prompt gebruikt (Bijlage D, paragraaf D.4), waarop de gegenereerde input/output-paren te vinden zijn in Bijlage D, paragraaf D.5. De gegenereerde stappen en testgevallen worden gecontroleerd op correctheid en bruikbaarheid.

In de eerste stap wordt gecontroleerd of het ingevoerde getal groter dan of gelijk aan 1 is. Dit maakt het mogelijk om de focus te leggen op het correct opvragen van input en het correct weergeven van output, zonder rekening te houden met verdere logica. ChatGPT genereert testgevallen waarmee kan worden nagegaan of de code correct werkt voor positieve getallen, nul en negatieve getallen.

De tweede stap richt zich op de controle of het getal deelbaar is door twee. Hierbij ligt de focus op de correcte toepassing van de modulo-operator, zonder afleiding door andere aspecten van de opgave. De output van de vorige stap kan eenvoudig worden hergebruikt. Ook voor deze stap voorziet ChatGPT voldoende testgevallen.

In de derde stap wordt deze controle uitgebreid naar alle oneven getallen tot en met het ingevoerde getal. De nadruk ligt hier op de implementatie van een lus, zonder dat optimalisaties al noodzakelijk zijn. Vanaf dit punt is een eerste werkende versie van de oplossing beschikbaar.

De vierde stap introduceert optimalisaties in de controle. Dit leert dat het proces niet eindigt zodra de code functioneert, maar dat verdere verfijning nodig is om efficiëntere en beter leesbare code te verkrijgen. ChatGPT richt zich in deze stap voornamelijk op prestatie.

In de vijfde en laatste stap wordt gecontroleerd of de output correct geformatteerd is. Het is mogelijk dat de uitvoer niet voldoet aan de verwachte structuur, bijvoorbeeld door ontbrekende woorden of een onjuiste zinsbouw. Deze stap benadrukt het belang van een laatste controle op de requirements nadat de code geïmplementeerd is. Voor de stappen drie tot en met vijf genereert ChatGPT aanvullende testgevallen om de oplossing te verifiëren.

Ook hier zijn de voordelen merkbaar: studenten hoeven geen input/output-paren te bedenken en de aanpak helpt tegen het leegschermssyndroom. Door de oefening op te splitsen, is direct zichtbaar welke taak uitgevoerd moet worden, waardoor het eenvoudiger wordt om te starten. Daarnaast biedt deze aanpak directe feedback per deeloplossing. Fouten in een vroege stap zijn eenvoudiger te debuggen dan wanneer de volledige oefening al geïmplementeerd is. Bovendien kan het snel behalen van de eerste deeloplossingen een motiverend effect hebben.

4.4 Conclusie

RQ1: Kan GenAI een oefening vereenvoudigen, zodat de student deze stapsgewijs kan oplossen? Ja, maar de effectiviteit verschilt per oefening en controle blijft noodzakelijk. In sommige gevallen maken LLM's eenvoudige opgaven onnodig complex. Een goed voorbeeld van een succesvolle toepassing is terug te vinden in Bijlage D.

RQ2: Kan GenAI testfuncties genereren voor deze vereenvoudigde opgaven? Ja, al moeten de gegenereerde testen grondig nagekeken worden. Foutieve output is een veelvoorkomend probleem, en niet alle randgevallen worden automatisch gedekt. Over het algemeen volstaan de gegenereerde testen om de deelopgaven te valideren.

RQ3: Kan GenAI tests genereren voor deze vereenvoudigde opgaven in het geval er geen functies aanwezig zijn? Ja, mits de gegenereerde testgevallen en oplossingen gecontroleerd worden. Een goed voorbeeld hiervan is te vinden in Bijlage D.

4.5 Toekomstig werk en mogelijke verbeteringen

Hoewel het gegenereerde stappenplan goed kan scoren, is het mogelijk dat de logica achter bepaalde stappen niet altijd vanzelfsprekend is voor de student. Zelfs wanneer het stappenplan inhoudelijk correct is en de juiste hoeveelheid tussenstappen bevat, kan de meerwaarde van elke afzonderlijke stap niet altijd duidelijk zijn.

Daarnaast wordt het aantal testcases momenteel niet adequaat weerspiegeld in de scores. In sommige gevallen volstaat één test per stap, terwijl in andere situaties zelfs vijf tests per stap onvoldoende blijken te zijn. Dit maakt een consistente evaluatie uitdagend.

Tot slot bleek het moeilijk om voor bepaalde categorieën een eenduidige score te bepalen. Dit suggereert dat verdere verfijning van de evaluatiemethode nodig is om een nauwkeuriger beeld te krijgen van de effectiviteit van de gegenereerde stappenplannen en testcases.

Deel II

Debuggen

Hoofdstuk 5

Overzicht onderzoek naar debuggen

5.1 Debug tools

De paper *What we can learn from how programmers debug their code* [13] bevat enkele inzichtelijke conclusies die voortkomen uit een questionnaire. Een van de bevindingen is dat voor het oplossen van bugs vaak meer dan één regel code moet worden aangepast. Daarnaast blijkt dat het lokaliseren van bugs aanzienlijk meer tijd kost dan het daadwerkelijk oplossen ervan. De meeste bugs zijn semantische fouten (waaronder ook verkeerd geïmplementeerde algoritmes vallen), terwijl concurrency- en geheugenbugs het langst duren om te vinden. Programmeurs volgen overwegend het *replication-observation-deduction*-patroon.

Daarnaast blijkt dat ontwikkelaars vrijwel uitsluitend debugtools gebruiken die in de IDE zijn ingebouwd, zoals printstatements. Zelfstandige debugtools worden nauwelijks benut.

De paper *Debugging strategies and tactics in a multi-representation software environment* [33] stelt dat de mogelijkheid om stap voor stap door een programma te lopen de meest behulpzame functionaliteit vormt binnen een SDE.¹ Bovendien wordt geconcludeerd dat grafische representaties nuttig zijn bij het debuggen. Deze bieden direct inzicht in relevante structuren binnen de probleemruimte, maar zijn niet per definitie superieur aan tekstuele representaties.

Foutboodschappen bevatten veel nuttige informatie. Een veelgebruikte methode is het kopiëren en plakken van de foutboodschap in een zoekmachine om snel een oplossing te vinden of de fout beter te begrijpen [5, 38].

5.2 Debuggen met LLM's

De paper *Debugging without error messages* [34] concludeert dat GPT-3.5 gemiddeld twee tot drie nuttige antwoorden genereert voor elke misleidende reactie wanneer om feedback wordt gevraagd zonder de oorspronkelijke foutboodschap aan de LLM te verstrekken. Zonder voorbeeld en zonder finetuning biedt 45,0% van de resultaten een correcte oplossing, 70,8% een behulpzaam antwoord en 29,2% een misleidende reactie. Dit resulteert in 2,43 behulpzame antwoorden per misleidend antwoord.

Daarnaast blijkt dat one-shot prompting en finetuning de nauwkeurigheid van door de LLM gegenereerde foutuitleg niet statistisch significant verbeteren. One-shot prompting vermindert echter de hoeveelheid overbodige informatie, terwijl finetuning preciezere antwoorden oplevert. One-shot prompting genereert in 50,8% van de gevallen een correcte oplossing, geeft in 76,7%

¹Software Development Environment

een nuttig antwoord en levert in 23,5% een misleidende reactie. Dit leidt tot 3,29 behulpzame antwoorden per misleidend antwoord. Het gefinetuneerde model bereikt respectievelijk 52,5%, 71,7% en 28,3%, wat neerkomt op 2,53 behulpzame antwoorden per misleidend antwoord. Een χ^2 -test toont aan dat er geen statistisch significante verschillen bestaan met betrekking tot *het genereren van een correcte oplossing, het geven van een behulpzaam antwoord en het geven van een misleidend antwoord*.

Een andere studie [35] onderzoekt GPT-4 en evalueert de invloed van context (broncode in Java) op de kwaliteit van gegenereerde antwoorden. Wanneer alleen de foutboodschap wordt verstrekt, geeft het model in 86% van de gevallen een correcte uitleg, in 48% een correcte oplossing en in 100% een nuttig antwoord [35]. Met broncode verbeteren deze percentages naar respectievelijk 99% en 83% [35]. Bovendien genereert ChatGPT in 79% van de gevallen een perfecte oplossing (een uitleg met een fix die exact past binnen de context) wanneer de broncode wordt meegeleverd, tegenover slechts 11% zonder broncode. Het aantal te vage antwoorden daalt van 32% naar 1%. Deze resultaten tonen aan dat het verstrekken van broncode aan ChatGPT eenduidig betere antwoorden oplevert.

Verschillende studies [26, 14] bespreken potentiële toepassingen van ChatGPT voor debugging. ChatGPT blijkt in staat syntaxisfouten,² logische fouten³ en semantische fouten⁴ te detecteren. Daarnaast kan het model vragen beantwoorden met betrekking tot de syntaxis, structuur en het gedrag van code. Ook biedt ChatGPT uitleg over foutboodschappen, informatie over correct gebruik van functies en suggesties voor het oplossen of lokaliseren van bugs.

5.3 Debugstrategieën

De paper *How developers choose debugging strategies for challenging web application defects* [2, 3] onderzoekt welke debugstrategieën ontwikkelaars toepassen in uiteenlopende scenario's.⁵ Het onderzoek bestaat uit een survey onder 36 ontwikkelaars, gevolgd door semigestructureerde interviews met 16 ontwikkelaars.

Verschillende factoren maken het debugproces complexer: client-side code, prestatieproblemen, externe bibliotheken, verouderde code, onbekendheid met de code, de ontwikkelomgeving, niet-reproduceerbare bugs en compatibiliteitsproblemen.

De paper onderscheidt negen debugstrategieën: hypothesetesten, achterwaarts redeneren, voorwaarts redeneren, versimpelen, gebruik van foutmeldingen, binair zoeken, systeemniveau-debugging (zoals databasepermissies), raadplegen van externe bronnen en historische analyse. Enkele van deze strategieën worden hieronder nader toegelicht.

Bij de strategie **hypothesetesten** wordt een hypothese opgesteld over de mogelijke oorzaak van een bug, waarna bewijs wordt verzameld om deze hypothese te testen [2, 3]. Een gangbare methode is het doorlopen van de code om een mentaal model op te bouwen, wat helpt bij het formuleren van hypothesen [5]. Deze aanpak is vooral effectief bij functionele fouten. Ook intuïtie speelt soms een rol. Hypothesen worden onder meer getest via inputmanipulatie.

De strategie **achterwaarts redeneren** begint bij de onverwachte output en werkt terug door de code om de fout te achterhalen [5, 38, 23, 9, 1]. Deze aanpak blijkt bijzonder efficiënt. De metafoor *The snake in the grass* illustreert deze methode: bij het waarnemen van een fout wordt als het ware aan de staart van de slang getrokken om de bron ervan te onthullen [23]. Deze aanpak is gangbaar bij crashes.

Traditionele debuggers ondersteunen achterwaarts redeneren door breakpoints eerder in het programma te plaatsen en het programma opnieuw uit te voeren tot aan het betreffende punt

²Bijvoorbeeld een ontbrekende puntkomma of haakje.

³Bijvoorbeeld een verkeerd geïmplementeerde formule.

⁴Bijvoorbeeld een verkeerde parameter die aan een functie wordt meegegeven.

⁵Deze paragraaf bevat een samenvatting van een Engelstalige paper. Vertaling kan nuances wegnemen; raadpleeg bij voorkeur het origineel.

[9]. Voorwaarde is dat de software deterministisch werkt. Voorbeelden van toepassingen zijn *Whyline* [17] en *SPYDER* [1]. Nieuwere versies van SPYDER ondersteunen het uitvoeren van afzonderlijke codecellen in Jupyter-stijl en bieden de mogelijkheid om variabelewaarden op te slaan en later opnieuw te laden. Dit maakt achterwaarts debuggen mogelijk, zolang er een back-up van de status beschikbaar is [42].

Bij de strategie **voorwaarts redeneren** wordt stap voor stap de uitvoering van een falende test gevolgd om te begrijpen waar het misgaat [5, 33]. Breakpoints versnellen dit proces [38]. Deze methode is bijzonder effectief bij oneindige lussen en bij functionele fouten met onverwachte resultaten.

De strategie **versimpelen** richt zich op het vereenvoudigen van het probleem door irrelevante informatie te verwijderen en de code op te splitsen in kleinere segmenten, wat helpt bij het zoeken naar oplossingen via zoekmachines [5, 38, 44].

Bij **binair zoeken** wordt de code verdeeld in delen die afzonderlijk worden getest. Hierdoor wordt het probleem stapsgewijs geïsoleerd en opgelost [5, 38].

De keuze voor een debugstrategie hangt af van diverse contextuele factoren. Deze hebben betrekking op het defect, de code, organisatorische aspecten, beschikbare tools, persoonlijke voorkeuren en projectvereisten.

Voor het defect zelf zijn drie hoofdfactoren van belang:

- *Duidelijkheid*: de mate waarin foutboodschappen helder zijn, de context waarin de bug zich voordoet en de zekerheid dat sprake is van een daadwerkelijke bug.
- *Reproduceerbaarheid*: de mogelijkheid om het probleem consistent op te wekken, inclusief prestatie- en compatibiliteitsproblemen.
- *Root-oorsprong*: oorzaken gerelateerd aan netwerken, data, configuratie of hardware.

Codegerelateerde factoren zijn onderverdeeld in zes categorieën:

- *Bekendheid*: ervaring met vergelijkbare bugs en vertrouwdheid met de codebasis.
- *Toegang*: beschikbaarheid van de broncode, toegang tot documentatie of andere bronnen en de mate van toegang.
- *Technische factoren*: gebruikte technologieën en de ontwikkelomgeving.
- *Testbaarheid*: de mate waarin de code al door unittests wordt afgedekt.
- *Complexiteit*: de leesbaarheid van de code en afhankelijkheden tussen modules.
- *Onderhoudbaarheid*: omvang, ouderdom, technische schuld en aanwezigheid van verouderde componenten.

Daarnaast beïnvloeden organisatorische factoren de strategiekeuze. Denk aan kosten, goedkeuring door management, klantcontact, controle over de code en de beschikbaarheid van technische ondersteuning.

De beschikbaarheid van tools zoals debuggers, logsystemen, versiebeheersystemen, monitoring-tools en online simulators speelt eveneens een belangrijke rol.

Ook persoonlijke factoren, zoals ervaring en gewoontes, beïnvloeden de voorkeur voor bepaalde strategieën.

Ten slotte worden debugstrategieën beïnvloed door projectvereisten, zoals deadlines en de beschikbaarheid van indicatoren voor gewenst systeemgedrag.

Teaching Tech Together Het boek *Teaching Tech Together* [43] stelt dat vaardigheid in debuggen vaak samenhangt met goed programmeerwerk, maar dat het omgekeerde niet altijd

geldt. Professionals besteden 25% tot 60% van hun tijd aan debuggen. Bij beginners ligt dit percentage vaak nog hoger, wat het belang van deze vaardigheid onderstreept.

Goede debuggers maken vaak gebruik van symbolische debuggers, traceren handmatig programma's, schrijven tests en herlezen regelmatig de requirements. Beginners maken vaak de fout hetzelfde printstatement in beide delen van een `if-else`-constructie te plaatsen of zetten correct werkende code in commentaar om problemen te isoleren. Het boek benadrukt ook het belang van goed leesbare code voor effectief debuggen.

Volgens het boek vormt het kunnen traceren van code een kernvaardigheid. Dit omvat:

- het volgen van de uitvoeringsvolgorde van instructies
- het observeren van de waarden van variabelen tijdens uitvoering
- het uitvoeren van omgekeerde tracing: terugredeneren van output naar input

Om deze vaardigheden aan te leren, worden verschillende oefenvormen voorgesteld:

- *minimale fix-oefeningen*: corrigeren van kleine fouten in bestaande code
- *thema- en variatie-oefeningen*: aanpassen van correcte code om een andere uitkomst te verkrijgen
- *refactoring*: herschrijven van werkende code om deze leesbaarder of efficiënter te maken

Daarnaast helpt klassikale samenwerking, bijvoorbeeld door hardop denken tijdens het debuggen. Dit stimuleert het corrigeren van typefouten, het herkennen van foutieve aannames en het gezamenlijk analyseren van de probleemoplossing.

A Survey on Algorithmic Debugging Strategies De paper [37] onderzoekt het gebruik van algoritmische methoden voor het lokaliseren van bugs. Hierbij beantwoordt de gebruiker vragen om het algoritme te helpen de fout te vinden. Hoewel directe toepassing door studenten onwaarschijnlijk is, biedt deze aanpak waardevolle inzichten voor debugstrategieën in het onderwijs.

Elke deelfunctie wordt voorgesteld als een knoop in een graaf. Er worden diverse zoekstrategieën beschreven:

- *bottom-up search*: beginnen bij de laagste lagen van de code en naar boven werken
- *top-down search*: met varianten zoals:
 - *left-to-right*: een breadth-first search vanaf de hoofdfunctie
 - *zooming*: een depth-first search met twee benaderingen:
 - * eerst de volledige functie analyseren voor naar subfuncties over te schakelen
 - * direct overschakelen naar subfuncties (Deze variant wordt niet expliciet besproken in de paper.)

Daarnaast komen algoritmen aan bod waarbij gewichten aan knopen worden toegekend. Functies met een hoger gewicht — en dus hogere kans op fouten — worden eerder onderzocht. Een voorbeeld hiervan is het eerst doorlopen van de langste subboom.

Tot slot merkt de paper op dat studenten vaak op intuïtieve wijze debuggen, een aspect dat buiten het bereik van algoritmen valt.

Hoofdstuk 6

Debuggen met LLM's

6.1 Achtergrond

Net als het vorige onderdeel is ook dit gedeelte gericht op studenten die de introductie cursus Imperatief Programmeren volgen. *Prof. dr. Frank Neven*, de docent van dit vak, merkt op dat studenten vaak moeite hebben met debuggen — terwijl dit juist een essentieel onderdeel is van het programmeerproces. In dit deel van het onderzoek is onderzocht hoe LLM's (Large Language Models) ondersteuning kunnen bieden bij het verbeteren van deze vaardigheid.

Eerder onderzoek (zie hoofdstuk 5) toont aan dat LLM's beter presteren op het gebied van debuggen wanneer ze toegang hebben tot de broncode. Daarom is er in dit gedeelte gefocust op GitHub Copilot. Deze keuze is niet alleen gebaseerd op het feit dat Copilot toegang heeft tot de code van studenten, maar ook omdat studenten hier via de universiteit gratis gebruik van kunnen maken.

GitHub Copilot biedt de mogelijkheid om verschillende modellen te selecteren om antwoorden te genereren. Voor dit onderzoek is beperkt tot het gebruik van GPT-4o, aangezien dit het enige model was dat bij de start van het onderzoek niet meer in preview was.

Bij dit deel van het onderzoek is ervan uitgegaan dat de student de code geopend heeft in een IDE waarin GitHub Copilot is geïntegreerd, zodat het model toegang heeft tot de relevante broncode. Daarnaast is aangenomen dat de student vragen kan stellen aan Copilot in de vorm van een conversatie. Dit is onder andere mogelijk in **Microsoft Visual Studio Code**. Hoewel ook **JetBrains PyCharm** deze functionaliteit ondersteunt, was de chatfunctie in PyCharm bij aanvang van het onderzoek defect. Daarom is er gekozen voor Visual Studio Code. Bovendien gaan de lessen zelf ervan uit dat studenten met Visual Studio Code werken, wat een extra argument vormt om voor deze omgeving te kiezen in dit onderzoek.

6.2 Methode van aanpak verkennende studie

Als eerste stap zijn er concrete scenario's uitgewerkt op basis van de debugstrategieën uit *How developers choose debugging strategies for challenging web application defects* [2, 3]. Daarbij werden enkel voor de volgende strategieën geschikte scenario's geïdentificeerd: hypothesetesten, voorwaarts redeneren, achterwaarts redeneren, externe bronnen raadplegen, foutboodschappen gebruiken en simplificatie.

Voor de strategie **hypothesetesten** formuleert de student een hypothese over de oorzaak van een bug en legt deze voor aan GitHub Copilot. Het doel is dat Copilot de hypothese bevestigt, verworpt, of ondersteuning biedt bij het testen ervan.

Bij **voorwaarts redeneren** wordt GitHub Copilot gevraagd suggesties te geven voor het plaatsen van breakpoints, conditionele breakpoints of printstatements, zodat een specifieke variabele kan worden onderzocht. Hiermee ondersteunt Copilot de student bij het lokaliseren van de bug.

Ook bij de strategie **achtwaarts redeneren** stelt GitHub Copilot, op basis van een foutmelding of foutcode, geschikte breakpoints of conditionele breakpoints voor.

In het geval van **foutboodschappen** geeft de student een foutmelding door aan GitHub Copilot met de vraag om deze toe te lichten.

De strategie **externe bronnen** beschouwt een LLM zoals GitHub Copilot zelf als een externe hulpbron. Hier kan de student bijvoorbeeld vragen hoe de `len()`-functie in Python werkt. Anderzijds kan hij vragen of er een ingebouwde functie bestaat om de lengte van een string te bepalen — waarop Copilot zou moeten wijzen op de `len()`-functie.

Bij de strategie **simplificatie** neemt de LLM het versimpelen van code over. Dit houdt in dat overbodige code wordt verwijderd, zodat de student zich beter kan richten op de essentie van het probleem of eenvoudiger online naar een oplossing kan zoeken.

De hierboven beschreven strategieën vormen de basis voor een verkennend onderzoek, waarin wordt nagegaan wat goed werkt en wat minder effectief blijkt. Op basis van de resultaten zijn enkele strategieën geselecteerd voor verdere uitwerking.

Voor het opstellen van de prompts is, net als in het eerste deel van het onderzoek, gebruikgemaakt van het *CLEAR*-framework [24].

Om het potentieel van GitHub Copilot te beoordelen, is bij elke strategie een eenvoudige scoremetriek opgesteld. Deze is mogelijk niet volledig dekkend. Daarom worden ook aanvullende observaties en opmerkingen meegenomen. Op basis van de uitkomsten van dit verkennend onderzoek wordt bepaald welke strategieën verder onderzocht worden. De resultaten dienen tevens als input voor het ontwikkelen van een gedetailleerdere en verfijndere scoretabel die beter aansluit bij de aangepaste onderzoeksdoelen.

6.2.1 Datasets

Voor de start van dit onderdeel is gezocht naar een dataset met voorbeelden van Python-fouten op het niveau van een beginnende student. Daarnaast moest de dataset beperkt genoeg zijn om handmatig geëvalueerd te kunnen worden. Hoewel er meerdere datasets werden gevonden, bleken deze om uiteenlopende redenen ongeschikt. Sommige datasets waren bedoeld voor het trainen van machine learning-algoritmes en daardoor te omvangrijk. Andere datasets waren wel beperkt genoeg, maar bevatten voornamelijk fouten gerelateerd aan het gebruik van externe bibliotheken. Daarom is ervoor gekozen om zelf een geschikte dataset samen te stellen.

Dataset 1 Om een zo representatief mogelijke dataset op te stellen, is gebruikgemaakt van het artikel *Python Debugging Handbook – How to Debug Your Python Code* [15], waarin veelvoorkomende foutmeldingen worden besproken. Het artikel behandelt negen types errors: `SyntaxError`, `IndentationError`, `NameError`, `AttributeError`, `FileNotFoundError`, `IndexError`, `ImportError`, `TypeError` en `ValueError`. Voor elk van deze fouttypes is één voorbeeld uitgewerkt waarin de fout voorkomt. Deze voorbeelden zijn telkens gebaseerd op *Assignment 3* (zie Bijlage B).

Hoewel deze dataset beperkt is en onvoldoende om sluitende conclusies te trekken, biedt ze wel een indicatie van waar GitHub Copilot sterk in is, waar het nog moeilijkheden ondervindt, en waar het faalt. Op basis van deze eerste evaluatie kan worden beslist waarop verder gefocust wordt in het vervolgonderzoek.

Dataset 2 Niet alle debugstrategieën zijn even goed toepasbaar bij syntactische of runtime errors. Voor strategieën waarbij dat niet het geval is, is een tweede dataset samengesteld,

gebaseerd op het artikel *10 Logical Error Examples in Python* [40]. Dit artikel behandelt logische fouten, zoals:

- incorrecte loopconditie
- verkeerd geplaatste variabele-initialisatie
- foutieve vergelijkingsoperator¹
- off-by-one fouten
- verkeerde volgorde van operaties
- incorrect gebruik van logische operatoren
- foutieve stringvergelijkingen
- verkeerd begrijpen van lijstindexering
- onjuist gebruik van `return`-statements
- verkeerd gebruik van muteerbare standaardargumenten

Daarnaast zijn drie extra fouten toegevoegd: een oneindige lus, een fout waarbij `+1` foutief aan het resultaat wordt toegevoegd, en een verkeerde operator waarbij een vermenigvuldiging (`*`) wordt gebruikt in plaats van een machtsverheffing (`**`).

Dataset 3 Voor het deel externe bronnen was het de bedoeling dat er LLMs gebruikt werden om de werking van een functie uit te leggen of om functies voor te stellen bij bepaalde functionaliteit. Hiervoor is er dus een dataset met functies nodig. Het boek *De Programmeursleerling* [39], heeft als leidraad gediend om deze dataset op te bouwen. Deze dataset bevat alle onderdelen van python die vermeld waren in het boek. Deze dataset is groot genoeg om een gedetailleerde studie uit te voeren. Voor deze verkennende studie is er echter gekozen om een subset van deze dataset te gebruiken. Deze subset bevat tien elementen. De functies: `int()`, `round()`, `input()`, `math.log()` en `random.random()`. Daarnaast bevat deze dataset de `for-else` constructie, de `insert()` methode op lijsten, de `get()` methode op dictionaries, de `remove()` methode op sets en de `readline()` methode op bestanden.

Voor de strategie waarbij externe bronnen worden geraadpleegd, is het de bedoeling dat LLM's zoals GitHub Copilot gebruikt worden om functies toe te lichten of geschikte functies aan te bevelen voor een bepaalde taak. Hiervoor is een dataset nodig bestaande uit verschillende Python-functies en constructies. Het boek *De Programmeursleerling* [39] diende als basis voor deze dataset. Deze bevat alle relevante programmeerconstructies zoals beschreven in het boek.

Hoewel de volledige dataset voldoende omvangrijk is voor een diepgaand onderzoek, is voor deze verkennende studie gekozen voor een subset bestaande uit tien elementen:

- functies: `int()`, `round()`, `input()`, `math.log()`, `random.random()`
- constructies: `for-else`
- methodes: `insert()` op lijsten, `get()` op dictionaries, `remove()` op sets en `readline()` op bestanden

6.2.2 Verkennende studie foutboodschappen

Voor deze studie wordt dataset 1 gebruikt (zie paragraaf 6.2.1). In elk geval wordt de code uitgevoerd totdat Python een foutmelding genereert. Deze foutmelding wordt gekopieerd en ingevoerd in een nieuw chatvenster van GitHub Copilot. Voor iedere prompt wordt een afzonderlijke chat gestart. De gebruikte prompt is weergegeven in Figuur 6.1. De scoreverdeling is

¹Het bleek niet mogelijk om in Python een `=` te gebruiken in plaats van een `==` zonder een syntaxfout te veroorzaken. Het omgekeerde was wel mogelijk. Daarom is hier een `==` gebruikt waar eigenlijk een `=` verwacht werd.

als volgt: 50% indien een correcte uitleg wordt gegeven, en 100% indien ook een codevoorbeeld aanwezig is. De resultaten worden vervolgens gemiddeld.

```
<foutboodschap>
Wat betekent deze error?
```

Figuur 6.1: Dit is de prompt die is gebruikt om GitHub Copilot foutmeldingen te laten verklaren in de verkennende studie. <foutboodschap> wordt vervangen door de foutmelding zoals gegenereerd door Python.

6.2.3 Verkennende studie achterwaarts redeneren

Ook hier wordt dataset 1 gebruikt (zie paragraaf 6.2.1). Elke foutmelding die Python geeft na het uitvoeren van de code, wordt ingevoerd in een nieuwe chat met GitHub Copilot. De gebruikte prompt is te vinden in Figuur 6.2. De scoring is gebaseerd op de bruikbaarheid van voorgestelde breakpoints: 0% bij afwezigheid, 50% indien gedeeltelijk bruikbaar, en 100% bij volledig geschikte suggesties. Deze scores worden gemiddeld.

```
Gegeven dat dit mijn error message is:
<foutboodschap>
Waar zou jij breakpoints of conditionele breakpoints plaatsen, zodat
ik de bug kan vinden?
```

Figuur 6.2: Deze prompt is gebruikt om GitHub Copilot breakpoints te laten voorstellen bij achterwaarts redeneren. Hier wordt <foutboodschap> vervangen door de foutmelding van Python.

6.2.4 Verkennende studie voorwaarts redeneren

Voor deze studie wordt dataset 2 gebruikt (zie paragraaf 6.2.1). Na het uitvoeren van de code wordt het bekomen resultaat, samen met het resultaat van de referentieoplossing, ingevoerd in een nieuwe chat met GitHub Copilot. Twee prompts worden gebruikt: één voor standaardgevallen (zie Figuur 6.3) en één specifiek voor de oneindige lus (zie Figuur 6.4), aangezien deze geen resultaat oplevert.

```
De evaluate functie geeft steeds verkeerde resultaten. Dit zijn de
resultaten die ik krijg:
<resultaten>
Dit zijn de verwachte resultaten:
<verwachte resultaten>
Waar zou jij breakpoints of conditionele breakpoints plaatsen, zodat
ik de bug kan vinden?
```

Figuur 6.3: Deze prompt is gebruikt bij voorwaarts redeneren. Hier moeten <resultaten> en <verwachte resultaten> vervangen worden door respectievelijk de gegenereerde en de verwachte output.

De scoreverdeling verloopt als volgt: 100% wanneer het voorgestelde breakpoint direct leidt tot het vinden van de bug, 50% bij een indirecte of gedeeltelijke aanwijzing en 0% wanneer het niet tot een oplossing leidt. De gemiddelde score wordt berekend op basis van deze verdeling.

6.2.5 Verkennende studie externe bronnen

In dit deel wordt de chatfunctie van GitHub Copilot gebruikt in een leeg Python-bestand (*main.py*). Door dit bestand te benoemen als een Python-bestand, wordt aangenomen² dat

²Deze aanname is niet onderbouwd.

Ik krijg geen resultaten, noch eindigt het programma.
 Waar zou jij breakpoints of conditionele breakpoints plaatsen, zodat ik de bug kan vinden?

Figuur 6.4: Dit is de prompt die gebruikt is bij voorwaarts redeneren in het geval van een oneindige lus.

Copilot zich in een Python-context bevindt. Bij elke prompt wordt een nieuwe chat gestart. De gebruikte dataset is beschreven in paragraaf 6.2.1.

Wat doet doe functie: <functie naam>, en hoe gebruik ik hem?

Figuur 6.5: Deze prompt is gebruikt om functies te laten uitleggen. Hier wordt <functie naam> vervangen door de betreffende functie.

Wat doet de method: <method naam> in de context van <datatype>, en hoe gebruik ik hem?

Figuur 6.6: Deze prompt is gebruikt om methods van lijsten, dictionaries en sets te laten uitleggen. Hier moeten <method naam> en <datatype> worden aangepast naar respectievelijk de effectieve methode naam en het effectieve data type.

Voor het tweede onderdeel wordt een lijst van vragen gebruikt, weergegeven in Figuur 6.8.

De resultaten van beide onderdelen worden afzonderlijk beoordeeld en nadien gemiddeld, zodat beide even zwaar doorwegen. In het eerste onderdeel wordt 50% toegekend bij een correcte uitleg van de functie. Bij een aanvullend codevoorbeeld wordt een extra 50% toegekend. In het tweede onderdeel wordt 50% toegekend bij het correct identificeren van de juiste functie of methode. Bij een correcte uitleg komt daar nog eens 50% bij.

6.2.6 Verkennende studie hypothesetest

Deze studie maakt gebruik van dataset 1 en dataset 2 (zie respectievelijk paragraaf 6.2.1 en paragraaf 6.2.1). Voor elke fout worden zowel een correcte als een incorrecte hypothese getest. Telkens wordt een nieuwe chat gestart met GitHub Copilot. De gebruikte prompts zijn weergegeven in Figuur 6.9, Figuur 6.10 en Figuur 6.11, afhankelijk van het type fout.

De scoring werkt als volgt: 25 punten zijn gereserveerd voor het herkennen van de correcte hypothese. De overige 75 punten zijn gekoppeld aan de behandeling van de foutieve hypothese. Indien deze als fout wordt herkend, worden alle 75 punten toegekend. Wanneer de hypothese niet als fout wordt herkend, maar wel wordt verbeterd, volgt een score van 50. Als de foutieve hypothese wordt gevolgd, zonder correctie, volgt 0 punten. Het eindresultaat wordt gemiddeld.

6.2.7 Verkennende studie simplificatie

Voor deze studie blijkt het moeilijk om een geschikte dataset te vinden of aan te maken. Uiteindelijk wordt gekozen voor de referentieoplossing van opdracht 3 (zie Bijlage B). Verschillende prompts worden getest met uiteenlopende doelen, wat de beoordeling bemoeilijkt. Door het gebrek aan een betrouwbare meetmethode ontstaat een risico op onvoldoende onderbouwde resultaten. Om deze reden wordt dit onderdeel niet verder uitgewerkt of opgenomen in het uiteindelijke onderzoek.

Wat is de <constructie> constructie, en hoe gebruik ik het?

Figuur 6.7: Deze prompt is gebruikt om programmeerconstructies zoals de for-else uit te leggen. <constructie> wordt vervangen door de betreffende constructie.

- Heeft python een ingebouwde functie om een string om te vormen tot een integer?
- Heeft python een ingebouwde functie om een float af te ronden op 5 beduidende cijfers?
- Heeft python een ingebouwde functie om input op te vragen aan de gebruiker?
- Heeft python een ingebouwde functie om de 2 log van een getal te berekenen?
- Heeft python een ingebouwde functie om een random getal tussen 0 en 1 terug te geven?
- Heeft python een ingebouwde constructie waarbij ik ervoor kan zorgen dat code enkel uitgevoerd wordt als mijn for loop niet vroegtijdig onderbroken wordt?
- Hebben lijsten in python een ingebouwde method om een waarde toe te voegen op positie 12?
- Hebben dictionaries in python een ingebouwde method om de waarde achter key 'naam' te verkrijgen?
- Hebben sets in python een ingebouwde method om element '3' uit de set te verwijderen?
- Hebben bestandsobjecten in python een ingebouwde method om een regel in te lezen?

Figuur 6.8: Overzicht van gebruikte prompts in het tweede onderdeel.

6.3 Resultaten verkennende studie

6.3.1 Verkennende studie foutboodschappen

Voor elk van de foutboodschappen biedt GitHub Copilot een correcte uitleg en een voorstel tot oplossing. Dit resulteert in een score van 100%.

Bij zowel `NameError` als `SyntaxError` verschijnt in de foutmelding reeds een suggestie van Python. GitHub Copilot verklaart de fout en licht toe waarom de voorgestelde oplossing werkt. In het geval van `SyntaxError` werd `=` gebruikt in een `if`-statement, in plaats van `==`. Copilot verduidelijkt het verschil tussen beide. Voor `NameError` werd een underscore weggelaten uit een functienaam, wat leidt tot een niet-herkenbare referentie. Copilot identificeert de correcte naam en verklaart het probleem. In beide gevallen stelt Copilot een één-klik-oplossing voor.

Voor `IndentationError`, `AttributeError` en `IndexError` worden eveneens correcte oplossingen gegenereerd. Bij `IndentationError` stelt Copilot voor om overbodige spaties te verwijderen. Bij `AttributeError`, waarbij `s.len()` werd gebruikt, wordt de correcte syntaxis `len(s)` voorgesteld. In het geval van `IndexError` werd getest of index 0 beschikbaar is, terwijl vervolgens index 10 werd opgevraagd. Copilot suggereert een aanpassing van index 10 naar index 0. Daarbij wordt expliciet vermeld dat dit 'waarschijnlijk' de bedoelde actie is — een belangrijke nuance die aangeeft dat controle door de gebruiker noodzakelijk blijft.

Bij `TypeError` worden twee argumenten meegegeven aan een functie die slechts één verwacht. Copilot verklaart het verwachte gebruik en suggereert dat het tweede argument "waarschijnlijk onjuist is".

Voor `FileNotFoundError` wordt een bestand opgevraagd via een relatief pad dat niet bestaat. Copilot adviseert om de bestandslocatie te controleren en, indien nodig, een absoluut pad te gebruiken.³ Copilot biedt bovendien debughulp aan in de vorm van code die de werkdirectory toont, wat nuttig is voor padverificatie.

Voor `ImportError`, veroorzaakt door het importeren van 'referenti' in plaats van 'referentie', ontstaat een `ModuleNotFoundError`.⁴ Copilot bespreekt mogelijke oorzaken, zoals typfouten

³Een absoluut pad is vaak niet wenselijk vanwege de verminderde portabiliteit. Beter zou zijn als Copilot uitlegt hoe naar een parent-directory te navigeren via `../` of naar een submap met `./child/file.txt`

⁴Er was een bestand `referentie.py` aanwezig in dezelfde directory.

```
Ik krijg deze foutboodschap:
<error message>
Ik denk dat dit het probleem is:
<hypothese>
Denk jij dat mijn hypothese correct is?
```

Figuur 6.9: Dit is de prompt die gebruikt is voor het testen van hypothesen bij crashes. Hierbij wordt `<error message>` vervangen door de foutboodschap zoals gegenereerd door Python, en `<hypothese>` door de te testen hypothese.

```
De evaluate functie geeft steeds verkeerde resultaten.
Dit zijn de resultaten die ik krijg:
<resultaten>
Dit zijn de verwachte resultaten:
<verwachte resultaten>
Ik denk dat dit het probleem is:
<hypothese>
Denk jij dat mijn hypothese correct is?
```

Figuur 6.10: Dit is de prompt die gebruikt is bij logische fouten in dataset 2. Hierbij wordt `<resultaten>` vervangen door de gegenereerde uitvoer, `<verwachte resultaten>` door de verwachte uitvoer, en `<hypothese>` door de veronderstelde oorzaak van de fout.

of ontbrekende installaties, en stelt het juiste `pip`-commando voor. Ook wordt aangeraden te controleren of het bestand zich in dezelfde map bevindt als het script.

Bij `ValueError`, waarbij `int("3 + 5")` werd toegepast, verklaart Copilot het correcte gebruik van de `int`-functie. Daarnaast wordt een alternatieve oplossing voorgesteld waarbij de string rechtstreeks gebruikt wordt, wat in deze context correct is.

6.3.2 Verkennende studie achterwaarts redeneren

Niet elk fouttype is geschikt voor achterwaarts redeneren. Vooral `IndentationError`, `NameError`, `SyntaxError`, `FileNotFoundError` en `ImportError` ontstaan rechtstreeks op de locatie van de fout. Daardoor is achterwaarts redeneren van beperkte meerwaarde.

Bij `IndentationError` en `SyntaxError` worden weliswaar meerdere breakpoints voorgesteld, maar deze zijn weinig relevant. Voor `FileNotFoundError` en `ImportError` zijn de voorgestelde breakpoints correct, maar afgeleid van de stack trace. Bij `NameError` ligt de focus van Copilot op uitleg en oplossingen, niet op het vinden van een geschikt breakpoint.

Deze beperkingen zijn logisch. Bij fouten zoals syntaxis- of importfouten is er weinig reden om het programma achterwaarts te analyseren; de oorzaak ligt voor de hand.

Voor `IndexError`, `TypeError` en `ValueError` stelt Copilot wél geschikte breakpoints voor, telkens met een duidelijke uitleg over waar aandacht aan besteed moet worden. Zo helpt bij `IndexError` inzicht in de lijstinhoud, bij `ValueError` de inputwaarde, en bij `TypeError` het type van de meegegeven argumenten. Deze breakpoints worden doorgaans overgenomen uit de stack trace.

Bij `AttributeError` worden eveneens relevante breakpoints voorgesteld. Copilot benadrukt het belang van het controleren van het variabeltype. De fout ontstaat doordat `len()` op een string wordt toegepast, wat incorrect is. Toch kan deze fout begrijpelijk zijn als de student dacht met een objecttype te werken. De voorgestelde breakpoints zijn dus nuttig bij het achterhalen van het foutieve typegebruik.

De uiteindelijke score bedraagt 78%, met twee ongeschikte en één ontbrekende breakpoint.

```
Ik krijg geen resultaten, noch eindigt het programma.
Ik denk dat dit het probleem is:
<hypothese>
Denk jij dat mijn hypothese correct is?
```

Figuur 6.11: Dit is de prompt die gebruikt is voor het testen van hypothesen bij oneindige lussen. Hierbij wordt `<hypothese>` vervangen door de te verifiëren hypothese.

6.3.3 Verkennende studie voorwaarts redeneren

In negen van de dertien scenario's leidt het voorgestelde breakpoint rechtstreeks tot de bug. In één geval wijst het naar de verantwoordelijke functie, maar niet naar de exacte foutregel. In een ander geval leidt het onrechtstreeks tot het identificeren van het probleem. Slechts in twee gevallen zijn de voorgestelde breakpoints niet bruikbaar.

Copilot biedt niet enkel breakpoints aan, maar legt ook uit waarop gelet moet worden. Dit versterkt de bruikbaarheid van de suggesties, wat resulteert in een score van 77%.

Bij fouten zoals *extra operatie* en *lijstindexering verkeerd begrijpen* werd de bug niet gevonden. In het laatste geval bevond de bug zich in de testfunctie, niet in de hoofdcode.

Voor fouten zoals *incorrecte vergelijkingsoperator*, *loopconditie*, *operatievolgorde*, *stringvergelijking*, *logische operator* en *off-by-one* wordt de bug telkens correct gelokaliseerd.

Bij *verkeerd gebruik van return statements* leidt de voorgestelde breakpoint niet rechtstreeks tot de bug, maar een extra stap met een stepper zou dit verhelpen.

In het geval van een *oneindige lus* identificeert Copilot de juiste lus en stelt meerdere breakpoints voor. Hierbij wordt specifiek gefocust op de betrokken variabele.

Bij *verkeerd geplaatste variabele-initialisatie* wordt de juiste functie geïdentificeerd, maar niet de exacte regel.

Voor *foutief gebruik van muteerbare standaardargumenten* en *verkeerde operator* leidt het breakpoint niet alleen tot de bug, maar wordt ook extra informatie gegeven over onderliggende inconsistenties.

6.3.4 Verkennende studie externe bronnen

Dit onderdeel bestaat uit twee luiken. In het eerste deel wordt GitHub Copilot gevraagd om functies toe te lichten. Bij het uitvoeren van de prompts geeft Copilot telkens een nauwkeurige en volledige uitleg van de betreffende functie, methode of constructie. De tool beschrijft wat de functie doet, hoe deze gebruikt moet worden en illustreert dit met codevoorbeelden. In veel gevallen vermeldt Copilot ook mogelijke exceptions die kunnen optreden, hoewel dit niet consistent gebeurt. Deze resultaten leiden tot een score van 100%.

In het tweede deel wordt een bepaald gedrag beschreven, met de vraag of er een ingebouwde functie bestaat die dit gedrag implementeert. Ook hier levert Copilot steeds een correct antwoord. De tool stelt de juiste functie voor,⁵ licht de werking ervan toe, en geeft relevante codevoorbeelden. Vaak worden ook mogelijke exceptions en hun afhandeling besproken. Indien pre-processing nodig is, zoals bij de `round()` functie, wordt dit eveneens vermeld. Verder worden randgevallen toegelicht, zoals bij de `insert()` methode, waar vermeld wordt dat een index groter dan de lijstlengte resulteert in toevoeging aan het einde van de lijst in plaats van op de opgegeven positie. Dit leidt tot een score van 100% voor zowel dit onderdeel als het totaal.

⁵Bijvoorbeeld, wanneer gevraagd wordt naar de 2-logaritme van een getal, stelt Copilot de `log2()` functie voor in plaats van `log()` met 2 als parameter. Hoewel dit niet exact de gevraagde functie is, blijkt deze correcter en beter geschikt binnen de context.

6.3.5 Verkennende studie hypothesetest

Wanneer een correcte hypothese wordt ingevoerd, herkent GitHub Copilot deze telkens als juist. Vervolgens legt de tool uit welke stappen genomen moeten worden, ondersteund met codevoorbeelden. De omvang van deze voorbeelden varieert: soms volstaat één regel, in andere gevallen wordt een volledige functie weergegeven. Daarnaast volgt een korte uitleg over wat er fout gaat en waarom. In alle gevallen wordt de volledige 25% van de score behaald die toegekend wordt aan het correct herkennen van een juiste hypothese.

Bij foutieve hypothesen hangt het resultaat af van het type fout. Indien er een crash plaatsvindt en het bijhorende crashrapport wordt meegegeven, herkent Copilot de hypothese als foutief en geeft dit expliciet aan. Vervolgens verduidelijkt de tool wat er werkelijk gebeurt. In specifieke gevallen is een kanttekening nodig. Bij een `IndentationError` wordt foutief gesuggereerd dat er te weinig spaties zijn, terwijl het probleem net een spatie te veel betreft. Copilot bevestigt onterecht de hypothese, maar geeft wel de correcte oplossing. Bij een `ImportError` wordt gesuggereerd dat de module niet geïnstalleerd is, terwijl het probleem een typfout blijkt te zijn in de bestandsnaam. Copilot biedt in dat geval een duidelijke uitleg over het installeren van een module, inclusief het gebruik van het juiste `pip install` commando. Daardoor wordt in slechts zeven van de negen gevallen de resterende 75% van de score toegekend. Eén geval leidt tot een score van 50%, en één geval tot 0%.

In het geval van logische fouten herkent Copilot in twee gevallen een foutieve hypothese onterecht als correct. In drie gevallen wordt de fout correct herkend. In de overige acht gevallen beschouwt Copilot de foutieve hypothese als correct, maar geeft vervolgens toch een correcte oplossing die afwijkt van de hypothese. Bij nader onderzoek blijkt dat Copilot deze afwijkende maar juiste oplossing vooral geeft wanneer de hypothese sterk lijkt op het werkelijke probleem. Bijvoorbeeld, indien het gedrag overeenkomt maar een ander deel van de functie de fout veroorzaakt. Echter, bij foutieve vergelijkingsoperatoren of incorrecte loopcondities treedt Copilot volledig buiten de correcte oplossingsruimte; hier wordt een oplossing voorgesteld die de foute hypothese volgt en daardoor volledig incorrect is. Dit resulteert in drie gevallen waarin 75% van de score wordt toegekend, acht keer 50% en twee keer 0%.

Wanneer deze resultaten worden samengevoegd, bedraagt de totale score 80%.

6.4 Conclusie verkennende studie

6.4.1 Verkennende studie foutboodschappen

De resultaten in deelparagraaf 6.3.1 tonen een duidelijk patroon waarin GitHub Copilot foutboodschappen op correcte wijze uitlegt. Daarnaast biedt Copilot telkens een correcte oplossing aan. Het onderzochte bestand bevat slechts 134 regels, inclusief witregels, wat de complexiteit van de test beperkt. Toch wijst het resultaat op potentieel voor het inzetten van de chatfunctie van GitHub Copilot bij het verklaren van foutboodschappen. In dit stadium van het onderzoek wordt nog geen definitieve conclusie getrokken over de algemene toepasbaarheid van deze functie.

6.4.2 Verkennende studie achterwaarts redeneren

De resultaten in deelparagraaf 6.3.2 wijzen erop dat deze aanpak weinig perspectief biedt. Ondanks een eindscore van 78% blijkt deze score misleidend. GitHub Copilot stelt vaak geschikte breakpoints voor, maar doet dit voornamelijk op basis van de stack trace. In de gevallen waar Copilot faalt, ontbreekt een patroon of logica waarop gestuurd kan worden. Aangezien de voorgestelde breakpoints rechtstreeks uit de stack trace komen, voegt deze functie weinig toe. Studenten kunnen dergelijke informatie immers ook rechtstreeks uit de foutboodschap halen. Om die reden wordt dit onderdeel niet verder onderzocht en ligt de focus op methoden die meerwaarde bieden.

6.4.3 Verkennende studie voorwaarts redeneren

De resultaten in deelparagraaf 6.3.3 tonen aan dat GitHub Copilot bij het oplossen van logische fouten vaak effectieve breakpoints voorstelt. Deze breakpoints leiden in veel gevallen rechtstreeks tot het identificeren van de onderliggende bug. Bovendien biedt Copilot bij deze suggesties extra toelichting over waar precies op gelet moet worden tijdens het debuggen. Dit is bijzonder waardevol, aangezien logische fouten doorgaans geen foutboodschap genereren die de precieze locatie van het probleem aangeeft. De aanpak vertoont dus potentieel voor ondersteuning bij foutanalyse zonder expliciete aanwijzingen. Tegelijkertijd blijft voorzichtigheid geboden; op basis van deze verkennende studie kan nog geen definitieve uitspraak worden gedaan over de structurele toepasbaarheid van deze methode in bredere contexten.

6.4.4 Verkennende studie externe bronnen

Wanneer de resultaten uit beide onderdelen van de verkennende studie over externe bronnen bekeken worden (zie deelparagraaf 6.3.4), komen positieve resultaten naar voren. In beide onderdelen geeft GitHub Copilot telkens een uitstekend resultaat. De antwoorden zijn volledig en van hoge kwaliteit. Deze consistentie maakt het moeilijk om tot een andere conclusie te komen dan dat hier potentieel in zit. Tegelijkertijd moet opgemerkt worden dat in dit deel van het onderzoek (nog) geen concrete conclusie getrokken kan worden dat Copilot dit zonder meer kan. De resultaten lijken daar echter wel op te wijzen.

6.4.5 Verkennende studie hypothesetest

De resultaten in deelparagraaf 6.3.5 geven aan dat bij een correcte hypothese ook een correct antwoord volgt. In gevallen waarin een foutieve hypothese wordt ingevoerd, en een crashrapport aanwezig is, wordt de fout meestal correct herkend. Ook dan volgt vaak een goed antwoord dat het probleem oplost, of in het geval van een ontbrekend package, een oplossing die zou werken indien het package bestond. Bij logische fouten levert Copilot over het algemeen eveneens bruikbare oplossingen. Wel blijkt dat Copilot soms te snel bevestigt dat de hypothese juist is, ook als deze fout blijkt. Dit kan verwarrend zijn, zeker wanneer toch de juiste oplossing wordt gegeven. In twee gevallen wordt zelfs de foute hypothese gevolgd, wat problematisch is. Alles bij elkaar genomen toont deze aanpak potentieel, vooral bij fouten die leiden tot crashes.

6.5 Methode van aanpak

Op basis van het verkennend onderzoek wordt ervoor gekozen om de verdere analyse te starten met foutboodschappen. Vervolgens komen externe bronnen aan bod, daarna hypothesetoetsing, en tot slot voorwaarts redeneren. Deze sectie bespreekt de methode van aanpak die wordt gehanteerd bij het verfijnde onderzoek binnen elk van deze categorieën.

Voor elk onderdeel wordt een meer verfijnde prompt opgesteld op basis van de resultaten uit het verkennend onderzoek. In sommige gevallen wordt aanvullend onderzoek uitgevoerd in de vorm van prompt engineering. Daarbij wordt telkens de prompt geselecteerd waarvan het gegenereerde antwoord inhoudelijk het dichtst aansluit bij het gewenste resultaat. In deze fase wordt nog geen scoreverdeling toegepast, om bias bij de promptselectie te vermijden.

Wanneer een minder geschikt antwoord hogere scores behaalt op basis van een niet-optimale scoreverdeling, bestaat het risico dat een suboptimaal prompt gekozen wordt. Om dit te voorkomen, gebeurt de selectie eerst op inhoudelijke overeenkomst. Pas nadat de meest geschikte prompt is vastgesteld wordt de scoreverdeling toegepast op de volledige datasets.

6.5.1 Foutboodschappen

Uit het verkennend onderzoek blijkt dat GitHub Copilot vaak antwoorden suggereert in plaats van enkel uitleg te geven. Dit wordt beschouwd als ongewenst gedrag en is daarom opgenomen

in de verfijnde puntenverdeling onder de categorie *integriteit* (zie Tabel 6.1).

Voor de categorie *accuraatheid* wordt nagegaan of de uitleg feitelijk correct is. Zowel *accuraatheid* als *integriteit* krijgen elk 25 punten toegekend, omdat deze aspecten als het belangrijkste worden beschouwd. De categorie *volledigheid* evalueert of de uitleg alle noodzakelijke elementen bevat en krijgt 20 punten toegekend. Voor *duidelijkheid*, die beoordeelt in hoeverre de uitleg helder en begrijpelijk is, worden eveneens 20 punten toegekend. Tot slot krijgt de categorie *structuur*, die nagaat of het antwoord logisch is opgebouwd (inclusief eventuele codevoorbeelden), 10 punten. Deze weging is lager, omdat structuur als minder doorslaggevend wordt beschouwd.

Categorie	Score
Accuraatheid	/25
Duidelijkheid	/20
Volledigheid	/20
Structuur	/10
Integriteit	/25
Totaal	/100

Tabel 6.1: scoreverdeling voor foutboodschappen

In het verkennend onderzoek krijgt het geven van een codevoorbeeld een gewicht van 50%. Deze keuze is oorspronkelijk gemaakt omdat een goed gekozen codevoorbeeld een krachtige vorm van verduidelijking kan zijn. Tegelijkertijd leidt het echter tot het risico dat GitHub Copilot onbedoeld het juiste antwoord voorgerekend aanbiedt. Om dit evenwichtiger te benaderen, wordt in de verfijnde evaluatie niet langer een aparte score toegekend aan het al dan niet geven van code, maar wordt dit mee opgenomen in de beoordeling van de structuur. Codevoorbeelden blijven dus toegelaten, maar hun aanwezigheid wordt enkel positief beoordeeld als ze bijdragen aan de helderheid en opbouw van het antwoord, zonder expliciete oplossingen te suggereren.

Voor de categorie *accuraatheid* worden 25 punten toegekend wanneer het antwoord volledig vrij is van feitelijke fouten of misinformatie. Zodra het antwoord incorrecte of misleidende informatie bevat, worden 0 punten toegekend.

De categorie *duidelijkheid* evalueert in hoeverre de uitleg begrijpelijk is. Een heldere uitleg levert 20 punten op. Indien het antwoord verwarrend geformuleerd is, worden 10 punten toegekend. In het geval van een totaal onduidelijk antwoord worden geen punten toegekend.

Bij *volledigheid* wordt beoordeeld of het antwoord alle noodzakelijke informatie bevat. Een volledig antwoord levert 20 punten op. Indien bepaalde elementen ontbreken, maar met een follow-upvraag eenvoudig kunnen worden aangevuld, worden 10 punten toegekend. Als de ontbrekende informatie essentieel is en niet eenvoudig te reconstrueren valt, worden 0 punten gegeven.

De categorie *structuur* beoordeelt of het antwoord logisch en overzichtelijk is opgebouwd. Een goed gestructureerd antwoord, eventueel ondersteund door een relevant codevoorbeeld, krijgt 10 punten. Als de structuur licht verbeterd kan worden, maar het antwoord bruikbaar blijft, worden 5 punten toegekend. Bij ernstige structurele problemen, waarbij het antwoord nauwelijks bruikbaar is, worden 0 punten gegeven.

Tot slot wordt onder *integriteit* geëvalueerd of het antwoord autonoom tot inzicht leidt, zonder dat GitHub Copilot expliciete oplossingen of suggestieve hints geeft. Indien het antwoord wordt verklapt of een duidelijke hint wordt gegeven, worden 0 punten toegekend. Wanneer GitHub Copilot enkel informatie herhaalt die reeds zichtbaar is in de foutboodschap of de stacktrace van Python, en er geen extra richting wordt gegeven, worden alsnog 25 punten toegekend. Deze gevallen worden apart geregistreerd, zodat ze transparant kunnen worden onderscheiden in de analyse.

De gebruikte dataset voor dit onderdeel bestaat uit dataset 1 uit Figuur 6.9, aangevuld met één oefening uit elk hoofdstuk (vanaf hoofdstuk 4) van de Dodona-cursus [29], alsook met alle programmeeropdrachten. De oefeningen worden willekeurig geselecteerd. Eventuele overlapping met de oefeningen vermeld in Deel I is toeval. Voor elke oefening wordt een bijpassende crash toegevoegd die realistisch kan optreden bij het implementeren van de oplossing.

```
<foutboodschap>
Wat betekent deze error?
Geeft geen oplossing, ik wil dit namelijk zelf vinden.
```

Figuur 6.12: Deze prompt is gebruikt om GitHub Copilot een foutboodschap te laten verklaren in de gedetailleerde studie. De placeholder `<foutboodschap>` wordt vervangen door de werkelijke foutmelding zoals gegenereerd door Python.

De gebruikte prompt is weergegeven in Figuur 6.12. Deze prompt is een uitbreiding van de eerder gebruikte prompt in het verkennend onderzoek (Figuur 6.1), met als toevoeging de expliciete vraag om geen oplossing te geven. Aangezien deze prompt effectief blijkt te zijn, wordt geen alternatief onderzocht.

6.5.2 Externe bronnen

Bij het gebruik van externe bronnen wordt de Large Language Model (LLM), in dit geval GitHub Copilot, ingezet als assistent bij het verklaren van programmeerconstructies. Deze evaluatie bestaat uit twee delen.

In het eerste deel start de analyse met 241 items, waaronder keywords,⁶ functies, methodes, constructies⁷ en operatoren. De verzameling bevat onder andere, maar is niet beperkt tot, voorgedefinieerde class-methodes, binaire operatoren, datastructuren zoals sets, dictionaries, lists, en strings, bestandsbewerking, en standaardbibliotheken zoals `os`, `os.path`, `random`, `math`, `statistics`, `glob`, `urllib`, `collections`, `json`, `pickle`, `csv`, `time`, `datetime` en `itertools`. Ook veelgebruikte basisfuncties en constructies zoals de `while`- en `for`-lussen, de `for-else`- en `if-else`-structuur worden opgenomen. Het doel hierbij is dat GitHub Copilot een duidelijke en correcte uitleg formuleert.

Categorie	Score
Accuraatheid	/35
Duidelijkheid	/25
Volledigheid	/25
Structuur	/15
Totaal	/100

Tabel 6.2: scoreverdeling voor externe bronnen — verklarende functieprompts

Tabel 6.2 toont de scoreverdeling voor deze evaluatie.

Accuraatheid wordt beoordeeld op een binaire schaal: ofwel is de uitleg volledig correct en worden 35 punten toegekend, ofwel is de uitleg incorrect en worden 0 punten toegekend. Aangezien accuraatheid de basis vormt voor de verdere beoordeling, wordt bij een fout antwoord expliciet vermeld dat de andere categorieën niet van toepassing zijn.

Voor *duidelijkheid* worden maximaal 25 punten toegekend. Een volledig duidelijke uitleg krijgt het maximum. Indien de uitleg deels verwarrend is, bedraagt de score 15 punten. Volledige onduidelijkheid resulteert in 0 punten.

⁶Keywords verwijzen naar gereserveerde woorden in Python, zoals het woord `def`.

⁷Constructies omvatten bijvoorbeeld het `if-else` statement of een `while`-lus.

De *structuur* van het antwoord kan maximaal 15 punten opleveren. Een goed gestructureerd antwoord ontvangt het maximum, bij lichte verbeterpunten worden 7 punten toegekend, en bij een slecht gestructureerde uitleg, die daardoor onbruikbaar is, wordt 0 punten toegekend.

De *volledigheid* wordt eveneens op 25 punten beoordeeld. Een volledig antwoord, inclusief uitleg van alle relevante fouten en randgevallen, ontvangt de hoogste score. Als sommige fouten ontbreken, worden 20 punten toegekend. Bij het ontbreken van uitleg over parameters, maar met correcte functietoepassing, worden 15 punten gegeven. Als slechts een functie met codevoorbeeld wordt gegeven zonder verdere uitleg, bedraagt de score 10 punten. Een onvolledige uitleg zonder bruikbare informatie leidt tot 0 punten.

Wat doet de functie: <functie naam>, en hoe gebruik ik hem?

Figuur 6.13: Deze prompt is gebruikt voor het verklaren van functies in de gedetailleerde studie. Hier wordt <functie naam> vervangen door de concrete functienaam.

Wat doet de method: <method naam>, in de context van <datatype>, en hoe gebruik ik hem?

Figuur 6.14: Deze prompt is gebruikt voor het verklaren van methodes. Hier moeten <method naam> en <datatype> worden ingevuld met de betreffende gegevens.

Wat is de <constructie> constructie, en hoe gebruik ik het?

Figuur 6.15: Deze prompt is gebruikt om GitHub Copilot een programmeerconstructie te laten verklaren. Hierbij wordt <constructie> vervangen door de specifieke constructie.

Voor de verklaring van functies wordt steeds de prompt in Figuur 6.13 gebruikt. Indien de functie deel uitmaakt van een bibliotheek zoals `math`, wordt de naam van de bibliotheek gevolgd door een punt en de functienaam gebruikt, bijvoorbeeld `math.log()`. Voor methodes geldt de prompt uit Figuur 6.14, voor constructies Figuur 6.15, operatoren Figuur 6.16, en voor keywords Figuur 6.17. Voor reguliere expressies wordt de prompt *Hoe vorm ik regular expressions?* gebruikt, aangezien dit type niet binnen de eerder gedefinieerde categorieën valt.

In het tweede deel wordt een korte beschrijving als uitgangspunt genomen. Hierna wordt onderzocht of GitHub Copilot de bijbehorende functie correct weet te suggereren. De niet-standaard bibliotheek `pcinput` is uit de dataset verwijderd, wat resulteert in een uiteindelijke dataset van 237 items.

Initieel werd de prompt uit Figuur 6.18 gebruikt. Al snel werd echter duidelijk dat deze prompt niet geschikt was, aangezien het enkel de functie en een codevoorbeeld opleverde zonder verdere toelichting. Na enige prompt engineering bleek het aangepaste prompt in Figuur 6.19 beter te functioneren.

Voor de puntentoekenning wordt gebruikgemaakt van Tabel 6.3. *Accuraatheid* wordt hierbij beoordeeld aan de hand van een binaire score: ofwel is de uitleg volledig correct en worden 35 punten toegekend, ofwel is deze onjuist en volgt een score van 0 punten. Wanneer de accuraatheid 0% bedraagt, heeft de verdere beoordeling weinig waarde. In dat geval wordt dit expliciet vermeld.

Voor *duidelijkheid* geldt een maximumscore van 25 punten. Indien de uitleg helder en goed te volgen is, worden alle punten toegekend. Bij een enigszins verwarrende uitleg wordt 15 punten toegekend, en bij een volledig onduidelijke uitleg bedraagt de score 0 punten.

Wat betreft *structuur* kan maximaal 15 punten worden behaald. Een duidelijke en goed opgebouwde uitleg leidt tot de volledige score. Indien slechts een beperkte verbetering nodig is, worden 7 punten toegekend. Wanneer de structuur dermate gebrekkig is dat het antwoord onbruikbaar wordt, volgt een score van 0 punten.

Wat is de <operator> operator, en hoe gebruik ik het?

Figuur 6.16: Deze prompt is gebruikt om GitHub Copilot een operator te laten verklaren. Hierbij wordt <operator> vervangen door de specifieke operator.

Wat is het <keyword> keyword, en hoe gebruik ik het?

Figuur 6.17: Deze prompt is gebruikt om GitHub Copilot een keyword te laten verklaren. Hierbij wordt <keyword> vervangen door het specifieke keyword.

Voor *volledigheid* wordt een maximum van 25 punten toegekend indien de uitleg volledig is. Wanneer slechts een foutgeval of edge case ontbreekt, wordt 20 punten toegekend. Indien de functie wel wordt toegelicht, maar niet alle parameters worden besproken, bedraagt de score 15 punten. Wanneer enkel een codevoorbeeld wordt gegeven zonder verdere toelichting, worden 10 punten toegekend. Indien de functie ontbreekt of de gegeven uitleg onvoldoende is om deze correct te gebruiken, wordt 0 punten toegekend.

6.5.3 Hypothesetest

In dit onderdeel wordt GitHub Copilot een hypothese voorgelegd, die vervolgens geëvalueerd wordt. De hypothesetest is onderverdeeld in twee delen.

Het eerste deel behandelt syntactische fouten. Hiervoor wordt gebruikgemaakt van Dataset 1, weergegeven in paragraaf 6.2.1, bestaande uit negen entries. Daarnaast worden per hoofdstuk ook oefeningen uit Dodona gebruikt, aangevuld met twee extra entries, zoals beschreven in deelparagraaf 6.5.1. In totaal zijn er 18 entries voor dit deel.

Het tweede deel richt zich op logische fouten. Hier wordt Dataset 2 (met 13 entries) gebruikt. Die is te zien in paragraaf 6.2.1. Bovendien wordt een bijkomende dataset opgebouwd waarin opzettelijk logische fouten worden geïntroduceerd in oefeningen uit Dodona, telkens één oefening per hoofdstuk. Alle hoofdstukken worden hierbij meegenomen, wat resulteert in 17 entries.

Voor elke entry wordt zowel een correcte als een foute hypothese getest. Dit is essentieel. Het is niet enkel van belang dat Copilot een correcte hypothese kan bevestigen. Het model mag ook geen oplossing vrijgeven wanneer een foute hypothese wordt voorgelegd. Wel mag het model de student in de juiste richting sturen.

De scoreverdeling voor de hypothesetest is te zien in Tabel 6.4. Deze toont welk percentage van de totaalscore aan elke categorie wordt toegekend. Daarnaast worden de te behalen scores voor de juiste en foute hypothese apart vermeld. Merk op dat bij *accuraatheid* en *integriteit* de foute hypothese net iets zwaarder doorweegt dan de juiste hypothese.

Voor *integriteit* komt dit doordat het belangrijker is dat er niets wordt voorgezegd aan de student bij een foute hypothese dan bij een juiste. Wanneer een student een foute hypothese heeft, kent hij het juiste antwoord nog niet. Bij een juiste hypothese weet hij het antwoord wel al.

Voor *accuraatheid* geldt dat wanneer een student een foute hypothese heeft en Copilot bevestigt dat deze correct is, het veel moeilijker wordt om de juiste hypothese te vinden. Terwijl wanneer een student een juiste hypothese heeft en Copilot aangeeft dat deze fout is, de student genoodzaakt wordt om grondiger na te denken. Dit kan ertoe leiden dat hij alsnog tot de correcte conclusie komt.

Aan *accuraatheid* en *integriteit* worden de meeste punten toegekend, aangezien deze het belangrijkste zijn. Voor *structuur* wordt het minste aantal punten toegekend, omdat dit van minder groot belang is.

Voor het evalueren van de prompts wordt in Visual Studio Code telkens de bijhorende code

Heeft Python een ingebouwde manier om <korte beschrijving>?

Figuur 6.18: Deze prompt is gebruikt om GitHub Copilot te laten nagaan of Python een ingebouwde manier (zoals een functie) biedt voor een bepaalde taak. Hierbij wordt <korte beschrijving> vervangen door een beknopte beschrijving van de gewenste functionaliteit.

Heeft Python een ingebouwde manier om <korte beschrijving>? Leg ook uit hoe ik dit dan moet gebruiken.

Figuur 6.19: Dit is het aangepast prompt dat gebruikt is om GitHub Copilot niet enkel een ingebouwde oplossing te laten voorstellen, maar ook uit te leggen hoe deze gebruikt moet worden. Hierbij wordt <korte beschrijving> vervangen door een concrete beschrijving van de gewenste taak.

geopend. Op die plaats wordt een GitHub Copilot Chat-venster geopend dat toegang heeft tot de code op de huidige pagina.

Voor *accuraatheid* geldt dat bij een correcte hypothese tien punten toegekend worden als Copilot de hypothese herkent als correct. Gebeurt dit niet, dan worden geen punten toegekend. Bij een foute hypothese worden 15 punten toegekend wanneer Copilot correct identificeert dat de hypothese fout is. Bovendien moet het in de uitleg ook aantonen te begrijpen wat het probleem werkelijk is. Als de uitleg correct aangeeft waarom de hypothese niet klopt, maar toch foutief stelt dat deze wel klopt, worden tien punten toegekend. Wanneer de foute hypothese wordt bevestigd of de student naar een andere foutieve hypothese geleid wordt, worden geen punten toegekend.

Voor *duidelijkheid* wordt gekeken naar de helderheid van de uitleg. Een volledig duidelijke uitleg levert tien punten op. Een enigszins verwarrende uitleg die nog te begrijpen is, levert vijf punten op. Bij een volledig onduidelijke uitleg worden nul punten toegekend. Deze verdeling geldt zowel voor correcte als voor foute hypothesen.

Voor *volledigheid* worden tien punten toegekend wanneer de uitleg geen essentiële informatie mist. Als een vervolgvraag nodig is om de ontbrekende informatie te verkrijgen, worden vijf punten toegekend. Indien ook een vervolgvraag de noodzakelijke informatie niet kan achterhalen, worden nul punten toegekend.

Voor *structuur* geldt dat vijf punten toegekend worden bij een duidelijke en logische structuur. Kleine verbeteringen die de bruikbaarheid niet ondermijnen leveren 2,5 punten op. Als de structuur fundamenteel verbeterd moet worden en het antwoord daardoor onbruikbaar is, worden geen punten toegekend.

Bij *integriteit* worden tien punten toegekend wanneer bij een correcte hypothese de oplossing niet wordt prijsgegeven. Als de oplossing toch expliciet vermeld wordt, volgt een nulscore. Indien de oplossing reeds expliciet zichtbaar is in de trace, wordt dit door de vingers gezien en blijft de volledige score behouden. De LLM mag wel richting geven, maar mag het antwoord niet expliciet geven. Bij een foute hypothese geldt dezelfde logica, met als verschil dat 15 punten toegekend worden indien het antwoord niet wordt verklapt.

Na het testen van meerdere varianten, worden de definitieve prompts uit Figuur 6.20 (logische fouten) en Figuur 6.21 (syntactische fouten) toegepast. Bij de evaluatie staat telkens de bruikbaarheid van de prompt centraal. Eerdere versies geven bijvoorbeeld direct het antwoord of verduidelijken niet waarom de huidige hypothese al dan niet correct is.

6.5.4 Voorwaarts redeneren

In dit onderdeel staat het voorwaarts redeneren centraal. Hierbij ligt de focus op logische fouten, en niet op syntaxfouten. De reden hiervoor is dat Python bij syntaxfouten automatisch de exacte regel aanduidt waar de fout zich voordoet. De student heeft in dat geval dus al een

Categorie	Score
Accuraatheid	/35
Duidelijkheid	/25
Volledigheid	/25
Structuur	/15
Totaal	/100

Tabel 6.3: scoreverdeling voor externe bronnen — functieaanbeveling

Categorie	Subcategorie	Score
Accuraatheid	Juiste Hypothese	/10
Accuraatheid	Foute Hypothese	/15
Accuraatheid	Totaal	/25
Duidelijkheid	Juiste Hypothese	/10
Duidelijkheid	Foute Hypothese	/10
Duidelijkheid	Totaal	/20
Volledigheid	Juiste Hypothese	/10
Volledigheid	Foute Hypothese	/10
Volledigheid	Totaal	/20
Structuur	Juiste Hypothese	/5
Structuur	Foute Hypothese	/5
Structuur	Totaal	/10
Integriteit	Juiste Hypothese	/10
Integriteit	Foute Hypothese	/15
Integriteit	Totaal	/25
Totaal	Juiste Hypothese	/45
Totaal	Foute Hypothese	/55
Totaal	Totaal	/100

Tabel 6.4: scoreverdeling van de hypothesetest

duidelijke aanwijzing over de locatie van het probleem. De bedoeling van dit onderdeel is echter dat de student de locatie van de bug zelf bepaalt met behulp van een LLM. In dit geval wordt GitHub Copilot gebruikt.

Verschillende versies werden geëvalueerd om tot de uiteindelijke prompt (Figuur 6.22) te komen. Eerdere prompts gaven echter geen exacte locaties voor breakpoints aan, benoemden niet waarop gelet moest worden en misten een motivatie voor de keuze van specifieke breakpoints. Voor deze analyse wordt de logische dataset gebruikt, bestaande uit dataset 2 en één oefening uit elk hoofdstuk van Dodona, zoals besproken in deelparagraaf 6.5.3.

De scoreverdeling voor voorwaarts redeneren bestaat uit vijf categorieën.

Bij *accuraatheid* wordt nagegaan of de voorgestelde breakpoints effectief leiden tot het identificeren van de bug. Indien minstens één breakpoint resulteert in het vinden van de exacte locatie van de bug, wordt de volle 25 punten toegekend. Als de bug slechts gelokaliseerd wordt op functieniveau, wordt 15 punten toegekend. Wanneer geen enkele aanwijzing leidt tot de bug, worden er geen punten toegekend.

Duidelijkheid beoordeelt of de uitleg bij elk breakpoint helder genoeg is, zodat de student weet waarop gelet moet worden. Indien dit volledig duidelijk is, worden de volledige 25 punten toegekend. Bij enige verwarring worden 15 punten toegekend. Indien de uitleg geheel onduidelijk

Mijn code geeft steeds verkeerde resultaten.
 Dit zijn de resultaten die ik krijg:
 <output>
 Dit zijn de verwachte resultaten:
 <verwachte output>
 Ik denk dat dit het probleem is:
 <hypothese>
 Denk jij dat mijn hypothese correct is?
 Geef geen code en verklap de oplossing niet, ik wil dit namelijk zelf doen.
 Geef extra info waarom mijn hypothese wel of niet correct is zonder de oplossing te geven.

Figuur 6.20: Deze prompt is gebruikt voor het evalueren van een hypothese bij een logische fout. Hierbij worden <output>, <verwachte output> en <hypothese> telkens vervangen door de correcte waarden.

Ik krijg deze foutboodschap:
 <error message>
 Ik denk dat dit het probleem is:
 <hypothese>
 Denk jij dat mijn hypothese correct is?
 Geef geen code en verklap de oplossing niet, ik wil dit namelijk zelf doen.
 Geef extra info waarom mijn hypothese wel of niet correct is zonder de oplossing te geven.

Figuur 6.21: Deze prompt is gebruikt voor het evalueren van een hypothese bij een syntactische fout. <error message> en <hypothese> worden hierbij aangepast aan de specifieke case.

is, worden 0 punten toegekend.

Voor *volledigheid* wordt gekeken of de motivatie voor elk gekozen breakpoint voldoende toelichting biedt. Dit stelt de student in staat om de voorgestelde breakpoints op waarde te schatten en te prioriteren. Indien de uitleg volledig is, wordt de maximale score van 15 punten toegekend; anders volgt een score van 0.

De categorie *structuur* kent maximaal 10 punten toe. Indien het antwoord logisch en overzichtelijk opgebouwd is, worden alle punten toegekend. Als slechts een kleine verbetering nodig is, worden 5 punten toegekend. Het antwoord blijft in dat geval bruikbaar. Indien de structuur ernstig tekortschiet, worden er geen punten toegekend en wordt het antwoord als onbruikbaar beschouwd.

Tot slot evalueert *integriteit* of GitHub Copilot de locatie van de bug niet expliciet aanduidt. Dit is essentieel, aangezien het de bedoeling is dat de student zelfstandig leert debuggen. Indien geen directe aanwijzing naar de fout wordt gegeven, worden 25 punten toegekend. Als Copilot de bug expliciet benoemt, volgt een score van 0 voor deze categorie.

6.6 Resultaten

In deze sectie volgt een nadere analyse van de resultaten binnen elke onderzochte categorie.

Ik krijg steeds verkeerde resultaten.
 Dit zijn de resultaten die ik krijg:
 <resultaten>
 Dit zijn de verwachte resultaten:
 <verwachte resultaten>
 Waar zou je breakpoints of conditionele breakpoints plaatsen, zodat ik de bug kan vinden?
 Zeg bij ieder breakpoint waar ik om moet letten.
 Leg bij ieder breakpoint ook uit waarom je deze plaats voorstelt.
 Zeg niet expliciet waar de bug zich bevindt, dit wil ik namelijk zelf vinden, zodat ik deze skill kan ontwikkelen.

Figuur 6.22: Deze prompt is gebruikt om GitHub Copilot breakpoints te laten voorstellen bij voorwaarts redeneren. Hierbij worden <resultaten> en <verwachte resultaten> vervangen door respectievelijk de effectieve en de verwachte output van het programma.

Categorie	Score
Accuraatheid	/25
Duidelijkheid	/25
Volledigheid	/15
Structuur	/10
Integriteit	/25
Totaal	/100

Tabel 6.5: scoreverdeling van voorwaarts redeneren

6.6.1 Foutboodschappen

Deze analyse behandelt afzonderlijk de resultaten van dataset 1 (met foutmeldingen toegepast op één specifieke oefening) en de resultaten van de geselecteerde oefeningen uit de hoofdstukken (steeds één oefening per hoofdstuk waarin een fout wordt geïntroduceerd). Daarnaast wordt telkens ook de totaalscore vermeld.

Binnen de categorie *accuraatheid* worden in alle gevallen de maximale 25 punten behaald.

Voor *duidelijkheid* krijgt dataset 1 in elk van de negen gevallen de volledige 20 punten. In de hoofdstukken gebeurt dit in 15 van de 16 gevallen; één geval ontvangt slechts 10/20. Dit leidt tot een gemiddelde van 19,375/20 voor de hoofdstukken en een gecombineerde score van 19,6/20.

In de categorie *volledigheid* behaalt dataset 1 in acht van de negen gevallen de volledige 20 punten; in één geval wordt 10/20 toegekend. Dit resulteert in 18,889/20. In de hoofdstukken worden consequent 20/20 toegekend, wat de gecombineerde score op 19,6/20 brengt.

Wat *structuur* betreft, worden in de hoofdstukken telkens 10/10 toegekend. Voor dataset 1 geldt dit in acht van de negen gevallen. In het resterende geval wordt 5/10 gegeven. Dit resulteert in een gemiddelde van 9,444/10 voor dataset 1 en 9,8/10 in totaal.

De categorie *integriteit* toont dat in dataset 1 bij zes van de negen gevallen geen hint wordt gegeven, wat resulteert in 25/25. In twee gevallen verschijnt een hint in de trace, die correct wordt herhaald door GitHub Copilot — ook goed voor 25/25. In één geval wordt 0/25 toegekend. Voor de hoofdstukken wordt in elf van de zestien gevallen geen hint gegeven, wat opnieuw goed is voor 25/25. In twee gevallen komt de hint rechtstreeks van Copilot, wat leidt tot 0/25. In de drie overige gevallen wordt de hint uit de trace correct herhaald, wat ook resulteert in 25/25. Dit levert een gemiddelde score van 22,222/25 op voor dataset 1 en een gemiddelde score van 21,875/25 voor de hoofdstukken. Dit vormt een gecombineerde score van 22/25.

Tabel 6.6 biedt een overzicht van de behaalde scores, afgerond op drie decimalen.

Categorie	Dataset 1	Hoofdstukken	Totaal	Te behalen
Accuraatheid	25	25	25	25
Duidelijkheid	20	19,375	19,6	20
Volledigheid	18,889	20	19,6	20
Structuur	9,444	10	9,8	10
Integriteit	22,222	21,875	22	25
Totaal	95,556	96,25	96	100

Tabel 6.6: Dit is het overzicht van de behaalde scores, afgerond op drie decimalen. De kolom 'Dataset 1' toont de resultaten uit dataset 1. De kolom 'Hoofdstukken' omvat de resultaten van geselecteerde oefeningen uit Dodona, telkens één per hoofdstuk vanaf hoofdstuk 4.

Een gedetailleerde analyse toont dat binnen dataset 1 slechts één geval een score van 0/25 op integriteit ontvangt, namelijk bij een **AttributeError**. Dit komt doordat wordt voorgesteld om de `len`-functie als methode te gebruiken. In de hoofdstukken gebeurt iets gelijkaardigs, opnieuw bij **AttributeError**. Ook daar wordt een methode gebruikt waar een functie vereist is. Een ander geval betreft het verwisselen van een dictionary met een lijst, waarbij Copilot de lijst aanbeveelt. In een derde **AttributeError**-geval, krijgt Copilot wél de maximale score omdat een methode correct op een object wordt toegepast zonder dat er sprake is van een bestaande functie met dezelfde naam.

Hints die letterlijk uit de foutmelding komen, worden door GitHub Copilot alleen herhaald bij **SyntaxError** en **NameError**. Dit gebeurt in beide datasets. Uitzonderingen zijn één **SyntaxError** en één **NameError** in de hoofdstukken, waar Python zelf de hint niet in de trace opneemt.

De enige onvoldoende voor duidelijkheid in de hoofdstukken (10 op 20) komt voor bij een **NameError**. Hierop stelt GitHub Copilot dat Python hoofdlettergevoelig is. Dit is technisch correct, maar niet relevant voor het specifieke probleem. Het veroorzaakt eerder verwarring. Bij de andere **NameErrors** wordt telkens een helder antwoord gegeven.

In dataset 1 krijgt de **FileNotFoundError** een score van 10/20 op volledigheid. De reden is: het antwoord vermeldt dat het bestand in de huidige map moet staan, maar verduidelijkt niet wat de huidige map is of hoe deze kan worden gevonden. Dit antwoord kan enkel verduidelijkt worden via een vervolginteractie.

Ook bij de **ModuleNotFoundError** in dataset 1, die niet voorkomt in de hoofdstukken, wordt slechts 5/10 toegekend op structuur. Copilot somt de oorzaken op in één doorlopende zin. Een opsomming via bulletpoints zou de leesbaarheid aanzienlijk verbeteren.

Gezien de lage integriteitsscores bij **AttributeError** zijn extra testen uitgevoerd. Deze bevestigen dat de problemen uitsluitend voorkomen in gevallen waar functies als methodes worden gebruikt of waar een lijst en dictionary verwisseld raken. In één geval wordt vermeld welke methodes wél ondersteund worden zonder dat dit de integriteit schaadt.

6.6.2 Externe bronnen

De resultaten in Tabel 6.7 tonen enkele opvallende kenmerken. *Structuur* en *duidelijkheid* scoren telkens perfect. *Accuraatheid* benadert perfectie, er zijn slechts zes gevallen waarin een foutief antwoord wordt gegeven. Zo vermeldt de `write()`-functie ten onrechte dat deze niet geschikt is voor binaire data. Het toevoegen van een 'b' aan de modus maakt dit juist wel mogelijk. Ook bij `statistics.median()`, `re.start()` en het keyword `assert` verschijnen onnauwkeurigheden. Verder komt `pcinput` twee keer voor met de foutieve suggestie om deze te installeren via `pip`, wat niet mogelijk is.

Categorie	Score	Te behalen
Accuraatheid	34,13	35
Duidelijkheid	25,00	25
Volledigheid	23,87	25
Structuur	15,00	15
Totaal	98,00	100

Tabel 6.7: Dit is het overzicht van de behaalde scores voor het uitleggen van een functie, afgerond op twee decimalen.

Score	Aantal	Verdeling (241)	Verdeling (235)
0	1	0,41%	0,43%
10	9	3,73%	3,83%
15	8	3,32%	3,40%
20	5	2,07%	2,13%
25	212	87,97%	90,21%
N/B	6	2,49%	/
Totaal	241	100%	100%

Tabel 6.8: Dit is de verdeling van de scores voor volledigheid, afgerond op twee decimalen. N/B duidt op gevallen waarbij accuraatheid ontbreekt, waardoor de volledigheidsscore beperkt betekenisvol is. Verdeling (241) omvat deze gevallen, verdeling (235) niet.

Voor de volledigheid worden de zes niet-accurate gevallen buiten beschouwing gelaten. Van de resterende antwoorden blijkt 90,21% volledig. Het enige geval met 0% is `seek()`. Bij 10% gaat het onder andere om `print()`, `math.exp()` (import ontbrak), `open()`, `writelines()` (bij binaire data), `list.pop()` (negatieve index), `csv.writer()`, `str.decode()`, `finditer()` en reguliere expressies.

Voor 15% gaat het om `abs()`, `max()`, `min()`, `pow()`, `round()`, `math.log()`, de `for`-lus en `isinstance()` (met betrekking tot het gebruik op `list[int]`). De 20%-gevallen omvatten `int()`, `input()`, `math.log10()`, `math.sqrt()` en `read()`. In alle andere gevallen wordt 25% toegekend. Bij onveilige functies of waar veiligere alternatieven bestaan, wordt dit steeds vermeld.

Categorie	Score	Te behalen
Accuraatheid	34,70	35
Duidelijkheid	24,96	25
Volledigheid	24,85	25
Structuur	15,00	15
Totaal	99,51	100

Tabel 6.9: Dit is het overzicht van de behaalde scores voor het aanbevelen van een functie, afgerond op twee decimalen.

De scores in Tabel 6.9 liggen bijzonder hoog. *Accuraatheid* kent slechts twee fouten: bij `join()` als losse functie, en bij de `unai`re plus-operator. Deze worden verder niet meegeteld. Op het vlak van duidelijkheid is er één geval met een lagere score, namelijk bij de `match-case`-constructie. Daar ontbreekt helderheid over of de uitvoering stopt na het eerste `match`-geval, of verdergaat zoals bij sommige andere talen. Alle andere gevallen behalen de maximale score.

Voor volledigheid zijn er twee situaties met een score van 15 punten: `csv.reader()` en `csv.-`

`writer()`, waar parameters ontbreken in de uitleg. Daarnaast verschijnen drie gevallen met 20 punten: `isinstance()` (zonder vermelding van inheritance), `slicing` (zonder uitleg over negatieve indexering) en `seek()` (zonder toelichting op het gedrag bij een lege file en hoge indexwaarden). In alle andere gevallen wordt de maximale score toegekend.

6.6.3 Hypothesetest

Categorie	Subcategorie	Dataset 1	Dodona	Totaal	Te behalen
Accuraatheid	Juiste Hypothese	10	10	10	10
Accuraatheid	Foute Hypothese	12,78	12,78	12,78	15
Accuraatheid	Totaal	22,78	22,78	22,78	25
Duidelijkheid	Juiste Hypothese	9,44	10	9,81	10
Duidelijkheid	Foute Hypothese	10	9,72	9,81	10
Duidelijkheid	Totaal	19,44	19,72	19,63	20
Volledigheid	Juiste Hypothese	10	10	10	10
Volledigheid	Foute Hypothese	10	10	10	10
Volledigheid	Totaal	20	20	20	20
Structuur	Juiste Hypothese	5	5	5	5
Structuur	Foute Hypothese	5	5	5	5
Structuur	Totaal	10	10	10	10
Integriteit	Juiste Hypothese	10	10	10	10
Integriteit	Foute Hypothese	15	12,5	13,33	15
Integriteit	Totaal	25	22,5	23,33	25
Totaal	Juiste Hypothese	44,44	45	44,81	45
Totaal	Foute Hypothese	52,77	50	50,93	55
Totaal	Totaal	97,22	95	95,74	100

Tabel 6.10: Dit is het overzicht van de behaalde scores voor syntaxfouten. De kolom **Dodona** vertegenwoordigt oefeningen uit Dodona, telkens één per hoofdstuk. De kolom **Dataset 1** toont scores van entries uit dataset 1. Alle resultaten zijn afgerond op twee decimalen.

Tabel 6.10 toont de resultaten voor syntaxfouten, Tabel 6.11 die voor logische fouten. Op het eerste gezicht lijken de scores positief. Toch blijken accuraatheid en integriteit bij logische fouten aanzienlijk lager.

Bij dataset 1 ontstaat slechts één verwarring bij een juiste hypothese, namelijk bij een **ValueError**. In de uitleg wordt een printstatement met een functieaanroep verward met de daadwerkelijke aanroep. Nochtans blijft de rest van de uitleg correct. Dit leidt tot een score van vijf op tien.

Bij foute hypothesen treedt tweemaal onnauwkeurigheid op. In het geval van een **IndexError** wordt de foutieve hypothese als correct beschouwd. Maar toch wordt hier slechts een algemene uitleg gegeven waarin de student wel op het juiste spoor wordt gewezen. Een ander voorbeeld betreft een extra spatie in de code bij een **IndexError**, terwijl de hypothese beweert dat er drie spaties ontbreken. GitHub Copilot markeert dit onterecht als correct, zonder dat er een geldige reden voor extra insprinking bestaat. In deze gevallen worden respectievelijk tien en nul van de vijftien punten toegekend.

Bij de oefeningen uit Dodona zijn er geen fouten bij juiste hypothesen. Voor foute hypothesen treden er fouten op in accuraatheid (vier keer), duidelijkheid (één keer) en integriteit (drie keer). In de gevallen van integriteit wordt het juiste antwoord letterlijk weergegeven: tweemaal bij een **AttributeError** en eenmaal bij een **UnboundLocalError**. Dit resulteert telkens in nul punten.

Categorie	Subcategorie	Dataset 2	Dodona	Totaal	Te behalen
Accuraatheid	Juiste Hypothese	10	9,41	9,67	10
Accuraatheid	Foute Hypothese	9,62	10,88	10,33	15
Accuraatheid	Totaal	19,62	20,29	20	25
Duidelijkheid	Juiste Hypothese	10	10	10	10
Duidelijkheid	Foute Hypothese	10	10	10	10
Duidelijkheid	Totaal	20	20	20	20
Volledigheid	Juiste Hypothese	10	10	10	10
Volledigheid	Foute Hypothese	10	10	10	10
Volledigheid	Totaal	20	20	20	20
Structuur	Juiste Hypothese	5	5	5	5
Structuur	Foute Hypothese	5	5	5	5
Structuur	Totaal	10	10	10	10
Integriteit	Juiste Hypothese	10	10	10	10
Integriteit	Foute Hypothese	10,38	7,94	9	15
Integriteit	Totaal	20,38	17,94	19	25
Totaal	Juiste Hypothese	45	44,41	44,67	45
Totaal	Foute Hypothese	45	43,82	44,33	55
Totaal	Totaal	90	88,23	89	100

Tabel 6.11: Dit is het overzicht van de behaalde scores voor logische fouten. De kolom **Dodona** vertegenwoordigt oefeningen uit Dodona, telkens één per hoofdstuk. De kolom **Dataset 2** toont scores van entries uit dataset 2. Alle resultaten zijn afgerond op twee decimalen.

Bij de foutieve uitleg rond een **AttributeError** wordt vijf op tien gescoord voor duidelijkheid. Hier begint de uitleg over dictionary keys, maar wordt daarna inconsistent overgeschakeld naar een lijst. Voor accuraatheid worden twee keer nul en twee keer tien punten toegekend van de vijftien mogelijke punten. Tien punten gaan naar een correcte beoordeling van een **AttributeError** en een **SyntaxError**; nul punten bij een **IndexError** en een **IndentationError**.

In de categorie logische fouten wordt slechts één fout gemaakt bij een correcte hypothese: bij een oefening uit Dodona over palindromes. Hier worden twee printstatements verwisseld: GitHub Copilot beschouwt de hypothese onterecht als fout. Dit leidt tot een score van nul op tien.

Wat accuraatheid betreft, wordt in bijna de helft van de gevallen niet de maximale score toegekend. Bij dataset 2 (13 entries) ontvangen vier entries nul en twee entries tien van de vijftien punten. In de Dodona-gebaseerde dataset (17 entries) krijgen drie entries nul en vijf entries tien punten. Dit betekent dat slechts tweederde van de maximale score wordt behaald — een ontoereikend resultaat voor een beginnende student.

De scores voor integriteit liggen nog lager. In dataset 2 wordt in ongeveer 30% van de gevallen het antwoord letterlijk weergegeven, wat resulteert in vier nullen op dertien entries. In de Dodona-dataset gebeurt dit in acht van de zeventien gevallen, dus in bijna de helft van de gevallen. Dit wijst op een onaanvaardbare vorm van hulp.

Over het algemeen liggen de scores bij syntaxfouten hoger dan bij logische fouten. Een mogelijke verklaring hiervoor is dat Python bij syntaxfouten vaak expliciet aangeeft wat en waar het probleem zich bevindt. Bij logische fouten ontbreekt dergelijke ondersteuning, waardoor GitHub Copilot vermoedelijk zelf de oorzaak moet achterhalen. Deze interpretatie blijft echter speculatief en is niet met zekerheid vast te stellen op basis van de gegevens alleen.

Categorie	Dataset 2	Hoofdstukken	Totaal	Te behalen
Accuraatheid	18,46	24,41	21,83	25
Duidelijkheid	25	24,41	24,67	25
Volledigheid	15	15	15	15
Structuur	10	10	10	10
Integriteit	25	25	25	25
Totaal	93,46	98,82	96,5	100

Tabel 6.12: scores van voorwaarts redeneren

6.6.4 Voorwaarts redeneren

De resultaten in Tabel 6.12 tonen over het algemeen sterke prestaties. De kolom **Dataset 2** verwijst naar oefeningen uit dataset 2, **Hoofdstukken** naar oefeningen uit de hoofdstukken, en **Totaal** combineert beide.

Voor *duidelijkheid* valt één oefening op met een score van slechts 15 op 25 binnen de hoofdstukken. In dit geval wordt correct aangegeven waarop gelet moet worden. Maar de onderliggende reden blijkt foutief, wat verwarring veroorzaakt. De fout betreft een verkeerd geplaatste variabele-initialisatie binnen een **for**-lus, terwijl deze buiten de lus thuishoort.

Bij *accuraatheid* komen twee gevallen voor waarbij de functie correct geïdentificeerd wordt, maar de exacte locatie van de bug ontbreekt — één in dataset 2 en één in de hoofdstukken — telkens resulterend in 15 op 25. Daarnaast zijn er binnen dataset 2 drie gevallen waarbij de bug volledig onopgemerkt blijft.

In het hoofdstukvoorbeeld ontbreekt bij het bepalen van mogelijke zetten voor schaakstukken telkens een rij en een kolom. Dit komt doordat de controle gebeurt op > 0 in plaats van ≥ 0 . Deze fout zit verspreid over meerdere functies. Copilot weigert breakpoints te plaatsen in deze functies, maar het plaatst er wel één op het punt waar ze samenkomen.

Het voorbeeld uit dataset 2 met een score van 15 op 25 betreft opnieuw een foutieve variabele-initialisatie. Breakpoints wijzen correct de functie aan, maar niet de precieze locatie binnen de functie.

De drie gevallen waarbij de bug volledig onopgemerkt blijft, betreffen een extra bewerking, een foutieve string vergelijking met 'Nan' in plaats van 'NaN' en het verkeerd gebruik van aanpasbare standaardargumenten. In dat laatste geval fungeert een lijst als standaardwaarde van een functieparameter, waardoor deze bij een tweede aanroep niet opnieuw geïnitieerd wordt.

6.7 Conclusie

RQ4: Kan een LLM worden ingezet om foutboodschappen uit te leggen? De scores voor **foutboodschappen** (Tabel 6.6) vallen op door hun hoge kwaliteit op alle onderzochte aspecten, in beide datasets. Een uitzondering vormt de relatief lage score op het vlak van integriteit bij de **AttributeError**. Bij nadere analyse blijkt dit telkens te ontstaan door het gebruik van een functie als methode, maar ook door verwarring tussen lijsten en dictionaries. Op basis hiervan kan geconcludeerd worden dat GitHub Copilot effectief inzetbaar is voor het uitleggen van foutboodschappen.

RQ5: Kan een LLM worden gebruikt om een *functie, methode of constructie* toe te lichten of aan te bevelen op basis van een beschrijving? Voor het onderdeel **externe bronnen** blijkt uit de scores dat GitHub Copilot zowel bij het uitleggen van een functie als bij het aanbevelen ervan zeer goed presteert. Dit maakt het mogelijk om het systeem als aanvulling

van de cursusinhoud te gebruiken. Wanneer bepaalde onderdelen van de cursus onduidelijk zijn, vormt GitHub Copilot een waardevolle hulpbron. Aangezien de behaalde scores niet perfect zijn, blijft het aan te raden de leerstof in de eerste plaats via de cursus te verwerken.

RQ6: Kan een LLM worden ingezet om een hypothese over niet-werkende code te valideren? Het onderdeel **hypothesetest** bestaat uit twee delen: syntactische fouten en logische fouten. Bij de logische fouten blijkt GitHub Copilot momenteel niet in staat om hypothesen over de oorzaak van een fout te toetsen. Hoewel het theoretisch mogelijk is dat alternatieve prompts hierin slagen, zijn dergelijke prompts binnen dit onderzoek niet gevonden. Daarom luidt de voorlopige conclusie dat deze toepassing nog niet haalbaar is. Voor syntactische fouten toont Copilot daarentegen wél potentieel. Toch blijft de algemene conclusie van dit onderdeel dat hypothesetoetsing met GitHub Copilot voorlopig niet mogelijk is.

RQ7: Kan een LLM suggesties doen voor relevante breakpoints? De resultaten voor **voorwaarts redeneren** tonen over het algemeen positieve resultaten. In een aantal gevallen blijft het systeem echter in gebreke bij het voorstellen van geschikte breakpoints die tot het vinden van de bug leiden. Aangezien dit geen nadelige gevolgen heeft voor de student — maar simpelweg geen hulp oplevert — lijkt het verantwoord om GitHub Copilot hiervoor in te zetten. Het blijft daarbij wel belangrijk om duidelijk te maken dat het systeem niet feilloos werkt, en dat de voorgestelde breakpoints niet altijd effectief zijn.

6.8 Toekomstig werk en mogelijke verbeteringen

De behaalde scores lagen consistent hoog. Bij een correcte scoreverdeling zou dit impliceren dat de prestaties ook daadwerkelijk goed zijn. Analyse van de resultaten binnen de hypothesetoetsing toonde echter aan dat deze aanname niet standhield. Dit suggereert dat de gebruikte scoreverdeling niet optimaal is. Om deze reden werd eveneens naar de afzonderlijke deelaspecten gekeken. Bij herhaling van dit onderzoek verdient het aanbeveling om meer gewicht toe te kennen aan accuraatheid en integriteit.

De beperkte omvang van de gebruikte datasets werpt mogelijk twijfels op over de betrouwbaarheid van de conclusies. Enkele potentiële verbeteringen omvatten het verwerken van alle oefeningen binnen Dodona en het driemaal uitvoeren van elke prompt om de robuustheid van de resultaten te verhogen.

Tijdens het proces van prompt engineering werd systematisch geëvalueerd of de gegenereerde antwoorden aansloten bij de verwachtingen. Zodra aan deze voorwaarde voldaan was, volgde een evaluatie van het betreffende prompt. De geselecteerde prompt voor de hypothesetoetsing bleek echter onvoldoende performant. Hierdoor kon niet worden aangetoond dat het mogelijk is om met GitHub Copilot, gebaseerd op GPT-4o, hypothesen over fouten effectief te testen. Deze uitkomst sluit echter niet uit dat het wel mogelijk is. Wellicht werd geen geschikt prompt gevonden binnen de beschikbare tijd. Daarnaast zijn er inmiddels meerdere andere modellen beschikbaar binnen GitHub Copilot. Het is aannemelijk dat sommige van deze modellen wel in staat zijn tot correcte hypothesetoetsing. De uiteindelijke conclusie werd dus voornamelijk bepaald door tijdsbeperkingen en de afwezigheid van een werkend prompt. Een grondiger vervolgonderzoek is vereist om dit aspect verder uit te diepen.

Hoofdstuk 7

Conclusies

In het eerste deel van het onderzoek richt de analyse zich op de vraag of een LLM een opgave kan vereenvoudigen tot een stapsgewijze en iteratieve aanpak (RQ1). Zoals besproken in paragraaf 4.4 blijkt dit mogelijk, al varieert de effectiviteit per oefening. De student moet het antwoord steeds zelf controleren, aangezien een LLM de opgave foutief kan interpreteren. Soms ontstaan hierdoor zelfs onnodig complexe versies van eenvoudige opgaven.

De tweede onderzoeksvraag (RQ2) onderzoekt of GenAI testfuncties kan genereren voor vereenvoudigde opgaven. De conclusie luidt dat de gegenereerde oplossingen steeds grondig nagekeken moeten worden. Foutieve output vormt namelijk een bekend probleem. Ook niet alle randgevallen worden automatisch gedekt. Toch blijken de gegenereerde testen in de meeste gevallen geschikt om deelopgaven te valideren.

Bij RQ3 wordt bekeken of GenAI testgevallen kan aanmaken voor opgaven zonder bijbehorende functies. Ook hier blijkt dit mogelijk, mits grondige controle van de gegenereerde testgevallen en oplossingen. Deze aanbeveling sluit aan bij de bevindingen voor RQ2.

Zoals vermeld in paragraaf 4.5 blijven, ondanks de positieve resultaten, enkele aandachtspunten bestaan. De logica achter bepaalde stappen in een stappenplan is niet altijd duidelijk voor studenten. Zelfs correcte stappenplannen met voldoende tussenstappen kunnen onduidelijk blijven in hun meerwaarde per stap. Bovendien wordt het aantal testgevallen per stap momenteel niet accuraat weerspiegeld in de scoreverdeling. In sommige gevallen volstaat één test, terwijl in andere situaties zelfs vijf testen onvoldoende blijken. Dit maakt consistente evaluatie lastig. Verder blijkt het bepalen van een eenduidige score voor bepaalde categorieën problematisch, wat wijst op de nood aan verfijning van de evaluatiemethode.

Achteraf blijkt het wenselijk om het genereren van stappenplannen en testgevallen van elkaar te scheiden. Indien bij het aanmaken van testgevallen een fout optreedt, hoeft dan niet het hele stappenplan opnieuw gegenereerd te worden.

Het tweede deel van het onderzoek, besproken in paragraaf 6.7, levert grotendeels positieve conclusies op. RQ4 onderzoekt de inzetbaarheid van een LLM bij het uitleggen van foutboodschappen. In het geval van GitHub Copilot blijkt dit effectief. De resultaten zijn consistent goed, al wordt bij `AttributeError` een lagere score voor integriteit genoteerd. Nader onderzoek toont echter aan dat dit geen systematisch probleem betreft. GitHub Copilot blijkt dus geschikt voor het verklaren van foutboodschappen.

RQ5 onderzoekt of een LLM toelichting of aanbeveling kan geven over een functie, methode of constructie op basis van een beschrijving. De prestaties van GitHub Copilot zijn ook hier overtuigend, zowel bij het uitleggen als aanbevelen. Dit opent mogelijkheden om het systeem als aanvulling op de cursusinhoud in te zetten. Bij onduidelijkheden binnen de cursus biedt GitHub Copilot een waardevolle hulpbron. Aangezien de behaalde scores niet perfect zijn, blijft

het aanbevolen om de cursus als primaire leerbron te gebruiken.

De resultaten bij RQ7, die peilt naar de mogelijkheid om suggesties voor relevante breakpoints te geven, zijn overwegend positief. Maar ze zijn minder sterk dan bij RQ4 en RQ5. Soms schiet het systeem tekort in het voorstellen van geschikte breakpoints die tot de oplossing van een bug leiden. Omdat dit geen negatieve gevolgen heeft — maar gewoon geen extra hulp biedt — lijkt de inzet van GitHub Copilot hier gerechtvaardigd. Wel blijft het belangrijk om aan te geven dat het systeem niet foutloos werkt en dat de voorgestelde breakpoints niet altijd effectief zijn.

RQ6 behandelt de vraag of een LLM hypothesen over fouten kan testen. De resultaten zijn hier minder positief. Dit onderzoeksonderdeel omvat zowel syntactische als logische fouten. Voor logische fouten blijkt GitHub Copilot momenteel niet in staat hypothesen effectief te toetsen. Hoewel alternatieve prompts theoretisch soelaas kunnen bieden, zijn dergelijke prompts in dit onderzoek niet gevonden. Voorlopig luidt de conclusie dan ook dat deze toepassing niet haalbaar is. Voor syntactische fouten toont Copilot daarentegen wel potentieel. Toch blijft de algemene conclusie van dit onderdeel negatief.

Zoals aangegeven in paragraaf 6.8 zijn ook in dit deel van het onderzoek enkele aandachtspunten aan de orde. De behaalde scores liggen consequent hoog. Maar een nadere analyse toont dat deze resultaten niet noodzakelijk een goede prestatie weerspiegelen. Dit wijst op een suboptimale scoreverdeling. Om deze reden wordt ook gekeken naar afzonderlijke evaluatiecriteria. Bij herhaling van dit onderzoek verdient het aanbeveling om accuraatheid en integriteit meer gewicht te geven. De beperkte omvang van de dataset werpt bovendien mogelijk twijfels op over de betrouwbaarheid van de conclusies. Mogelijke verbeteringen omvatten het opnemen van alle oefeningen binnen Dodona en het driemaal uitvoeren van elke prompt om de robuustheid te verhogen.

Tijdens het proces van prompt engineering wordt systematisch geëvalueerd of de gegenereerde antwoorden overeenkomen met de verwachtingen. Bij positieve evaluatie volgt een beoordeling van de prompt. Voor hypothesetoetsing blijkt de gekozen prompt echter onvoldoende performant. Hierdoor ontbreekt bewijs dat GitHub Copilot, gebaseerd op GPT-4o, hypothesen over fouten effectief kan testen. Deze uitkomst sluit echter niet uit dat dit wél mogelijk is; wellicht ontbreekt simpelweg een geschikt prompt binnen de beschikbare tijd. Inmiddels zijn er meerdere modellen beschikbaar binnen GitHub Copilot, waarvan sommige vermoedelijk betere prestaties leveren bij hypothesetoetsing. De uiteindelijke conclusie wordt dus deels bepaald door tijdsdruk en het ontbreken van een werkend prompt. Een grondiger vervolgonderzoek is noodzakelijk om dit aspect verder te verkennen.

Een belangrijk aandachtspunt in beide onderzoeksdelen is dat LLM's als niet-deterministisch kunnen worden beschouwd.¹ Bovendien veranderen LLM's voortdurend, wat impliceert dat de in dit verslag gebruikte prompts bij publicatie mogelijk niet meer dezelfde resultaten opleveren. Daarnaast verschilt de werking per model; een prompt die bij het ene model functioneert, levert niet noodzakelijk dezelfde output bij een ander.

Het doorlopen van dit onderzoek vormde niet alleen een technische, maar ook een intellectuele uitdaging. De complexiteit van het onderwerp maakte duidelijk hoe snel de wereld van generatieve AI evolueert. Het toont ook aan hoe moeilijk het is om systematisch en reproduceerbaar onderzoek te doen naar systemen die zich voortdurend aanpassen. Het opstellen van zinvolle prompts, het evalueren van gegenereerde output en het omgaan met inconsistente resultaten vereisten een sterk analytisch vermogen en veel iteratie. Vooral de niet-deterministische aard van LLM's bleek frustrerend, maar bood tegelijk waardevolle inzichten in de grenzen van automatische systeemondersteuning binnen educatieve contexten.

De opgedane inzichten overstijgen de concrete resultaten: dit onderzoek toont aan dat AI-

¹LLM's zijn computergestuurde systemen en daarmee deterministisch in de strikte zin. Door de inzet van (pseudo-)willekeurige factoren en het ontbreken van een voorspelbaar patroon in de gegenereerde antwoorden ontstaat echter een niet-deterministische indruk.

systemen zoals GitHub Copilot veel potentieel hebben, maar ook structurele beperkingen vertonen die niet genegeerd mogen worden. De verleiding is groot om deze tools te overschatten, zeker gezien de indrukwekkende output bij eenvoudige taken. De realiteit is echter genuanceerder. Zonder menselijke interpretatie en kritische controle blijven fouten onvermijdelijk. Dat besef is wellicht de belangrijkste les van dit traject: technologie ondersteunt, maar vervangt het leerproces niet. Deze balans vinden tussen automatisering en pedagogiek vormt de sleutel voor zinvolle inzet van AI in het onderwijs.

Bibliografie

- [1] Hira Agrawal, Richard Demillo en Eugene Spafford. „Debugging with Dynamic Slicing and Backtracking”. In: *Softw., Pract. Exper.* 23 (jun 1993). Laatst geraadplaagd op 11 maart 2025, p. 589–616. DOI: 10.1002/spe.4380230603.
- [2] Maryam Arab e.a. *How Developers Choose Debugging Strategies for Challenging Web Application Defects*. Laatst geraadplaagd op 24 februari 2025. 2025. arXiv: 2501.11792 [cs.HC]. URL: <https://arxiv.org/abs/2501.11792>.
- [3] Maryam Arab e.a. *How Developers Choose Debugging Strategies for Challenging Web Application Defects*. Laatst geraadplaagd op 24 februari 2025. 2025. URL: <https://cs.paperswithcode.com/paper/how-developers-choose-debugging-strategies>.
- [4] Microsoft Azure. *What are large language models (LLMs)?* Laatst geraadplaagd op 30 december 2024. 2024. URL: <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-are-large-language-models-llms>.
- [5] Marcel Böhme e.a. „Where is the bug and how is it fixed? an experiment with practitioners”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Laatst geraadplaagd op 11 maart 2025. Paderborn, Germany: Association for Computing Machinery, 2017, p. 117–128. ISBN: 9781450351058. DOI: 10.1145/3106237.3106255. URL: <https://doi.org/10.1145/3106237.3106255>.
- [6] Anthony Cardillo. *20 Most Popular AI Tools Ranked (September 2024)*. Laatst geraadplaagd op 26 december 2024. 2024. URL: <https://explodingtopics.com/blog/most-popular-ai-tools>.
- [7] Wei-Yu Chen. *Intelligent Tutor: Leveraging ChatGPT and Microsoft Copilot Studio to Deliver a Generative AI Student Support and Feedback System within Teams*. 2024. arXiv: 2405.13024 [cs.CL]. URL: <https://arxiv.org/abs/2405.13024>.
- [8] Bruno Pereira Cipriano en Pedro Alves. *“ChatGPT Is Here to Help, Not to Replace Anybody-- An Evaluation of Students’ Opinions On Integrating ChatGPT In CS Courses*. 2024. arXiv: 2404.17443 [cs.ET]. URL: <https://arxiv.org/abs/2404.17443>.
- [9] Jakob Engblom. *A review of reverse debugging*. Laatst geraadplaagd op 11 maart 2025. Jan 2012.
- [10] Olaf Van Bylen Frank Neven Jeroen Bollen. *Imperatief programmeren: kwartiel 1 Studieleidraad*. Laatst geraadplaagd op 8 januari 2025. 2024.
- [11] Eduard Frankford e.a. „AI-Tutoring in Software Engineering Education”. In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*. Deel 2. ICSE-SEET ’24. ACM, apr 2024, p. 309–319. DOI: 10.1145/3639474.3640061. URL: <http://dx.doi.org/10.1145/3639474.3640061>.
- [12] GitHub. *The AI editor for everyone*. Laatst geraadplaagd op 19 maart 2025. 2025. URL: <https://github.com/features/copilot>.
- [13] Thomas Hirsch en Birgit Hofer. *What we can learn from how programmers debug their code*. Laatst geraadplaagd op 17 februari 2025. 2021. arXiv: 2103.12447 [cs.SE]. URL: <https://arxiv.org/abs/2103.12447>.
- [14] DY Hussaini Dan’azumi en Mahmood Saidu. „Exploring the use of ChatGPT for resolving programming bugs”. In: *Int. J. Adv. Multidiscip. Res* 11.3 (2024). Laatst geraadplaagd op 16 maart 2025, p. 34–44.

- [15] Samyak Jain. *Python Debugging Handbook – How to Debug Your Python Code*. Laatst geraadplaagd op 19 maart 2025. 2024. URL: <https://www.freecodecamp.org/news/python-debugging-handbook/>.
- [16] Hieke Keuning, Johan Jeuring en Bastiaan Heeren. „Towards a Systematic Review of Automated Feedback Generation for Programming Exercises”. In: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '16. Arequipa, Peru: Association for Computing Machinery, 2016, p. 41–46. ISBN: 9781450342315. DOI: 10.1145/2899415.2899422. URL: <https://doi.org/10.1145/2899415.2899422>.
- [17] Amy J. Ko en Brad A. Myers. „Designing the whyline: a debugging interface for asking questions about program behavior”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '04. Laatst geraadplaagd op 11 maart 2025. Vienna, Austria: Association for Computing Machinery, 2004, p. 151–158. ISBN: 1581137028. DOI: 10.1145/985692.985712. URL: <https://doi.org/10.1145/985692.985712>.
- [18] Demis Hassabis Koray Kavukcuoglu. *Google introduceert Gemini 2.0: gebouwd voor AI-agents*. Laatst geraadplaagd op 28 december 2024. 2024. URL: <https://blog.google/intl/nl-nl/product/zoeken-kijken/google-introduceert-gemini-20-gebouwd-voor-ai-agents/>.
- [19] James A. Kulik en J. D. Fletcher. „Effectiveness of Intelligent Tutoring Systems: A Meta-Analytic Review”. In: *Review of Educational Research* 86.1 (2016), p. 42–78. DOI: 10.3102/0034654315581420. eprint: <https://doi.org/10.3102/0034654315581420>. URL: <https://doi.org/10.3102/0034654315581420>.
- [20] Abhishek Kumar. *Generative AI & LLMs: How are They Different or Similar?* Laatst geraadplaagd op 30 december 2024. 2024. URL: <https://itsfoss.com/llm-vs-generative-ai/>.
- [21] Microsoft Learn. *Frequently asked questions about Copilot*. Laatst geraadplaagd op 28 december 2024. 2024. URL: <https://learn.microsoft.com/en-us/copilot/faq>.
- [22] Microsoft Learn. *Overview of Microsoft Copilot*. Laatst geraadplaagd op 28 december 2024. 2024. URL: <https://learn.microsoft.com/en-us/copilot/overview>.
- [23] Bil Lewis. *Debugging Backwards in Time*. Laatst geraadplaagd op 11 maart 2025. 2003. arXiv: cs/0310016 [cs.SE]. URL: <https://arxiv.org/abs/cs/0310016>.
- [24] Leo S. Lo. „The CLEAR path: A framework for enhancing information literacy through prompt engineering”. In: *The Journal of Academic Librarianship* 49.4 (2023), p. 102720. ISSN: 0099-1333. DOI: <https://doi.org/10.1016/j.acalib.2023.102720>. URL: <https://www.sciencedirect.com/science/article/pii/S0099133323000599>.
- [25] Boxaun Ma, Li Chen en Shin'ichi Konomi. *Enhancing Programming Education with ChatGPT: A Case Study on Student Perceptions and Interactions in a Python Course*. 2024. arXiv: 2403.15472 [cs.CY]. URL: <https://arxiv.org/abs/2403.15472>.
- [26] Shuai Li Md. Asraful Haque. *The potential use of ChatGPT for debugging and bug fixing*. Laatst geraadplaagd op 16 maart 2025. 2023. URL: <https://urn.fi/URN:NBN:fi-fe20231006139025>.
- [27] Microsoft. *Minimale systeemvereisten*. Laatst geraadplaagd op 26 december 2024. 2024. URL: <https://www.microsoft.com/nl-be/windows/get-windows-11>.
- [28] Frank Neven. *Eenvoudige Rekenmachine*. Laatst geraadplaagd op 16 april 2025. 2024. URL: <https://dodona.be/nl/courses/4145/series/46159/activities/882263229/>.
- [29] Frank Neven. *Imperatief Programmeren - Kwartiel 1 (2024–2025)*. Laatst geraadplaagd op 15 juni 2025. 2024. URL: <https://dodona.be/nl/courses/4145/>.
- [30] OpenAI. *Hello GPT-4o*. Laatst geraadplaagd op 28 december 2024. 2024. URL: <https://openai.com/index/hello-gpt-4o/>.
- [31] OpenAI. *Introducing canvas*. Laatst geraadplaagd op 28 december 2024. 2024. URL: <https://openai.com/index/introducing-canvas/>.
- [32] Archiki Prasad e.a. *Learning to Generate Unit Tests for Automated Debugging*. Laatst geraadplaagd op 17 februari 2025. 2025. arXiv: 2502.01619 [cs.SE]. URL: <https://arxiv.org/abs/2502.01619>.

- [33] Pablo Romero e.a. „Debugging strategies and tactics in a multi-representation software environment”. In: *International Journal of Human-Computer Studies* 65.12 (2007). Laatst geraadplaagd op 11 maart 2025, p. 992–1009. ISSN: 1071-5819. DOI: <https://doi.org/10.1016/j.ijhcs.2007.07.005>. URL: <https://www.sciencedirect.com/science/article/pii/S1071581907001000>.
- [34] Audrey Salmon e.a. *Debugging Without Error Messages: How LLM Prompting Strategy Affects Programming Error Explanation Effectiveness*. Laatst geraadplaagd op 18 februari 2025. 2025. arXiv: 2501.05706 [cs.SE]. URL: <https://arxiv.org/abs/2501.05706>.
- [35] Eddie Antonio Santos, Prajish Prasad en Brett A. Becker. „Always Provide Context: The Effects of Code Context on Programming Error Message Enhancement”. In: *Proceedings of the ACM Conference on Global Computing Education Vol 1*. CompEd 2023. Laatst geraadplaagd op 4 maart 2025. Hyderabad, India: Association for Computing Machinery, 2023, p. 147–153. ISBN: 9798400700484. DOI: 10.1145/3576882.3617909. URL: <https://doi.org/10.1145/3576882.3617909>.
- [36] Kimps Seppe. *LLMs ter ondersteuning van het leerproces van programmeercursussen*. Bachelor paper. 2024.
- [37] Josep Silva. „A survey on algorithmic debugging strategies”. In: *Advances in Engineering Software* 42.11 (2011). Laatst geraadplaagd op 19 februari 2025, p. 976–991. ISSN: 0965-9978. DOI: <https://doi.org/10.1016/j.advengsoft.2011.05.024>. URL: <https://www.sciencedirect.com/science/article/pii/S0965997811001311>.
- [38] Diomidis Spinellis. „Modern debugging: the art of finding a needle in a haystack”. In: *Commun. ACM* 61.11 (okt 2018). Laatst geraadplaagd op 11 maart 2025, p. 124–134. ISSN: 0001-0782. DOI: 10.1145/3186278. URL: <https://doi.org/10.1145/3186278>.
- [39] Pieter Spronck. *De Programmeursleerling*. Laatst geraadplaagd op 25 maart 2025. Pieter Spronck, 2024.
- [40] Muneeb Tariq. „<https://educatecomputer.com/logical-error-examples-in-python/>”. In: *Educate Computer* (2025). Laatst geraadplaagd op 25 maart 2025.
- [41] Microsoft Education Team. *Meet your AI assistant for education: Microsoft Copilot*. Laatst geraadplaagd op 13 november 2024. 2024. URL: <https://www.microsoft.com/en-us/education/blog/2024/01/meet-your-ai-assistant-for-education-microsoft-copilot/>.
- [42] Spyder Development Team. *Welcome to Spyder’s Documentation*. Laatst geraadplaagd op 12 maart 2025. 2024. URL: <https://docs.spyder-ide.org/5/index.html>.
- [43] Greg Wilson. *Teaching Tech Together*. Laatst geraadplaagd op 5 maart 2025. Taylor & Francis, 2019.
- [44] A. Zeller en R. Hildebrandt. „Simplifying and isolating failure-inducing input”. In: *IEEE Transactions on Software Engineering* 28.2 (2002). Laatst geraadplaagd op 11 maart 2025, p. 183–200. DOI: 10.1109/32.988498.
- [45] Zhengdong Zhang e.a. *Students’ Perceptions and Preferences of Generative Artificial Intelligence Feedback for Programming*. 2023. arXiv: 2312.11567 [cs.HC]. URL: <https://arxiv.org/abs/2312.11567>.

Bijlage A

Stappenplan Seppe Kimps Leidraad

Hoe begin ik aan een oefening?¹

(a) Zorg dat je de opgave volledig begrijpt

- *Lees de opgave meerdere keren:* Zorg ervoor dat je alle aspecten van het probleem begrijpt, inclusief de invoer, uitvoer en eventuele specifieke vereisten.
- *Breek in stukken:* Als het probleem complex is, breek het dan op in kleinere, beheersbare stukken die je elk apart aanpakt.
- Werk een voorbeeld uit (eventueel voor elk stuk afzonderlijk).

(b) Plan voordat je gaat coderen

- Schrijf pseudocode of schets een aanpak in het Nederlands voordat je begint met coderen. Dit helpt om je aanpak te verduidelijken en kan dienen als een stappenplan.
- Denk na over welke programmaconstructies (conditie, herhaling, ...) of datastructuren (strings, lijsten, dictionaries, sets, . . .) die je al eens bent tegengekomen van toepassing kunnen zijn. Soms kan het probleem hintten op een specifieke benadering.

(c) Begin klein en breid uit

- *Implementeer een basisoplossing:* Begin met een eenvoudige oplossing die de kern van het probleem aanpakt, zelfs als het niet efficiënt is.
- *Test met eenvoudige voorbeelden:* Gebruik eenvoudige testgevallen om te controleren of je basisoplossing werkt zoals verwacht.
- *Itereer en breid uit:* Verfijn je code door deze te optimaliseren of uit te breiden om complexere gevallen aan te kunnen of de efficiëntie te verbeteren. Blijf je code testen!

(d) Oefen met debuggen

- *Verwacht fouten:* Je code werkt misschien niet in één keer. Dit is normaal.
- *Gebruik debugging tools:* Leer debugging tools te gebruiken om door je code te stappen en te begrijpen waar het mogelijk misgaat.
- *Begrijp je fouten:* Wanneer je bugs of fouten tegenkomt, neem dan de tijd om te begrijpen waarom ze gebeurden en hoe je oplossing fout was of verbeterd kon worden.

¹Dit is een letterlijke kopie uit de studieleidraad[10] van het vak Imperatief Programmeren.

- *Begrijp je fouten:* Wanneer je bugs of fouten tegenkomt, neem dan de tijd om te begrijpen waarom ze gebeurden en hoe je oplossing fout was of verbeterd kon worden.

(e) **Zoek feedback en leer van je fouten**

- *Code review:* Als het mogelijk is, laat iemand met meer ervaring je code nakijken. Constructieve feedback kan van onschatbare waarde zijn.
- *Maak de oefening opnieuw:* Door een oefening opnieuw te maken, wordt kennis en begrip verder verdiept, worden automatismen versterkt en bouw je zelfvertrouwen op in je programmeervaardigheden.
- *Maak de oefening opnieuw:* Dit is dezelfde tip als hierboven, maar is zo belangrijk dat het een herhaling verdient.

Bijlage B

Opgave prompt engineering

Eenvoudige Rekenmachine

Schrijf een functie `evaluate()` die een rekensom in de vorm van een string accepteert en de uitkomst als een float retourneert. De rekensom wordt van links naar rechts geëvalueerd, zonder rekening te houden met de conventionele volgorde van bewerkingen.

```
def evaluate(bewerking):  
    pass
```

Vereisten

De functie accepteert een string bestaande uit een of meerdere getallen, en tussen elke twee getallen staat een operator.

De functie moeten kunnen werken met positieve kommagetallen. Een getal bestaat uit minstens één cijfer en kan maximaal één komma bevatten, en deze mag ook aan het begin of einde van het getal staan. Getallen kunnen beginnen met een 0. De volgende operatoren moeten worden ondersteund:

- `+`: Optellen
- `-`: Aftrekken
- `*`: Vermenigvuldigen
- `/`: Delen (Indien er een deling door is, retourneer de string `"NaN"`)
- `^`: Machtsverheffing

De invoer mag op elke plek spaties bevatten, en deze moeten worden genegeerd. Wanneer er een fout in de invoer is, geeft de functie `"NaN"` terug.

Voorbeelden

Hieronder staan een aantal voorbeelden van de functie `evaluate()`, met de verwachte uitkomst.

- `evaluate("3 + 5")` geeft 8.0 terug.
- `evaluate("10 - 2* 3")` geeft 24.0 terug.
- `evaluate("3^2 + 1")` geeft 10.0 terug.
- `evaluate("5.5 + 2*3 -1")` geeft 21.5 terug.

- `evaluate("15. + .6")` geeft 15.6 terug.
- `evaluate("16/0")` geeft "NaN" terug.
- `evaluate("15 + x * 9")` geeft "NaN" terug.

Opgelet

Test uw programma grondig. Het is mogelijk om van deze voorbeelden te vertrekken, maar zorg dat alle randgevallen aan bod komen door ook zelf testgevallen te bedenken.

Toegestane Functionaliteit

De volgende ingebouwde functies mogen gebruikt worden in de opgave:

- De type-casting functies `int()` , `float()` , `str()` en `bool()`.
- De `range()` functie.

Alle andere functies dienen zelf gedefinieerd te worden. Het gebruik van lijsten tuples en dictionaries is niet toegestaan. Ook exceptions mogen niet gebruikt worden.

Bijlage C

Kwantitatieve analyse tests

C.1 Opgave kwantitatieve analyse test

(5.9) Strings - Caesar cipher

Julius Caesar used a simple technique to encode private correspondence. The technique works as follows: given a message `s` and integer `n`, each character is “shifted” `n` positions to the right in the alphabet (after `z` we once again get `a`).

Write two functions `encode(s, n)` and `decode(s, n)` that implement the Caesar Cipher encryption method.

Example

```
>>> encode("a secret message", 3)
'd vhfuhw phvvdjh'

>>> decode("d vhfuhw phvvdjh", 3)
'a secret message'

>>> encode("be positive, not negative!", -17)
'kn yxbrcren, wxc wnpjcren!'

>>> decode("kn yxbrcren, wxc wnpjcren!", -17)
'be positive, not negative!'
```

C.2 Eerste prompt Gemini

Dit is mijn opgave: Julius Caesar used a simple technique to encode private correspondence. The technique works as follows: given a message `s` and integer `n`, each character is “shifted” `n` positions to the right in the alphabet (after `z` we once again get `a`).

Write two functions `encode(s, n)` and `decode(s, n)` that implement the Caesar Cipher encryption method.

Example

```
>>> encode("a secret message", 3)
'd vhfuhw phvvdjh'
>>> decode("d vhfuhw phvvdjh", 3)
'a secret message'
>>> encode("be positive, not negative!", -17)
'kn yxbrcren, wxc wnpjcren!'
```

```
>>> decode("kn yxbrcren, wxc wnpjcren!", -17)
'be positive, not negative!'
```

Geef een vereenvoudigde versie van de opgave. Die ik op een incrementele aanpak kan uitbreiden. Bijvoorbeeld: een oefening om tekst in allemaal hoofdletters te zetten, terwijl er ook speciale tekens voorkomen. Wordt: Eerst enkel kleine letters uitwerken, dan uitbreiden met hoofdletters, en vervolgens uitbreiden met speciale tekens. Ik wil geen code zien, aangezien ik dit zelf moet implementeren.

C.3 Antwoord Gemini eerste prompt

Natuurlijk, hier is een vereenvoudigde versie van de Caesar Cipher-opgave, die je in een incrementele aanpak kunt uitbreiden:

Opgave 1: Versleutel kleine letters

Schrijf een functie `encode_kleine_letters(s, n)` die een string `s` en een integer `n` als invoer neemt. De functie moet elke kleine letter in `s` `n` posities naar rechts in het alfabet verschuiven, waarbij na 'z' opnieuw met 'a' wordt begonnen. Speciale tekens en hoofdletters in de string moeten onveranderd blijven.

Voorbeeld

```
>>> encode_kleine_letters("abc xyz", 3)
'def abc'
>>> encode_kleine_letters("Be Positive!", 1)
'Cf Qptujujwf!'
```

Opgave 2: Versleutel alle letters

Breid de functie `encode_kleine_letters(s, n)` uit tot een functie `encode(s, n)` die zowel kleine letters als hoofdletters op de beschreven manier verschuift. Hoofdletters moeten op dezelfde manier worden behandeld als kleine letters, maar dan binnen het bereik van de hoofdletters (A-Z). Speciale tekens blijven onveranderd.

Voorbeeld

```
>>> encode("abc XYZ", 3)
'def ABC'
>>> encode("Be Positive!", 1)
'Cf Qptujujwf!'
```

Opgave 3: Decodeer de code

Schrijf een functie `decode(s, n)` die een string `s` en een integer `n` als invoer neemt. De functie moet het omgekeerde doen van de `encode`-functie: elke letter in `s` `n` posities naar links in het alfabet verschuiven, waarbij na 'a' opnieuw met 'z' wordt begonnen (en hetzelfde voor hoofdletters). Speciale tekens blijven onveranderd.

Voorbeeld

```
>>> decode("def ABC", 3)
'abc XYZ'
>>> decode("Cf Qptujujwf!", 1)
'Be Positive!'
```

Incrementele aanpak

1. Begin met het implementeren van `encode_kleine_letters(s, n)`. Test je code met voorbeelden die alleen kleine letters en speciale tekens bevatten.
2. Breid je code uit om ook hoofdletters te verwerken in `encode(s, n)`. Test je code met voorbeelden die kleine letters, hoofdletters en speciale tekens bevatten.
3. Implementeer de `decode(s, n)` functie. Test je code met voorbeelden die zowel versleutelde als niet-versleutelde strings bevatten.

Deze aanpak helpt je om de opgave in behapbare stappen op te delen en je code geleidelijk te testen. Succes!

C.4 Tweede prompt Gemini

Maak voor elk van deze stappen een testfunctie. Deze testfuncties moeten in de main aanroepen kunnen worden. Ik wil dat deze functies verschillende print statements hebben met de vorm:

```
print("parameter:".rjust(20), <arg>)
print("resultaat:".rjust(20), <functie(arg)>)
print("verwacht resultaat:".rjust(20), <verwacht resultaat>)
Print()
...
```

Print bij iedere functie ook uit wat er getest wordt. Ik verwacht dat je de vorm aanpast aan de opgave, dit wil zeggen dat bij een resultaat dat op meerdere lijnen uitgeprint hoort te worden, je dit ook op nieuwe regel uitprint. Geef me geen implementatie (code) van de opgave.

C.5 Antwoord Gemini tweede prompt

Oké, hier zijn de testfuncties voor elke stap van de Caesar Cipher-opgave, met de gevraagde indeling en print statements:

```
def test_encode_kleine_letters():
    print("Test encode_kleine_letters functie:")
    tests = [
        ("abc xyz", 3, "def abc", "Kleine letters verschuiven"),
        ("Be Positive!", 1, "Cf Qptujujwf!",
         "Hoofdletters en speciale tekens onveranderd"),
        ("", 5, "", "Lege string"),
        ("z", 1, "a", "Terug naar begin van alfabet"),
        ("abc", 26, "abc", "n is veelvoud van 26")
    ]

    for s, n, verwacht, beschrijving in tests:
        print(f"\nTest: {beschrijving}")
        print("parameter s:".rjust(20), s)
        print("parameter n:".rjust(20), n)
        resultaat = encode_kleine_letters(s, n)
        print("resultaat:".rjust(20), resultaat)
        print("verwacht resultaat:".rjust(20), verwacht)
        if resultaat != verwacht:
            print("Test mislukt!")

def test_encode():
```

```

print("\nTest encode functie:")
tests = [
    ("abc XYZ", 3, "def ABC", "Kleine en hoofdletters verschuiven"),
    ("Be Positive!", 1, "Cf Qptujujwf!",
     "Hoofdletters en speciale tekens onveranderd"),
    ("!@#", 5, "!@#", "Speciale tekens blijven onveranderd"),
    ("Z", 1, "A", "Hoofdletter Z verschuiven")
]

for s, n, verwacht, beschrijving in tests:
    print(f"\nTest: {beschrijving}")
    print("parameter s:".rjust(20), s)
    print("parameter n:".rjust(20), n)
    resultaat = encode(s, n)
    print("resultaat:".rjust(20), resultaat)
    print("verwacht resultaat:".rjust(20), verwacht)
    if resultaat != verwacht:
        print("Test mislukt!")

def test_decode():
    print("\nTest decode functie:")
    tests = [
        ("def ABC", 3, "abc XYZ", "Decoderen van kleine en hoofdletters"),
        ("Cf Qptujujwf!", 1, "Be Positive!",
         "Decoderen met hoofdletters en speciale tekens"),
        ("!@#", 5, "!@#", "Speciale tekens blijven onveranderd"),
        ("A", 1, "Z", "Hoofdletter A decoderen")
    ]

    for s, n, verwacht, beschrijving in tests:
        print(f"\nTest: {beschrijving}")
        print("parameter s:".rjust(20), s)
        print("parameter n:".rjust(20), n)
        resultaat = decode(s, n)
        print("resultaat:".rjust(20), resultaat)
        print("verwacht resultaat:".rjust(20), verwacht)
        if resultaat != verwacht:
            print("Test mislukt!")

def main():
    test_encode_kleine_letters()
    test_encode()
    test_decode()

if __name__ == "__main__":
    main()

```

Belangrijk: Deze code bevat geen implementatie van de `encode_kleine_letters`, `encode` en `decode` functies zelf. Je dient deze functies zelf te implementeren. De testfuncties roepen deze functies aan en vergelijken de resultaten met de verwachte uitkomsten.

De `main` functie roept alle testfuncties aan, zodat je alle tests in één keer kunt uitvoeren. Je kunt de tests aanpassen of uitbreiden met meer voorbeelden om je code grondig te testen.

C.6 Output test 1

```
>>> test_encode_kleine_letters()
Test encode_kleine_letters functie:

Test: Kleine letters verschuiven
      parameter n: 3
      resultaat: def abc
      verwacht resultaat: def abc

Test: Hoofdletters en speciale tekens onveranderd
      parameter s: Be Positive!
      parameter n: 1
      resultaat: Bf Pptjujwf!
      verwacht resultaat: Cf Qptujujwf!
Test mislukt!

Test: Lege string
      parameter s:
      parameter n: 5
      resultaat:
      verwacht resultaat:

Test: Terug naar begin van alfabet
      parameter s: z
      parameter n: 1
      resultaat: a
      verwacht resultaat: a

Test: n is veelvoud van 26
      parameter s: abc
      parameter n: 26
      resultaat: abc
      verwacht resultaat: abc
```

C.7 Output test 2

```
>>> test_encode()

Test encode functie:

Test: Kleine en hoofdletters verschuiven
      parameter s: abc XYZ
      parameter n: 3
      resultaat: def ABC
      verwacht resultaat: def ABC

Test: Hoofdletters en speciale tekens onveranderd
      parameter s: Be Positive!
      parameter n: 1
      resultaat: Cf Qptjujwf!
      verwacht resultaat: Cf Qptujujwf!
Test mislukt!
```

```
Test: Speciale tekens blijven onveranderd
      parameter s: !@#
      parameter n: 5
      resultaat: !@#
verwacht resultaat: !@#
```

```
Test: Hoofdletter Z verschuiven
      parameter s: Z
      parameter n: 1
      resultaat: A
verwacht resultaat: A
```

C.8 Output test 3

```
>>> test_decode()
```

Test decode functie:

```
Test: Decoderen van kleine en hoofdletters
      parameter s: def ABC
      parameter n: 3
      resultaat: abc XYZ
verwacht resultaat: abc XYZ
```

```
Test: Decoderen met hoofdletters en speciale tekens
      parameter s: Cf Qptujujwf!
      parameter n: 1
      resultaat: Be Postitive!
verwacht resultaat: Be Positive!
Test mislukt!
```

```
Test: Speciale tekens blijven onveranderd
      parameter s: !@#
      parameter n: 5
      resultaat: !@#
verwacht resultaat: !@#
```

```
Test: Hoofdletter A decoderen
      parameter s: A
      parameter n: 1
      resultaat: Z
verwacht resultaat: Z
```

Bijlage D

Kwantitatieve analyse input/output-paren

D.1 Opgave kwantitatieve analyse input/output-paren

Een priemgetal is een positief geheel getal dat deelbaar is door precies twee verschillende getallen, namelijk 1 en het priemgetal zelf. Het laagste en enige even priemgetal is 2. De eerste 10 priemgetallen zijn

2, 3, 5, 7, 11, 13, 17, 19, 23, 29

Input

Een getal $n \in \mathbb{N}_0$.

Uitvoer

Een zin die aangeeft of n een priemgetal is.

Tip

Test in een loop alle mogelijke delers van het getal. In deze loop kun je concluderen dat een getal geen priemgetal is zodra je een deler (anders dan 1 of het getal zelf) hebt gevonden. Je kunt echter alleen na afloop van de loop constateren dat het getal een priemgetal is, want dan heb je pas alle mogelijke delers getest.

Voorbeeld

Input:

11

Uitvoer:

11 is een priemgetal

Voorbeeld

Input:

12

Uitvoer:

12 is geen priemgetal

D.2 Eerste prompt ChatGPT

Dit is mijn opgave: Een priemgetal is een positief geheel getal dat deelbaar is door precies twee verschillende getallen, namelijk 1 en het priemgetal zelf. Het laagste en enige even priemgetal is 2. De eerste 10 priemgetallen zijn

2, 3, 5, 7, 11, 13, 17, 19, 23, 29

Input

Een geheel getal $n > 0$.

Uitvoer

Een zin die aangeeft of n een priemgetal is.

Tip

Test in een loop alle mogelijke delers van het getal. In deze loop kun je concluderen dat een getal geen priemgetal is zodra je een deler (anders dan 1 of het getal zelf) hebt gevonden. Je kunt echter alleen na afloop van de loop constateren dat het getal een priemgetal is, want dan heb je pas alle mogelijke delers getest.

Voorbeeld

Input:

11

Uitvoer:

11 is een priemgetal

Voorbeeld

Input:

12

Uitvoer:

11 is geen priemgetal

Geef een vereenvoudigde versie van de opgave. Die ik op een incrementele aanpak kan uitbreiden. Bijvoorbeeld: een oefening om tekst in allemaal hoofdletters te zetten, terwijl er ook speciale tekens voorkomen. Wordt: Eerst enkel kleine letters uitwerken, dan uitbreiden met hoofdletters, en vervolgens uitbreiden met speciale tekens. Ik wil geen code zien, aangezien ik dit zelf moet implementeren.

D.3 Antwoord ChatGPT eerste prompt

Hier is een vereenvoudigde versie van je opgave, die je stapsgewijs kunt uitbreiden:

Stap 1: Bepaal of een getal groter dan 1 is

Begin met een programma dat controleert of een gegeven geheel getal groter dan 1 is. Als het kleiner of gelijk aan 1 is, geef dan aan dat het géén priemgetal is.

Stap 2: Controleer deelbaarheid door 2

Voeg toe dat het programma kijkt of het getal deelbaar is door 2. Als het getal precies 2 is, dan is het een priemgetal. Als het een groter even getal is, dan is het géén priemgetal.

Stap 3: Controleer deelbaarheid door alle oneven getallen tot het getal zelf

Breid de controle uit door te testen of het getal deelbaar is door oneven getallen vanaf 3 tot het getal zelf (of tot een bepaald maximum, bijvoorbeeld de vierkantswortel van het getal).

Stap 4: Optimaliseer de controle

In plaats van elk oneven getal tot het getal zelf te testen, probeer enkel te testen tot de vierkantswortel van het getal. Dit maakt de controle efficiënter.

Stap 5: Geef de juiste output

Zorg ervoor dat het programma correct aangeeft of het getal een priemgetal is of niet, in de vorm van een volledige zin zoals gevraagd in de originele opgave.

Op deze manier bouw je stap voor stap je programma op, terwijl je de complexiteit geleidelijk verhoogt.

D.4 Tweede prompt ChatGPT

Maak voor elk van deze stappen verschillende input-output paren. Dit zodat ik kan testen of dat mijn implementatie correct is. Ik wil dat ik de input makkelijk kan kopiëren.

D.5 Antwoord ChatGPT tweede prompt

Hier zijn testgevallen voor elke stap. Je kunt de input gemakkelijk kopiëren en plakken om je implementatie te testen.

—

Stap 1: Bepaal of een getal groter dan 1 is

Testgevallen: [v] Geldige invoer → geeft een betekenisvolle uitkomst [x] Ongeldige invoer → geeft aan dat het geen priemgetal is

Input:

1

Output:

1 is geen priemgetal

Input:

0

Output:

0 is geen priemgetal

Input:

-5

Output:

-5 is geen priemgetal

Input:

2

Output:

2 is een priemgetal

Input:

3

Output:

3 is een priemgetal

—

Stap 2: Controleer deelbaarheid door 2

Testgevallen: [v] Herkent 2 als priemgetal [v] Herkent alle andere even getallen als ****geen**** priemgetal

Input:

2

Output:

2 is een priemgetal

Input:

4

Output:

4 is geen priemgetal

Input:

6

Output:

6 is geen priemgetal

Input:

10

Output:

10 is geen priemgetal

—

Stap 3: Controleer deelbaarheid door alle oneven getallen tot het getal zelf

Testgevallen: [v] Controleert deelbaarheid door oneven getallen vanaf 3

Input:

3

Output:

3 is een priemgetal

Input:

5

Output:

5 is een priemgetal

Input:

9

Output:

9 is geen priemgetal

Input:

15

Output:

15 is geen priemgetal

Input:

17

Output:

17 is een priemgetal

—

Stap 4: Optimaliseer de controle (tot de vierkantswortel)

Testgevallen: [v] Zelfde testgevallen als vorige stap, maar sneller uitgevoerd

Input:
25
Output:
25 is geen priemgetal

Input:
29
Output:
29 is een priemgetal

Input:
49
Output:
49 is geen priemgetal

Input:
53
Output:
53 is een priemgetal

Stap 5: Geef de juiste output in een volledige zin

Testgevallen: [v] Output in juiste vorm zoals in de originele opgave

Input:
31
Output:
31 is een priemgetal

Input:
33
Output:
33 is geen priemgetal

Input:
97
Output:
97 is een priemgetal

Input:
100
Output:
100 is geen priemgetal

Met deze testgevallen kun je stapsgewijs controleren of jouw implementatie correct werkt!