

Machine Learning Pipeline Automation

Amar Kisoensingh

16 juni 2021

Voorwoord

Voor u ligt de scriptie "Machine Learning Pipeline Automation". Deze scriptie is geschreven in het kader van mijn afstuderen aan de opleiding Informatica aan de Hogeschool Rotterdam en in opdracht van het stagebedrijf NGTI. De afstudeerstage liep van februari 2021 tot juni 2021.

De onderzoeksvraag is bedacht door mijn bedrijfsbegeleider Okke van 't Verlaat. Halverwege de afstudeerstage heeft Kolja van der Vaart de bedrijfsbegeleiding overgenomen. Terugkijkend was de vraag complex en ambitieus. Ik heb me beziggehouden met machine learning, automatisering en cloud platformen.

Bij deze wil ik mijn afstudeerbegeleiders Okke van 't Verlaat, Kolja van der Vaart en vanuit school Stelian Paraschiv en Marian Slingerland bedanken voor de zorgvuldige begeleiding. Ik heb met Okke en Kolja effectief kunnen sparren over het onderzoek. Mijn begeleiders hebben mij ook geholpen om de scriptie in de juiste richting te sturen. Tot slot wil ik mijn familie en vrienden bedanken voor de kritische en motiverende feedback.

Ik wens u veel leesplezier toe.

Amar Kisoensingh

Zoetermeer, 16 juni 2021

Samenvatting

Een probleem met machine learning (ML) is dat het gebruik ervan tijd en kennis vereist. Ook is er kans op vendor lock-in als ML wordt gebruikt op een cloud platform zoals Google Cloud of Microsoft Azure. Het is dus vrijwel lastig om te verhuizen naar een andere cloud platform.

Het doel van deze scriptie is om niet alleen te onderzoeken in hoeverre het mogelijk is om ML te automatiseren, maar ook of dit kan ongeacht het cloud platform. De onderzoeksvraag kan als volgt opgesteld worden: kan ML geautomatiseerd worden op een cloud platform-agnostische wijze?

Om deze vraag te beantwoorden is er onderzoek gedaan naar ML pipelines. Hiervoor is literatuuronderzoek en experimenteren van pas gekomen. Met ML pipelines kan op een gestructureerd manier gewerkt worden met ML. Dit verlaagt de leercurve en maakt ML toegankelijker voor nieuwe developers. Naast het onderzoek is ook een proof of concept (PoC) gebouwd om de platform-agnostische kant te valideren. Met het onderzoek en experimentatie dat is gedaan blijkt dat het mogelijk is om ML te automatiseren op een cloud platform-agnostische manier.

Het onderzoek kan verder uitgebreid worden door te kijken naar hoe ML toegankelijker gemaakt kan worden voor developers en hoe het systeem robuuster nog opgezet kan worden.

Summary

One problem with machine learning (ML) is that using it requires time and knowledge. There is also a chance of vendor lock-in if ML is being used on a cloud platform such as Google Cloud or Microsoft Azure. It is very hard to move to another cloud platform.

The aim of this thesis is not only to investigate to what extent ML can be automated, but also whether this is possible regardless of the cloud platform. The research question can be formulated as follows: can ML be automated in a cloud platform agnostic way.

To answer this question, research has been done on ML pipelines. Literature research and experimentation were utilized. With ML pipelines you can work with ML in a structured way. This not only makes ML more accessible, but lowers the learning curve for new developers. In addition to the research, a proof of concept (PoC) has also been built to validate the platform-agnostic aspect. The research and experimentation that has been done shows that it is possible to automate ML in a cloud platform agnostic way.

The research can be further expanded by looking at how ML can be made more accessible for developers and how the system can be set up more robustly.

Afkortingen

API application programming interface 5, 26, 27, 34, 40–42

CLI command line interface 40

CRUD create, read, update en delete 57

DRY don't repeat yourself 6, 57, 58

ERD entity relation diagram 43

IaC infrastructure as code 5, 32, 33

MLPA machine learning pipeline automation 40

MVP minimal viable product 46–48

PaaS Platform as a Service 15, 16

REST representational state transfer 34

SPA single page application 5, 40, 41

SVC support vector classification 5, 25

UI user interface 48, 50

VM virtual machine 6, 44, 54–56, 59, 79–81

Begrippenlijst

artifact 6, 48, 55–57, 59

scope creep 47

Lijst van figuren

1.1	Organogram van NGTI op 29-03-2021 [3].	11
1.2	Screenshot van de Swiss Climate Challenge app [5].	12
1.3	Screenshot van de My Swisscom App [7]	12
4.1	Machine learning in de context van andere domeinen	21
4.2	Lifecycle van een model volgens Hapke en Nelson [14, p. 4].	22
4.3	Voorbeeld van integer encoding	24
4.4	Voorbeeld van hot encoding	24
4.5	Gamma hyperparameter van een support vector classification (SVC) algoritme met waarde 0.1	25
4.6	Gamma hyperparameter van een support vector classification (SVC) algoritme met waarde 100	25
4.7	Model uitgerold met behulp van een API.	27
4.8	Vorm van expliciete feedback gebruikt door YouTube.	28
4.9	Machine learning pipeline met een algoritme exploratie tussenstap.	29
4.10	Machine learning pipeline waarbij stappen geautomatiseerd kunnen worden voor het gemak van developers.	30
5.1	Voorbeeld van een infrastructure as code (IaC) plan [24].	32
5.2	Proces van een infrastructure as code (IaC) om aan de gewenste situatie te voldoen.	33
5.3	Sequence diagram van het experiment met orkestratietool Pulumi	35
5.4	Node.js server voor het experiment met orkestratietool Pulumi	36
5.5	De stack voor het experiment	36
5.6	POST endpoint van de Pulumi experiment	37
6.1	Voorbeeld van een C4 model [38].	39
6.2	Context niveau diagram van het architecturaal ontwerp.	40
6.3	Container niveau diagram van het architecturaal ontwerp.	41
6.4	Single page application (SPA) component niveau diagram van het architecturaal ontwerp.	41
6.5	Node.js server component niveau diagram van het architecturaal ontwerp.	42
6.6	Sequence diagram van het aanmaken van een pipeline.	43
6.7	Sequence diagram van het starten van een pipeline.	44
7.1	Requirements opgesteld voor de proof of concept.	46
7.2	Trello bord in het begin van sprint 1	47
7.3	Mock up van de pipeline details pagina.	48
7.4	Gedrag van Pulumi met een incorrect plan.	49
7.5	Aanmaak dialoog van een pipeline.	50
7.6	Service om een pipeline aan te maken.	51
7.7	Functie om een storage bucket aan te maken in Google Cloud.	51
7.8	Status en upload-mogelijkheid van datasets en artefacts in de details pagina van een pipeline.	52
7.9	Service om datasets te uploaden naar een opslaglocatie binnen een cloud platform.	52
7.10	Functie om datasets te uploaden naar een storage bucket in Google Cloud.	53

7.11	Configuratie dialoog van een pipeline.	53
7.12	Code om de configuratie op te slaan in de database.	54
7.13	Status en startknop van een pipeline.	54
7.14	Code om een virtual machine (VM) in Google Cloud te starten.	55
7.15	Start-up script van een virtual machine (VM) in Google Cloud.	56
7.16	Model en overige artifacts van het trainproces geüpload in een storage bucket.	57
7.17	Voorbeeld van de repository-service pattern.	57
7.18	Voorbeeld van de don't repeat yourself (DRY) principe in de backend.	58
7.19	Voorbeeld van de don't repeat yourself (DRY) principe in de frontend.	58
7.20	Mock up van een verbeterd configuratie dialoog.	60
1	Mindmap van machine learning algoritmes	72
2	Stack configuratie van het experiment met Pulumi	75
3	Mock up van het pipeline overzicht	76
4	Mock up van het aanmaken van een pipeline	76
5	Mock up van het configuratie dialoog	77
6	Validation sequence diagram	78
7	Code om de status van een VM op te halen in Google Cloud	79
8	virtual machine (VM) output in de frontend	80
9	Code om een virtual machine (VM) te verwijderen in Google Cloud	81

Lijst van tabellen

3.1	Onderzoeks methode deelvraag 1	18
3.2	Onderzoeks methode deelvraag 2	18
3.3	Onderzoeks methode deelvraag 3	18
3.4	Onderzoeks methode hoofdvraag	19
5.1	Knock-out criteria voor orkestratietools dat cloud computing platformen beheerd.	33
5.2	Knock-out criteria tegen orkestratietools dat cloud computing platformen beheerd.	34
1	Scope deelvraag 1	69
2	Scope deelvraag 2	70
3	Scope deelvraag 3	70
4	Scope hoofdvraag	71

Inhoudsopgave

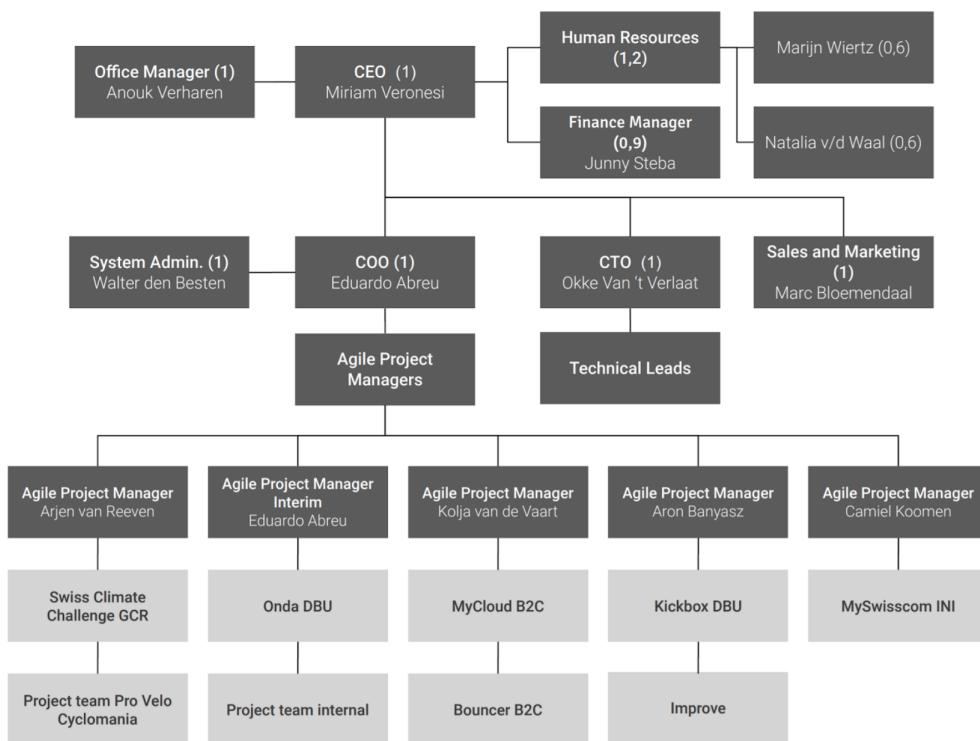
1 Inleiding	10
1.1 Projecten van NGTI	11
1.2 Tools die worden gebruikt	12
1.3 Aanleiding opdracht	13
2 Probleemanalyse	14
2.1 Obstakels voor NGTI	15
2.2 Vooronderzoek naar bestaande oplossingen	16
2.3 Doelstelling	16
2.4 Hoofd- en deelvragen	16
3 Onderzoeksmethoden en scope	17
4 Stappen in een machine learning pipeline	20
4.1 Wat is machine learning?	21
4.2 De stappen in een machine learning pipeline	22
4.3 Conclusie	27
4.4 Advies	28
5 Orkestratietools om platformen te beheren	31
5.1 Wat is infrastructure as code?	32
5.2 Orkestratietools vergeleken met elkaar	33
5.3 Experiment met de orkestratietool Pulumi	34
5.4 Conclusie	37
6 Architecturale ontwerp van de oplossing	38
6.1 Het C4 model	39
6.2 Architecturaal ontwerp	40
6.3 Sequence diagrammen	43
7 Proof of concept	45
7.1 User requirements verzamelen	46
7.2 Mock up van de proof of concept	48
7.3 Wijzigingen in het architecturaal ontwerp	48
7.4 De proof of concept	50
7.5 Kwaliteit van de code	57
7.6 Conclusie	59
7.7 Advies	59
8 Discussie	61
9 Reflectie	63
Bijlagen	68
I Scope deelvraag 1	69

II	Scope deelvraag 2	70
III	Scope deelvraag 3	70
IV	Scope hoofdvraag	71
V	Mindmap van machine learning algoritmes	72
VI	ML theorie	73
VII	Machine learning pipeline experiment	74
VIII	Stack configuratie van het experiment met Pulumi	75
IX	Mock up - Overzicht pipeline en pipeline aanmaken	76
X	Mock up - Configuration dialoog	77
XI	Validation sequence diagram	78
XII	Proof of concept code - <i>gc_getRunStatus()</i>	79
XIII	Proof of concept code - VM output in de frontend	80
XIV	Proof of concept code - <i>gc_stopRun</i>	81

1 Inleiding

NGTI is een software ontwikkelbedrijf dat gevestigd is in Rotterdam. Opgericht in 2012 maakt NGTI applicaties (apps) voor mobiel en/of webgebruik. Naast het ontwikkelen werkt NGTI aan het hele traject om een app heen, namelijk de probleemstelling, mockups, wireframes en prototyping. Daarnaast levert NGTI ook support en lost bugs op nadat een app live is gegaan [1]. Ook maakt NGTI white label apps en frameworks [2].

Het bedrijf heeft, zoals de meeste bedrijven, een organogram (Figuur 1.1). In de praktijk is dit echter niet terug te vinden en wordt de structuur gezien als "plat". Collega's kunnen elkaar laagdrempelig benaderen waardoor niemand een onbekende is en verschillende disciplines makkelijk met elkaar samen kunnen werken.



Figuur 1.1: Organogram van NGTI op 29-03-2021 [3].

NGTI is een dochterbedrijf van Swisscom [4]. Sinds maart 2021 is het bekend gemaakt dat Swisscom van plan is om een afdeling, Swisscom DevOps Center, te fuseren met NGTI. Omdat de fusie onzekerheid met zich meebrengt voor de structuur en manier hoe NGTI werkt, zal de situatie vóór de fusie aangehouden worden gedurende het afstuderen.

1.1 Projecten van NGTI

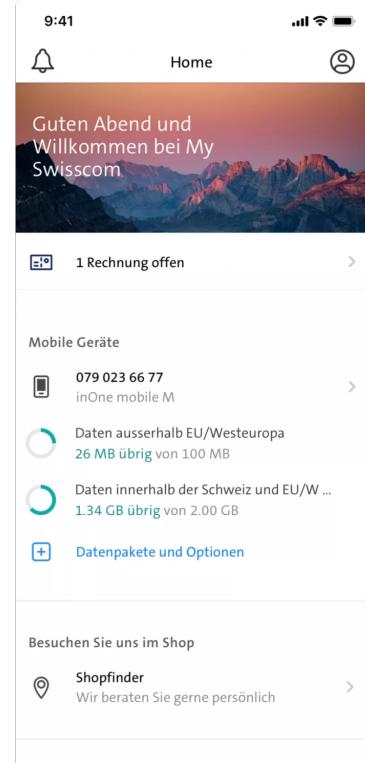
NGTI heeft een vrij breed portfolio met apps voor verschillende doeleinden. Een van deze apps is de Climate Challenge App [5]. Met deze app kunnen gebruikers hun CO₂-voetafdruk en impact in kaart brengen. Er wordt bijgehouden hoeveel kilometer de gebruiker reist en met welk vervoersmiddel. De app is onderdeel van twee bestaande nieuwsapps, Blick en Bluewin [6]. Het

doel is om de gebruiker aan te sporen om groener te reizen. Een screenshot van de app is te zien in Figuur 1.2.

Een andere oplossing is de My Swisscom App [7]. Dit is een native app voor Android en iOS waarbij Swisscom-klanten hun contract kunnen bestellen, wijzigen of beëindigen. In de app kunnen klanten ook de dataverbruik zien en instellingen voor abonnementen wijzigen. Een screenshot van de app is te zien in Figuur 1.3.



Figuur 1.2: Screenshot van de Swiss Climate Challenge app [5].



Figuur 1.3: Screenshot van de My Swisscom App [7]

1.2 Tools die worden gebruikt

Om productief te zijn, gebruikt NGTI een aantal tools en programma's om producten te maken en te communiceren met zowel collega's als klanten. De meest gebruikte en belangrijkste zijn Slack, Google Workspace, Zoom en Microsoft Teams.

1.2.1 Slack

Interne communicatie gaat via Slack. Het programma faciliteert collega's om elkaar met een lage instap te benaderen en berichten die voor het hele bedrijf relevant zijn te versturen. Ook zijn er 'channels' beschikbaar over specifieke onderwerpen, zoals: `#dev`, `#ios` en `#test-automation`.

1.2.2 Google Workspace

Met Google Workspace kunnen bestanden en documenten gemaakt, opgeslagen en gedeeld worden. Dit is mogelijk via een browser waardoor werknemers geen software hoeven te installeren. NGTI gebruikt het ook om collaboratief en parallel te werken aan hetzelfde document.

1.2.3 Microsoft Teams en Zoom

Voorheen werd Zoom alleen gebruikt om te videobellen met collega's en geïnterviewden. In de tijd van de pandemie is Zoom echter een belangrijke speler geworden om effectief samen te werken. Meetings zoals introducties van nieuwe collega's of demo's van producten worden online gehouden.

1.3 Aanleiding opdracht

NGTI wilt de gebruikerservaring van haar apps verbeteren en een voorsprong hebben op haar concurrenten. Dit kan NGTI op een aantal manieren doen waarvan apps "slimmer" maken er een van is. Het slimmer maken houdt voor NGTI in dat de app bijvoorbeeld beter kan anticiperen wat de gebruiker wilt en nodig heeft op een gegeven moment. Dit kan onder andere door het toepassen van machine learning (ML). De opdracht kan verdeeld worden in twee onderdelen:

- Onderzoek naar hoe een pipeline opgezet kan worden op een cloud computing platform door middel van een framework
- Onderzoek naar het maken van een platform-agnostische oplossing

Daarnaast zal een proof-of-concept (PoC) gemaakt worden om aan te tonen of het haalbaar is in de praktijk. Een diepere duik in het probleem en het definiëren van de onderdelen is te vinden in hoofdstuk 2.

2 Probleemanalyse

Zoals beschreven in paragraaf 1.3 wilt NGTI ML toepassen om haar apps slimmer te maken. Hier zijn een aantal redenen voor, onder andere om de gebruikerservaring te verbeteren en om een voorsprong te hebben op concurrenten.

Het 'slimmer' maken van applicaties kan op verschillende manieren, maar met machine-learning kan een platform gebouwd worden waarmee elke richting op gegaan kan worden. Om machine-learning te implementeren in haar applicaties loopt NGTI tegen een aantal obstakels aan, namelijk: expertise vereist in het ML domein, benodigde tijd om een pipeline op te zetten en vendor lock-in.

2.1 Obstakels voor NGTI

NGTI loopt tegen de voorgenoemde obstakels aan omdat NGTI weinig ervaring heeft met ML. In samenwerking met een extern bedrijf worden modellen getraind die vervolgens gebruikt worden in apps van NGTI. Om zelf modellen te trainen heeft NGTI kennis over ML, ML pipelines en tijd nodig. Bovendien wilt NGTI niet bij één cloud computing platform haar infrastructuur opzetten maar gemakkelijk kunnen schakelen tussen platformen. De obstakels worden in de volgende koppen verduidelijkt.

2.1.1 Expertise Machine Learning

ML is ingewikkeld en diepgaand onderwerp. Om een model te trainen is kennis nodig van verschillende domeinen: data mining, software engineering en statistieken. Doordat er voorkennis nodig is om een model te trainen en een pipeline goed op te zetten, is het vaak te hoogdrempelig voor developers om een start te maken met ML. De expertise is daarnaast niet in een korte tijd te vergaren.

2.1.2 Opzetten pipeline

Bovenop de complexiteit van ML zelf bestaan er verschillende manieren om een model te trainen. Het opzetten van ML pipelines is daar een van. Een pipeline is een workflow dat bestaat uit een aantal stappen die doorgelopen worden om een model te trainen. In elke stap worden acties uitgevoerd, zoals het verwijderen van onbruikbare data of de prestatie van modellen vergelijken en een rapport met uitslagen genereren. Het opzetten van zo een pipeline én de actie(s) in de stappen definiëren kost tijd en vereist specifieke kennis. Daarnaast zijn de stappen en acties vaak hetzelfde voor verschillende pipelines. Het automatiseren en hergebruiken van stappen en acties tussen pipelines zou tot onder andere tijdwinst en het verminderen van herhaling van code kunnen leiden.

2.1.3 Vendor lock-in

Er bestaan een aantal diensten, zogenoemde Platform as a Service (PaaS), waarbij je een pipeline kan opzetten en acties kan definiëren. Een van de problemen met een PaaS is vendor lock-in. Dit betekent dat, als er eenmaal een pipeline is opgezet, de overdraagbaarheid van de pipeline naar een andere PaaS vrijwel onmogelijk is. Ook zijn de opties en mogelijkheden om uit te breiden in de toekomst gelimiteerd.

2.2 Vooronderzoek naar bestaande oplossingen

Het gebruik van ML pipelines is geen nieuwe techniek en is mogelijk bij PaaS en bedrijven die zich specialiseren in ML pipelines. Het gemak met zulke services is dat de gebruiker gelijk kan beginnen met het trainen van ML modellen en zich niet zorgen hoeft te maken over infrastructurele details. Echter is er sprake van vendor lock-in en kunnen services van deze bedrijven niet gebruikt worden.

Gedurende de vooronderzoek zijn "Infrastructure as Code" frameworks naar voren gekomen. Dit zijn frameworks waarmee, middels code, een infrastructure binnen een Platform as a Service (PaaS) opgezet kan worden. Dit kan gebruikt worden als onderdeel van de PoC en wordt in een van de deelvragen verder onderzocht.

2.3 Doelstelling

Om haar doel, de gebruikerservaring van apps verbeteren en een voorsprong hebben op concurrent, te bereiken is NGTI van plan ML pipelines te gebruiken om ML toe te passen. De gewenste oplossing is een systeem waarbij developers met weinig tot geen kennis een model kunnen trainen. Het systeem moet de infrastructurele taken voor zich nemen, zoals het opzetten van een pipeline en de stappen en acties automatiseren. Daarnaast moet een ML pipeline pijnloos doorlopen kunnen worden op verschillende platformen zodat het systeem platform-agnostisch is.

2.4 Hoofd- en deelvragen

Uitgaand van de drie obstakels kan de hoofdvraag als volgt worden geformuleerd:

*In welke mate kan een machine learning pipeline worden
geautomatiseerd onafhankelijk van het onderliggende cloud computing
platform?*

De hoofdvraag kan worden onderbouwd met vier deelvragen. Om te beginnen is het verstandig om te weten welke stappen er in een machine learning pipeline zit:

Waar bestaat een machine learning pipeline uit?

Daarnaast is een framework nodig dat, door middel van code, verschillende cloud computing platformen kan beheren:

*Hoe kan een framework verschillende cloud computing platformen beheren om een
machine learning pipeline op te zetten?*

Ten slotte wordt een PoC gemaakt om te laten zien of het probleem oplosbaar is. Hiervoor is een doordachte voorbereiden onmisbaar:

*Hoe ziet de architecturale blauwdruk van een applicatie, waarmee een
platform-onafhankelijk machine learning pipeline opgezet kan worden, eruit?*

3 Onderzoeksmethoden en scope

Om elke hoofd- en deelvraag te beantwoorden, wordt er bij elk gebruik gemaakt van een onderzoeks methode. Volgens Scribbr [8] zijn er twee onderzoeks methoden: kwantitatief en kwalitatief. Bij een kwantitatief onderzoeks methode wordt data verzameld waarmee bijvoorbeeld grafieken of tabellen gemaakt kunnen worden. De focus bij een kwalitatief onderzoeks methode ligt bij het verzamelen van verschillende interpretaties en opvattingen. Hierop kan optioneel een eigen interpretaties op gemaakt worden. [9].

Onder kwantitatief en kwalitatief vallen verschillende dataverzamelings methoden. Deze beschrijft simpelweg de manier hoe data wordt verzameld. Dit kan bijvoorbeeld met een enquête, literatuuronderzoek op websites en in boeken of een onderzoek over een lange periode [9].

Elke hoofd- en deelvraag is gekoppeld aan een onderzoeks methoden. Vervolgens is beschreven welk(e) dataverzamelings methode(n) wordt gebruikt met een korte toelichting. Daarnaast wordt op een hoog niveau de scope bepaald.

D1: Waar bestaat een machine learning pipeline uit?	
Methode(s)	Kwalitatief
Dataverzamelings methode(n)	Literatuuronderzoek, fundamenteel onderzoek, toegepast onderzoek
Scope	Bijlage I

Tabel 3.1: Onderzoeks methode deelvraag 1

D2: Hoe kan een orkestratietool verschillende cloud computing platformen beheren om een machine learning pipeline op te zetten?	
Methode	Kwalitatief
Dataverzamelings methode(n)	Literatuuronderzoek, vergelijkend onderzoek
Scope	Bijlage II

Tabel 3.2: Onderzoeks methode deelvraag 2

D3: Hoe ziet de architecturale blauwdruk van een applicatie, waarmee een platform-onafhankelijk machine learning pipeline opgezet kan worden, eruit?	
Methode	Kwalitatief
Dataverzamelings methode(n)	Literatuuronderzoek
Scope	Bijlage III

Tabel 3.3: Onderzoeks methode deelvraag 3

H: In welke mate kan een machine learning pipeline worden geautomatiseerd onafhankelijk van de onderliggende cloud computing platform?	
Methode	Kwalitatief
Dataverzamelingsmethode(n)	Literatuuronderzoek
Scope	Bijlage IV

Tabel 3.4: Onderzoeks methode hoofdvraag

4 Stappen in een machine learning pipeline

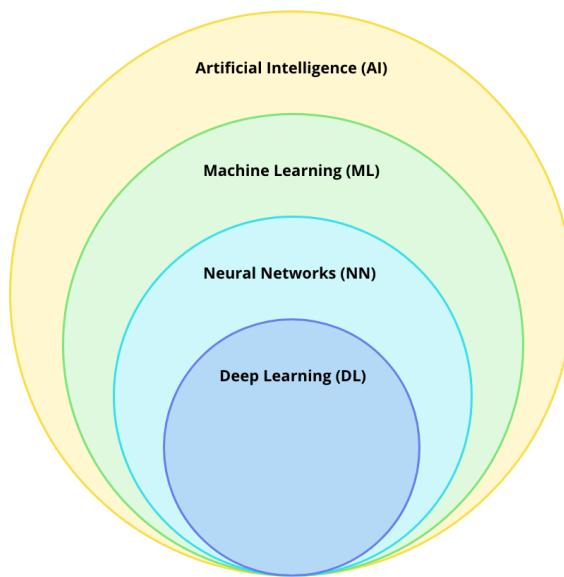
Een ML pipeline is zoals beknopt beschreven in deelparagraaf 2.1.2 een collectie van stappen dat wordt doorlopen om een model te trainen. Elke stap bevat een aantal acties dat wordt uitgevoerd, zoals onbruikbare data weghalen of de prestatie analyseren. De stappen en acties worden in paragraaf 4.2 uitgelegd met behulp van theorie en een experiment dat is uitgevoerd. Het volledige experiment is te vinden in bijlage VII. Een van de stappen in een pipeline is het trainen van het model. Om een idee te krijgen van ML en het trainen van modellen zal dit kort uitgelegd worden.

4.1 Wat is machine learning?

ML houdt in dat een computer een taak kan uitvoeren zonder daarvoor expliciet geprogrammeerd te zijn. Dit wordt gedaan door een ML model te laten leren van een gegeven dataset. Vervolgens kan er een voorspelling worden gemaakt [10, p. 1-3].

De domeinen deep learning (DL), neural networks (NN) en artificial intelligence (AI) komen vaak voor als het over ML gaat. Zoals weergegeven in Figuur 4.1 is te zien dat ML een subset is van AI, NN een subset van ML en als laatste DL dat een subset is van NN [11].

Bij AI wordt niet alleen ML toegepast, maar ook concepten zoals beredeneren, plannen, vooruitdenken, onthouden en terug refereren. Een voorbeeld hiervan is dat een ML model kan voorspellen wat het volgende woord in een zin kan zijn, maar een AI kan beredeneren waarom de zin gebouwd is zoals het is en hoe het binnen de context van de alinea past [12].



Figuur 4.1: Machine learning in de context van andere domeinen

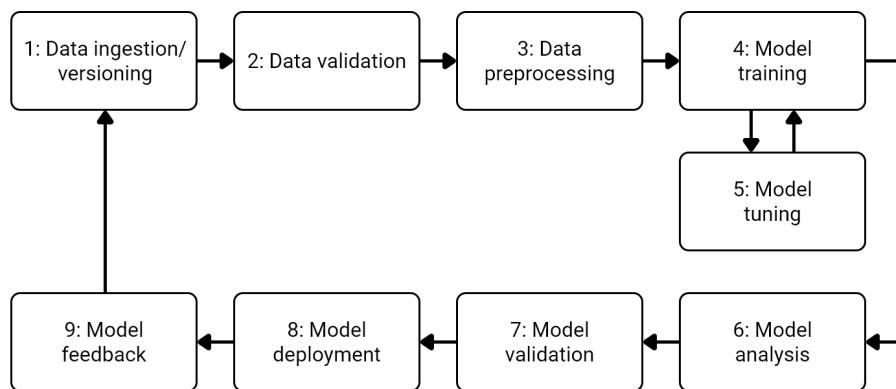
Een NN bestaat uit een collectie van nodes dat gemodelleerd is naar de hersenen. Een NN heeft minimaal 3 lagen: een inputlaag, een verborgen laag en een outputlaag. Elke laag bevat neuronen dat data als input kan krijgen en data als output aan de volgende laag meegeeft. DL is een NN dat meerdere verborgen lagen bevat [13].

4.1.1 Een manier om een machine learning model te trainen

Het trainen van een ML model is een proces waarbij vooraf data wordt opgeschoond en achteraf de prestatie van het model wordt gevalideerd. Normaliter wordt dit gedaan door specifieke code te schrijven voor deze taken. Een valkuil is dat de code niet bij elke developer werkt en schaalt niet in alle gevallen naar een productieomgeving. Een manier om dit wel te behalen is om te werken met ML pipelines. Zoals kort uitgelegd in deelparagraaf 2.1.2 bestaat een ML pipeline uit stappen en acties. Er is echter geen consensus binnen het ML domein over wat de juiste stappen en acties in een pipeline zijn. Voor de scriptie is gekozen voor een pipeline van Hapke en Nelson uit het boek "Building Machine Learning Pipelines". Het boek is gemaakt en gepubliceerd door O'Reilly; een bekend en gecrediteerd bedrijf dat boeken maakt binnen het software engineering domein.

4.2 De stappen in een machine learning pipeline

Een ML pipeline begint met het opnemen van data en eindigt met het ontvangen van feedback om de prestatie van het model te verbeteren. De pipeline bevat een aantal stappen zoals data voorbereiden, het model trainen en het uitrollen van het model (Figuur 4.2).



Figuur 4.2: Lifecycle van een model volgens Hapke en Nelson [14, p. 4].

In totaal zijn er, zonder de feedbackloop stap, acht stappen die elke keer doorlopen moeten worden om een model te trainen. In de volgende subkoppen zullen de stappen worden doorlopen met een korte uitleg over wat er gebeurd in een stap.

4.2.1 Stap 1: Dataopname en versiebeheer (Data ingestion/versioning)

De eerste stap in de pipeline is het opnemen van data. Met deze data zal het model getraind, gevalideerd en getest worden. De dataset kan van een of meerdere bronnen komen, zoals lokaal, een online opslag locatie of van een database. Het dataset kan gekopieerd worden en op een plek worden opgeslagen waar alle datasets te vinden zijn. Zodra de data op een bereikbare plek is opgeslagen en ingeladen in, moet het verdeeld worden tussen een train-, validatie- en testdataset. Normaal gebeurt dit met een splitratio van 6:2:2. De train dataset is 60% en de validatie en testdatasets zijn allebei 20% van de originele dataset [14, p. 27-37].

Een use case van een pipeline is dat een nieuw model getraind kan worden door een geüpdateerde dataset te gebruiken. Dit wordt gedaan door de voorgaande dataset te gebruiken, waarbij nieuwe data is toegevoegd. Door het gebruik van verschillende datasets is het verstandig om versiebeheer toe te passen. Zo is goed te zien welk dataset welke model produceert. Een versie geven aan een dataset gebeurt voordat de dataset wordt ingeladen [14, p. 39-40]. Versiebeheer voor datasets kan bijvoorbeeld met DVC [15] of Pachyderm [16]. Beide zijn frameworks om bij te houden waar datasets zijn opgeslagen, welke versie het is en welk model het heeft geproduceerd.

4.2.2 Stap 2: Datavalidatie (Data validation)

Nu de dataset verdeeld is, een versie heeft en op een bereikbare plek is, kan de data gevalideerd worden. Deze stap is vooral belangrijk om te voorkomen dat een model wordt getraind dat niet nuttig is aangezien het trainen veel tijd in beslag kan nemen. Een bekende uitdrukking is "garbage in = garbage out". Dit betekent dat als de dataset niet goed is, het model ook niet goed zal presteren [14, p. 43]. Tijdens de validatiestap wordt gecontroleerd op het volgende:

- Afwijkingen in de dataset
- Wijzigingen in de structuur
- Algemene statistieken in vergelijkingen met voorgaande datasets [14, p. 44]

Bij het controleren van afwijkingen in de dataset wordt gekeken naar waarden die opmerkelijk zijn. Afwijkende waarden liggen te ver van het gemiddelde en kunnen een verkeerd beeld schetsen bij het trainen van het model. Deze uitschieters kunnen simpelweg uit de dataset gefilterd worden.

Het kan voorkomen dat bij een nieuwe dataset de type van waarden zijn gewijzigd. Een *int* kan bijvoorbeeld veranderd zijn in een *string* of *boolean*. Er is dan sprake van een wijziging in de structuur van de dataset. Dit is problematisch omdat er een vertaalstap gemaakt moet worden naar iets bruikbaars. Als dit niet mogelijk is moeten de waarden uitgefilterd worden wat de prestatie van het model negatief kan beïnvloeden.

De algemene statistieken is een hulpmiddel om te controleren op afwijkingen en wijzigingen. Vaak kan de controle in een oogopslag gedaan worden.

4.2.3 Stap 3: Data voorbereiden (Data preprocessing)

Het voorbereiden van de dataset is een stap dat de prestatie van het model verbetert en het proces van het trainen versneld. Deze stap kan verdeeld worden in twee substappen: het opschonen en het optimaliseren van de dataset.

Bij het opschonen worden bijvoorbeeld duplicaten of waarden die onbruikbaar zijn uit de dataset weggehaald. Onbruikbare waarden zijn waarden die simpelweg niet kloppen of verkeerd zijn ingevoerd. Hierbij kan bijvoorbeeld gedacht worden aan een medewerker die de interactietijd met een klant moet bijhouden, maar is vergeten om de eindtijd te noteren. Voor het algoritme zal het dan lijken alsof de medewerker een klant tot sluitingstijd heeft geholpen.

Datasets kunnen geoptimaliseerd worden om twee redenen: algoritmes werken sneller met waarden die dichter bij 0 liggen en algoritmes kunnen niet met elke waarde in de dataset omgaan. Om de efficiëntie van het algoritme te verbeteren kunnen een aantal technieken gebruikt worden:

- Schalen

Bij het schalen van data worden de waarden getransformeerd die liggen tussen een schaal, zoals tussen 0 en 100 of 0 en 1. Bij het schalen wordt het **bereik** getransformeerd [17].

- Normaliseren

Als een dataset wordt gestandaardiseerd, worden de waarden getransformeerd in een standaard normale verdeling waarbij het gemiddelde 0 en de afwijking 1 is. Hierbij wordt de **vorm** van de dataset getransformeerd [17]. Het normaliseren is belangrijk als het algoritme waarden vergelijkt met verschillende eenheden [18]. In sommige gevallen is normalisering een vereiste bij een aantal algoritmes [19].

- Categorische codering

Een groot aantal algoritmes kan alleen omgaan met numerieke waarden. Het kan voorkomen dat datasets categorische waarden bevatten. Dit zijn waarden die een label voorstellen, zoals *first*, *second* of *third*. Om deze waarden te gebruiken moeten ze worden gecodeerd als een numerieke waarde. De label kan bijvoorbeeld als de waarde 1, 2 of 3 gecodeerd worden. Deze techniek heet label encoding of integer encoding en kan gebruikt worden als de volgorde van de codering klopt (deelparagraaf 4.2.3).

Categorical label	Encoded
<i>first</i>	1
<i>second</i>	2
<i>third</i>	3

Figuur 4.3: Voorbeeld van integer encoding

- Hot encoding

Een andere techniek om waarden te coderen is hot encoding. Deze techniek wordt gebruikt waarbij de labels geen relatie met elkaar hebben en maakt gebruik van meerdere waarden om een label te beschrijven (Figuur 4.2.3).

Animal	C1	C2	C3
<i>cat</i>	1	0	0
<i>dog</i>	0	1	0
<i>mouse</i>	0	0	1

Figuur 4.4: Voorbeeld van hot encoding

Dit is een goed moment om de getransformeerde dataset op te slaan als "tussen-dataset" zodat deze stap niet herhaald hoeft te worden. Het voorbereiden van grote datasets kan een grote

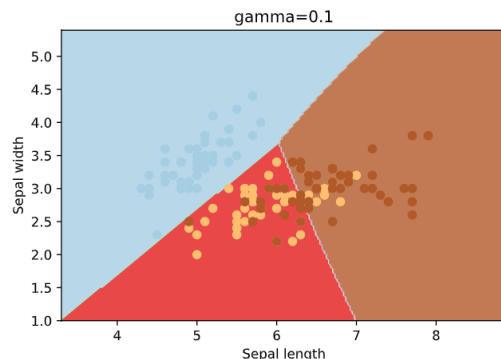
hoeveelheid tijd in beslag nemen. Nu de dataset is gevalideerd en voorbereid, kan het model getraind en getuned worden.

4.2.4 Stap 4 en 5: Model trainen en tunen (Model training and Model tuning)

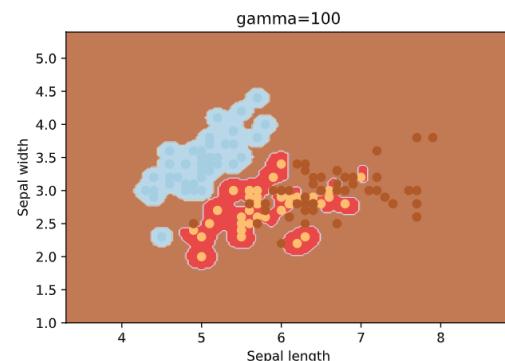
Het trainen van een model gaat door middel van een ML algoritme. Een ML algoritme is vergelijkbaar met een conventioneel algoritme in software engineering zoals bijvoorbeeld binary search, merge sort en depth first search. Het algoritme slaat na het trainen met de dataset regels, nummers en algoritme specifieke datastructuren op in de vorm van een model. Het algoritme kan worden gezien als een specifiek programma waarmee, gegeven een input, voorspellingen mee gedaan kunnen worden [20].

Elk algoritme heeft variabelen om er voor te zorgen dat het algoritme beter aansluit op de dataset. Deze variabelen heten *hyperparameters*. Het algoritme heeft standaard hyperparameters die niet altijd de optimale prestatie behalen. Op een heuristische wijze kunnen de optimale hyperparameters gevonden worden [21]. Dit proces heet *tunen*.

Tijdens het experiment zijn diagrammen gegenereerd met verschillende hyperparameters. In Figuur 4.5 en Figuur 4.6 is te zien hoe de *gamma* hyperparameter van een support vector classification (SVC) algoritme invloed heeft op het model. Dit algoritme classificeert op basis van gegeven waarden. De waarden worden geplot en zal binnen een van de drie kleuren vallen. Het model met als hyperparameter waarde 100 zal vaker de classificatie met de bruine kleur als voorspelling geven dan als de waarde 0.1 zou zijn.



Figuur 4.5: Gamma hyperparameter van een support vector classification (SVC) algoritme met waarde 0.1



Figuur 4.6: Gamma hyperparameter van een support vector classification (SVC) algoritme met waarde 100

In het ML domein bestaan talloze algoritmes om voorspellingen te maken. In Bijlage V is een mind-map te vinden van de algoritmes die gedurende de scriptie naar voren zijn gekomen. De algoritmes kunnen grotendeels gegroepeerd worden in vier stijlen: supervised, unsupervised, semi-supervised en reinforcement learning. In bijlage VI is meer informatie over de stijlen te vinden.

4.2.5 Stap 6 en 7: Modelanalyse en -validatie (Model analysis and Model validation)

Na het trainen en tunen kan analyse en validatie plaatsvinden. Bij het analyseren wordt de prestatie van het model vergeleken met voorgaande modellen. Om dit te doen kunnen, net als in deelparagraaf 4.2.2, statistieken gegenereerd worden van het model. De soort statistiek hangt af van wat voor soort algoritme is gebruikt. Een classificatie-algoritme heeft bijvoorbeeld andere statistieken dan een regressie-algoritme. Op basis van de uitkomst kunnen de dataset of hyperparameters aangepast worden om de accuraatheid te verhogen.

Met de validatie van een model moet er gekeken worden met een genuanceerd perspectief. Validatie gaat namelijk over hoe eerlijk een model is als er gekeken wordt naar een bepaalde groep. Een groep kan gedefinieerd worden als geslacht, etniciteit, locatie of leeftijd. Het kan namelijk voorkomen dat de dataset voornamelijk bestaat uit bijvoorbeeld vrouwen. Dit *kan* ervoor zorgen dat het model minder accurate voorspellingen maakt voor mannen [10, p. 109-110]. Een voorbeeld waar het gebruik van ML grote negatieve gevolgen had was de toeslagenaffaire in 2020. De Nederlandse overheid maakte gebruik van een systeem dat aangaf of een burger mogelijk bijstandsfraude had gepleegd. Het systeem maakte gebruik van ML om fraude aan te geven. Jaren na het gebruik bleek dat er sprake was van etnische profiling. De dataset bestond voornamelijk uit immigranten uit Islamitische landen. Het systeem heeft hierdoor een *bias* [22].

4.2.6 Stap 8: Model uitrollen (Model deployment)

Na het analyseren en valideren kan het model uitgerold worden. Het uitrollen kan op twee manieren: inbakken in een app of serveren met een application programming interface (API).

Met het inbakken in een app wordt bedoelt dat het model meegenomen wordt als de app gebouwd wordt. Voordelen van deze methode is dat het model offline bruikbaar is en de rekenkracht van het apparaat gebruikt kan worden om te voorspellen of classificeren. Een nadeel is dat het model niet makkelijk geüpdatet kan worden.

De andere optie is om het model te serveren door middel van een API. De app die het model wilt gebruiken zal een request moeten doen met input. De API geeft de input door aan het model. Het model geeft een predictie terug dat de API doorstuurt naar de applicatie. Het updaten van het model gaat gemakkelijker dan het model inbakken met een app. Een nadeel is de eis dat de app een internetverbinding moet hebben om het model te gebruiken [10, p. 130].

Bij het experiment is gekozen voor het uitrollen met een API. In Figuur 4.7 is te zien hoe een model geserveerd wordt via een API. Een endpoint genaamd "classify" is gedefinieerd waarop een *POST* request gedaan kan worden. De endpoint haalt de input uit de request en geeft het door aan het model. De model maakt vervolgens een predictie dat uiteindelijk naar de app teruggestuurd wordt.

```

1 @app.route("/classify", methods=["POST"])
2 def classify():
3     # Get input from request
4     json = request.get_json()
5     data = json["data"]
6
7     # Get prediction from model
8     classification = model.predict([data])
9
10    # Return prediction to app
11    return jsonify({ "family": classification[0] })

```

Figuur 4.7: Model uitgerold met behulp van een API.

4.2.7 Stap 9: Feedbackloop (Model feedback)

Om het model continue te verbeteren kan feedback verzameld worden. De feedback geeft een beeld van de effectiviteit van het model in een productieomgeving. Feedback kan verdeeld worden in twee soorten: impliciet en expliciet [10, p. 264].

Het verzamelen van impliciete feedback wordt gedaan zonder dat de gebruiker zich daar bewust van is. Dit wordt gedaan door bij te houden of een voorspelling correct was volgens een gebruiker. Een voorbeeld van deze methode is het bijhouden of een gebruiker een video kijkt dat door het algoritme voorgesteld is. Als een gebruiker de voorgestelde video bekijkt, wordt dit gezien als een succesvolle voorspellen. Mocht de gebruiker niet een voorgestelde video bekijken, wordt dit gezien als onsuccesvol.

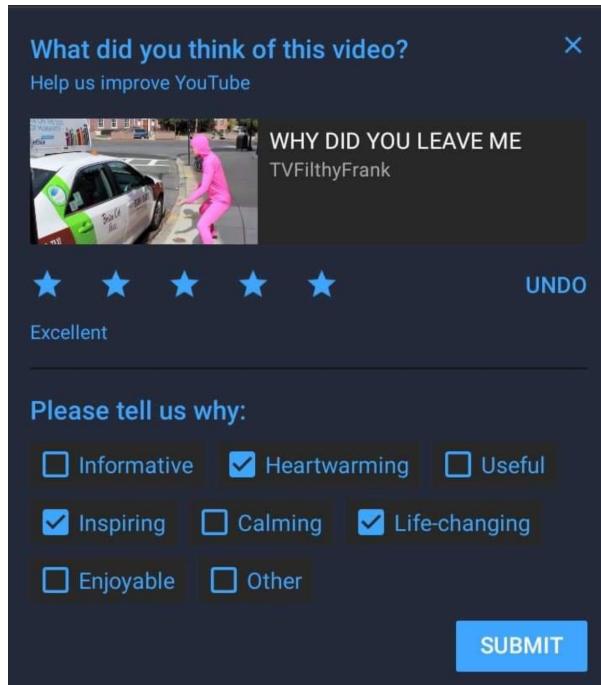
In tegenstelling tot impliciet wordt bij expliciete feedback gevraagd aan de gebruiker of een voorspelling gepast is. Een voorbeeld van deze vorm in de praktijk is hoe YouTube expliciete feedback vraagt (Figuur 4.8). Naast het aangeven of de voorspelling gepast was, kan de gebruiker ook aangeven waarom de video gepast was.

Het verzamelen van feedback is niet noodzakelijk maar helpt met het verbeteren van het model. Deze output van deze stap gaat samen met een nieuwe dataset naar de eerste stap in de lifecycle (Figuur 4.2). Met de nieuwe dataset en de feedback kan een nieuw model getraind worden dat beter presteert dan de voorganger.

4.3 Conclusie

In dit hoofdstuk is er onderzoek gedaan naar het antwoord op de deelvraag: **D1: Waar bestaat een machine learning pipeline uit?** Het onderzoek is uitgevoerd met behulp van zowel digitale bronnen, een boek en een experiment.

Een ML pipeline bestaat uit verschillende stappen die op hun beurt bestaan uit acties. De stappen zijn gericht op het waarborgen van de kwaliteit van de dataset en model, het voorbereiden van de dataset en het trainen van het model. Daarnaast kan het model uitgerold worden in een productieomgeving en kan er feedback verzameld worden over de prestatie van het model volgens



Figuur 4.8: Vorm van expliciete feedback gebruikt door YouTube.

gebruikers.

De acties in de stappen moeten bij het eerste gebruik van de pipeline gedefinieerd worden. De acties kunnen bijvoorbeeld het filteren van onbruikbare data, een model trainen of statistieken genereren zijn. Als dit vast staat, kan de pipeline hergebruikt worden om nieuwe modellen te trainen met nieuwe datasets. Op deze manier is een ML pipeline een gestructureerde en reproduceerbaar proces.

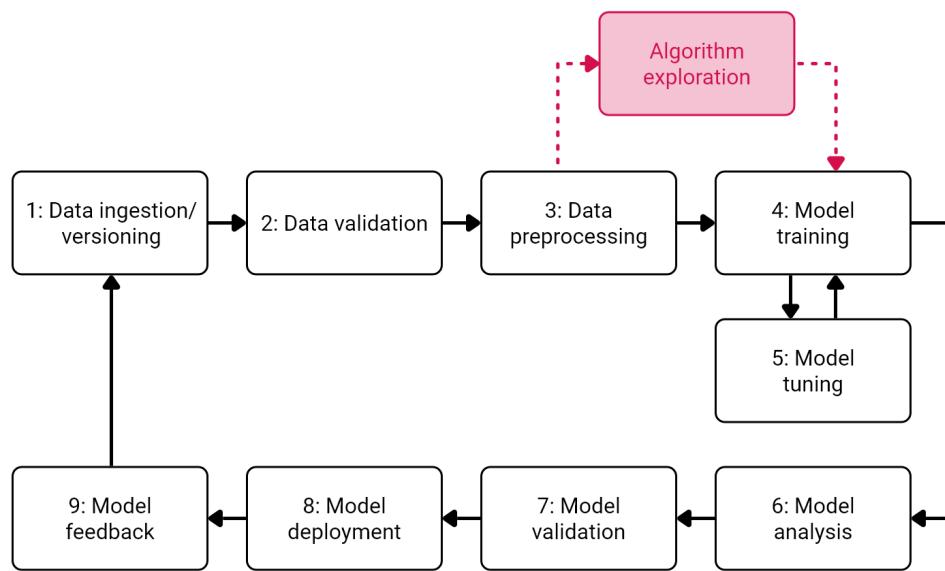
4.4 Advies

Het in gebruik nemen van een ML pipeline werkwijze is initieel ingewikkeld, maar lonend op de lange termijn. Het opzetten van de pipeline en de acties definiëren kost tijd en kennis. Echter als dit eenmaal gedaan is, kan eenvoudig een verbeterd model worden getraind door een nieuwe dataset aan te leveren. Een verbeterd model is vrijwel gegarandeerd omdat de stappen in de ML pipeline de dataset en het model analyseren en valideren.

4.4.1 Een betere keuze maken tussen machine learning algoritmes

Zoals eerder aangegeven in paragraaf 4.1 bestaat er niet één ML pipeline dat correct is. Een ML pipeline is een werkwijze waarbij de stappen anders geïnterpreteerd kunnen worden. Hierdoor is er geen regelmaat tussen verschillende pipelines. Een pipeline kan stappen of acties weglaten of extra hebben. Overeenkomst tussen pipeline is ook niet haalbaar aangezien stappen gemaakt kunnen worden die specifiek zijn voor het algoritme of workflow van het bedrijf. De stappen in de pipeline van Hapke en Nelson (Figuur 4.2) is een valide basis, maar kan uitgebreid worden om de ervaring van developers te verbeteren. In stap 4 en 5 (deelparagraaf 4.2.4) wordt een

model getraind waarbij het beste algoritme om het ML probleem op te lossen al bekend is. De verwachting is dat de keuze/onderzoek vóór het starten van de pipeline is gedaan. Tussen stap 3 en 4 kan echter een stap tussen zitten om te experimenteren met een combinatie van verschillende algoritmes en hyperparameters. In Figuur 4.9 is te zien hoe zo een stap zou passen in de lifecycle. De stap is met een stippellijn aangegeven omdat de stap de eerste keer meegenomen kan worden als de pipeline wordt doorlopen. Na de eerste keer zijn de beste algoritme en hyperparameters al bekend en kan deze stap overgeslagen worden.

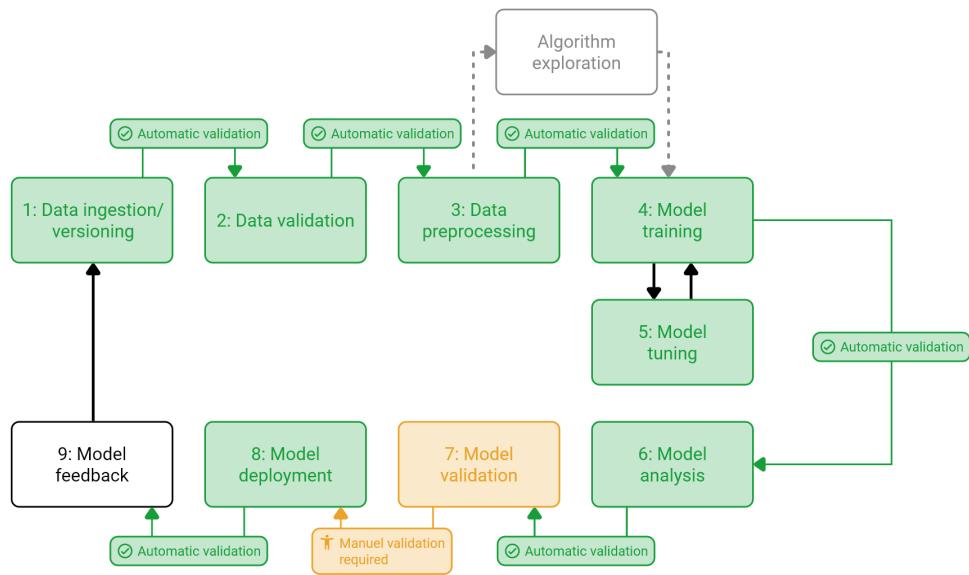


Figuur 4.9: Machine learning pipeline met een algoritme exploratie tussenstap.

In de "Algorithm exploration" stap kunnen verschillende algoritmes die hetzelfde doel bereiken en een variatie van hyperparameters voor elk algoritme getest worden tegen de dataset. Uit deze tests kan een prestatiescore gegenereerd worden om vervolgens te kiezen voor het algoritme dat het meest geschikt lijkt. Na de keuze kan de pipeline verder gaan met de volgende stappen.

4.4.2 Machine learning versimpelen

Een van de focuspunten is om te onderzoeken in hoeverre ML te vergemakkelijken is voor developers. Met de huidige pipeline is er vrij veel kennis vereist over ML om ermee te werken. Toch kan een groot deel van de stappen geautomatiseerd worden. Tussen elke stap kan een validatie plaatsvinden met een aantal randvoorwaarden waaraan voldaan moeten worden om door te gaan naar de volgende stap. In Figuur 4.10 is te zien hoe tussen elke stap een validatie plaats vindt. Om bijvoorbeeld van stap 6 (Model analysis) naar stap 7 (Model validation) te gaan, moet het model even goed of zelfs beter presteren dan de voorganger. Hoe veel beter het model moet kunnen presteren is aan een developer. Mocht het zo zijn dat het model niet voldoet, kan de pipeline stopgezet worden.



Figuur 4.10: Machine learning pipeline waarbij stappen geautomatiseerd kunnen worden voor het gemak van developers.

Niet alle aspecten van de pipeline kunnen versimpeld worden. Een van de stappen die niet kan is stap 7 (Model validation) in Figuur 4.10. Deze stap vereist input van een mens aangezien de bias van het model wordt beoordeeld. Wel kunnen rapporten over de bias gegenereerd worden zodat de workflow van developers gestroomlijnd wordt.

5 Orkestratietools om platformen te beheren

Om resources in een cloud platform te beheren kan gebruik gemaakt worden van een portaal. In een portaal kunnen databases, servers en netwerken aangemaakt worden om een infrastructuur te creëren. Omdat een portaal uitgebreid en complex kan zijn, moet de developer een infrastructuur kunnen opzetten zonder het gebruik van een portaal. Dit is mogelijk met infrastructure as code (IaC) orkestratietools. IaC zal als eerst worden uitgelegd. Vervolgens worden verschillende IaC orkestratietools met elkaar vergeleken. Als laatst wordt een experiment uitgevoerd om de haalbaarheid met een orkestratietool te valideren.

5.1 Wat is infrastructure as code?

IaC is een manier om een infrastructuur op te zetten binnen een cloud platform zonder gebruik te maken van een interface zoals een portaal. De gewenste infrastructuur wordt beschreven waarna de orkestratietool de gewenste situatie een realiteit probeert te maken [23].

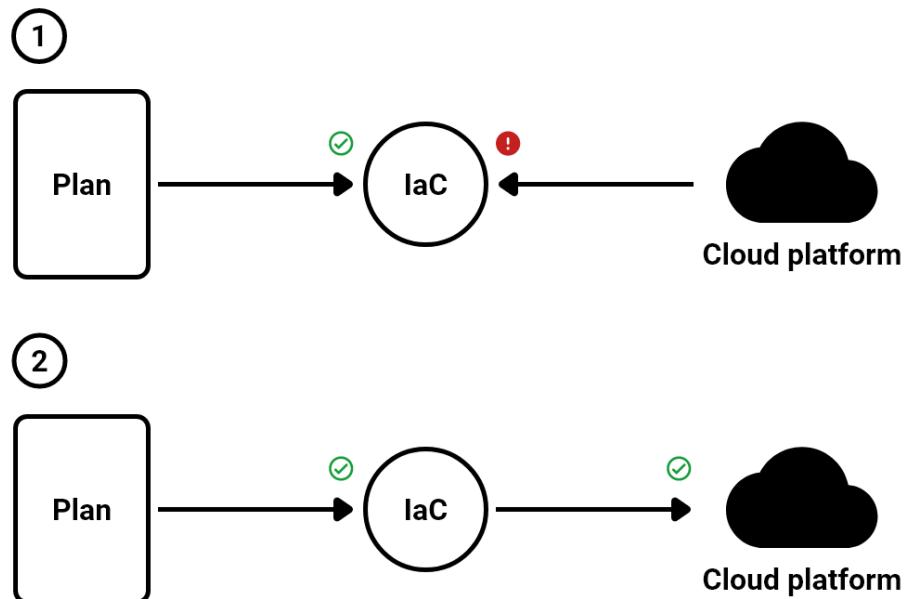
De manier waarmee een orkestratietool de infrastructuur aanmaakt, wijzigt of verwijderd is met een plan. In het plan staat welke resources moeten bestaan om aan de gewenste situatie te voldoen. Vaak gaat dit gepaard met configuratie zoals een naam, locatie of tags. In Figuur 5.1 is een voorbeeld van een plan als een YAML bestand te zien. YAML is een bestandstype dat vaak wordt gebruikt voor configuratie. Het plan beschrijft op welke cloud platform dit uitgevoerd moet worden (lijn 12) en wat er aangemaakt moet worden. In dit geval is het een resource group (lijn 16) met als configuratie een naam (*myTFResourceGroup*) en locatie (*westus2*).

```
1 terraform {
2   required_providers {
3     azurerm = {
4       source  = "hashicorp/azurerm"
5       version = "> 2.26"
6     }
7   }
8
9   required_version = "> 0.14.9"
10 }
11
12 provider "azurerm" {
13   features {}
14 }
15
16 resource "azurerm_resource_group" "rg" {
17   name      = "myTFResourceGroup"
18   location  = "westus2"
19 }
```

Figuur 5.1: Voorbeeld van een infrastructure as code (IaC) plan [24].

Een IaC volgt een proces om aan de gewenste situatie te voldoen. In Figuur 5.2 is te zien hoe zo een proces verloopt. Zodra de orkestratietool start, wordt zowel het plan als de situatie in de cloud platform gecontroleerd en met elkaar vergeleken. In stap 1 is het plan valide, maar de situatie in de cloud platform niet. Dit betekent bijvoorbeeld dat er in het plan staat dat er een resource group moet bestaan met een naam en locatie zoals het beschreven staat in Figuur 5.1, maar dit niet bestaat in de cloud platform. In de volgende stap wordt de resource aangemaakt

door de IaC om aan de gewenste situatie te voldoen. De wijzigingen gebeuren programmatisch en wordt gedaan door de IaC. Er is geen tussenkomst van een persoon of interface nodig.



Figuur 5.2: Proces van een infrastructure as code (IaC) om aan de gewenste situatie te voldoen.

5.2 Orkestratietools vergeleken met elkaar

Om een keuze te maken tussen IaC orkestratietools kan zogenoemde "knockout" criteria opgesteld worden. Een IaC orkestratietool wordt niet valt af zodra de tool niet voldoet aan één van de criteria. In Tabel 5.1 zijn de knockout criteria te vinden. De criteria in samenwerking met NGTI opgesteld.

Criteria	Toelichting
Programmatisch beheren	Het orkestratietool moet via code een resources kunnen aanmaken, wijzigen en verwijderen.
Ondersteuning voor cloud computing platformen	Om platform-agnostisch te zijn moet de orkestratietool minstens twee cloud computing platformen ondersteunen waarop een ML model getraind kan worden.
Uitgebreide documentatie	De documentatie moet toegankelijk en duidelijk zijn. De documentatie moet tutorials, concepten en een reference bevatten.

Tabel 5.1: Knock-out criteria voor orkestratietools dat cloud computing platformen beheert.

Voor de keuzes uit de tools is onderzoek gedaan op het internet en een enquête geraadpleegd van Stack Overflow [25]. De enquête is ingevuld door developers en bevat vragen over de ervaring met frameworks, technologieën en de website zelf. Uit de vraag over welke frameworks, libraries en tools het meest gebruikt werkt, kwam Ansible, Terraform, Puppet en Chef naar boven [26]. Volgens een vraag over de meest geliefd en gevreesde frameworks, libraries en tools zijn Ansible en Terraform het meest geliefd en Puppet en Chef het meest gevreesd [27]. Uit algemeen onderzoek naar orkestratietools is, naast de bovengenoemde tools, Pulumi naar voren gekomen als kandidaat [28]. Ansible, Terraform en Pulumi zijn in Tabel 5.2 tegen de knockout criteria gehouden.

Orkestratie-tools	Programmatisch beheren	Ondersteuning voor cloud computing platformen	Uitgebreide documentatie
Ansible	Nee, orkestratie gaat via een YAML bestand [29]	AWS, Google Cloud en Azure. 11 platformen in totaal [30]	Uitgebreide documentatie met tutorials, references en migration guide [31]
Pulumi	Ja, orkestratie via CLI of TypeScript, JavaScript, Python, .NET en Go [32]	AWS, Google Cloud en Azure. 13 platformen in totaal [33]	Uitgebreide documentatie met tutorials en references [34]
Terraform	Nee, orkestratie gaat via een YAML bestand [35]	AWS, Google Cloud en Azure. 5 platformen in totaal [36]	Uitgebreide documentatie met tutorials, references en videos [37]

Tabel 5.2: Knock-out criteria tegen orkestratietools dat cloud computing platformen beheert.

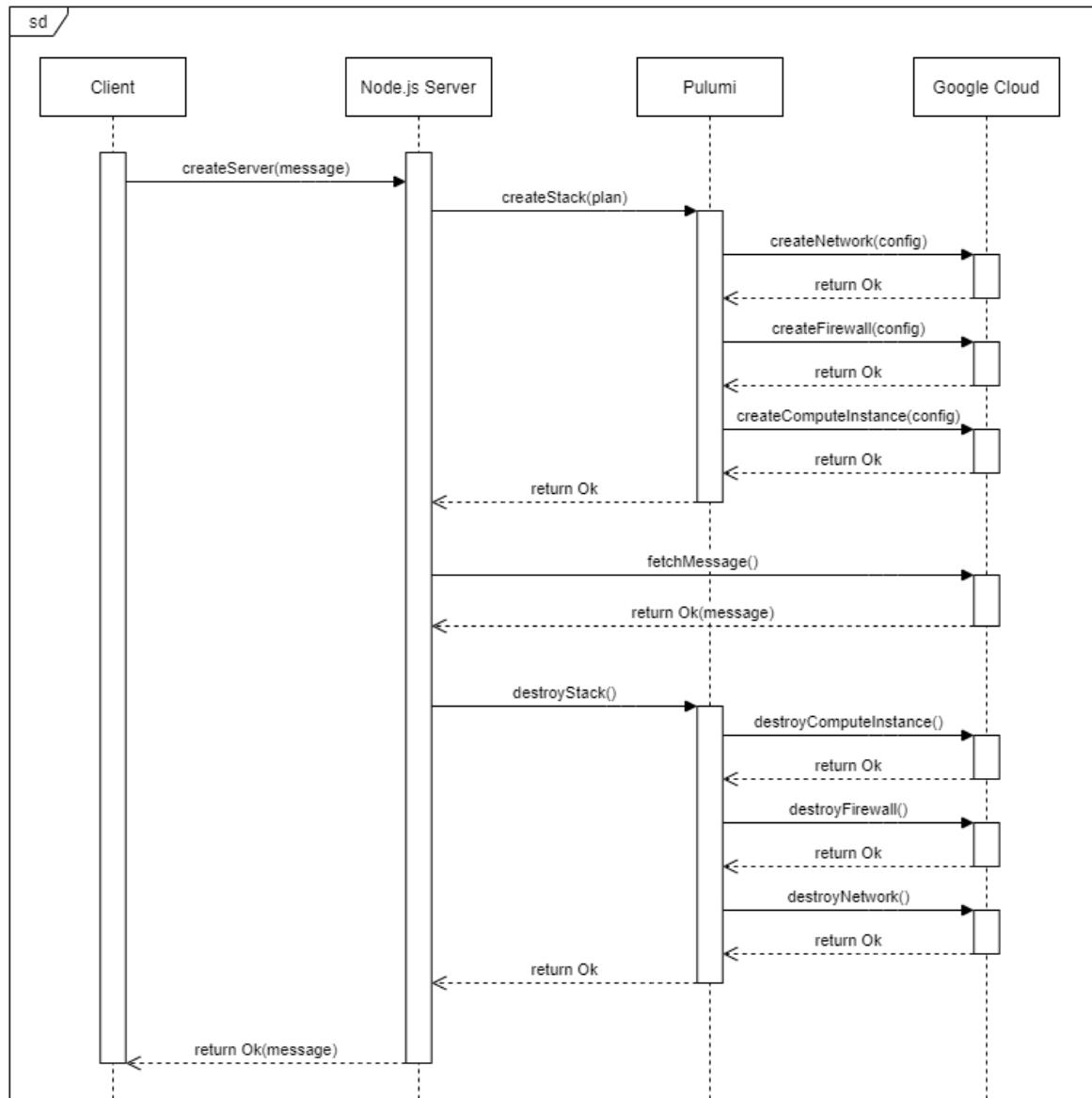
In Tabel 5.2 is te zien dat de drie orkestratietools gelijk zijn als het gaat om de ondersteuning van cloud platform en documentatie. Programmatisch beheren kan met Ansible en Terraform alleen door middel van een YAML bestand in tegenstelling tot Pulumi dat via code een plan kan uitvoeren. Omdat het praktischer voor de PoC is om via code een plan uit te voeren, is er voor gekozen om met Pulumi verder te werken. In het volgende hoofdstuk zal een experiment met Pulumi uitgevoerd worden om te valideren of Pulumi geschikt is.

5.3 Experiment met de orkestratietool Pulumi

Het idee achter het experiment is om te achterhalen of Pulumi een geschikte orkestratietool is. Op dit moment in het afstudeerproces wordt er verwacht dat de PoC een frontend krijgt dat praat door middel van een representational state transfer (REST) API met een backend. De backend zal vervolgens gebruik maken van de orkestratietool om te praten met een cloud platform. Om het PoC na te bootsen op kleinere schaal wordt er bij het experiment gebruik gemaakt van een client waarbij een opdracht met behulp van een REST API naar een Node.js server wordt verstuurt. De Node.js server stuurt om zijn beurt opdrachten naar Pulumi die communiceert met Google Cloud. Voor het experiment is arbitrair gekozen voor het cloud platform; de PoC moet ten slotte platform-agnostisch zijn.

Het experiment is als sequence diagram uitgewerkt in Figuur 5.3. Hierin is goed te zien welke

acties gedaan worden en door welke speler. In de sequence diagram is ook te zien wat voor taak wordt uitgevoerd aan de hand van de naam tussen de spelers.



Figuur 5.3: Sequence diagram van het experiment met orkestratietool Pulumi

De Node.js server is vrij simpel en bevat één endpoint op lijn 12 genaamd /api/run-python-code (Figuur 5.4). In deze *POST* endpoint wordt een stack aangemaakt, een *GET* request uitgevoerd naar de aangemaakte compute instance en vervolgens wordt de stack weer verwijderd. Een stack is de manier hoe Pulumi een plan beschrijft.

```
1 import express, { response } from 'express'
2 import fetch from 'node-fetch'
3 import { compute } from '@pulumi/gcp'
4 import { LocalWorkspace } from "@pulumi/pulumi/automation"
5
6 const app = express()
7 const port = 8080
8
9 app.use(express.json())
10 app.use(express.urlencoded({ extended: true }))
11
12 app.post("/api/run-python-code", async (req, res) => { ... })
13
14 app.listen(port, () => {
15   console.log(`server started at http://localhost:${port}`)
16 })
```

Figuur 5.4: Node.js server voor het experiment met orkestratietool Pulumi

In de stack worden drie resources aangemaakt: een network, een firewall en een compute instance (Figuur 5.5). De compute instance moet een HMTL bestand serveren met een bericht dat de client heeft gespecificeerd.

```
1 const CreateResource = (message: string) => async () => {
2   // Create a network
3   const computeNetwork = new compute.Network("network", {
4     // Configuration
5   })
6
7   // Create a firewall in the network
8   const computeFirewall = new compute.Firewall("firewall", {
9     // Configuration
10  })
11
12  // Define script that runs when the compute instance runs
13  const startupScript = `#!/bin/bash
14 echo "${message}" > index.html
15 nohup python -m SimpleHTTPServer 80 &
16
17  // Create a compute instance that runs startupScript
18  const computeInstance = new compute.Instance("instance", {
19    // Configuration
20  }, { dependsOn: [computeFirewall] })
21
22  const ip = computeInstance.networkInterfaces.apply(ni => ni[0].accessConfigs![0].natIp);
23
24  return { name: computeInstance.name, ip: ip }
25 }
```

Figuur 5.5: De stack voor het experiment.

Elke resource heeft zijn eigen configuratie om ervoor te zorgen dat het HTML bestand beschikbaar is voor de buitenwereld. De code met de volledige configuratie is te vinden in bijlage VIII.

In Figuur 5.6 is te zien wat er in de POST endpoint gebeurd. Als eerst wordt een stack in geheugen aangemaakt met behulp van de code in Figuur 5.5. Na het aanmaken wordt de stack uitgevoerd met de *stack.up()* functie op lijn 12. Nadat de stack is aangemaakt in Google Cloud wordt een *GET* request uitgevoerd naar de server op lijn 15. Nadat de request voltooid is wordt de stack verwijderd uit Google Cloud (lijn 21) met *stack.destroy()* en wordt het bericht teruggestuurd naar de client.

```
1 app.post("/api/run-python-code", async (req, res) => {
2   const stackName = 'mlpa-py'
3   const projectName = 'disco-sky-312109'
4
5   const stack = await LocalWorkspace.createOrSelectStack({
6     stackName,
7     projectName,
8     program: CreateResource(req.body.message)
9   })
10
11 // Comparing stack against Google Cloud
12 const upRes = await stack.up({ onOutput: console.info })
13
14 // Fetch HTML file with message
15 const action_response = await fetch(`http://${upRes.outputs.ip.value}`)
16   .then(async r => await r.text())
17   .then(t => t)
18   .catch(e => console.log(e))
19
20 // Destroying stack in Google Cloud
21 await stack.destroy({ onOutput: console.info })
22
23 res.send({ returned_response: action_response })
24 })
```

Figuur 5.6: POST endpoint van de Pulumi experiment

5.4 Conclusie

In dit hoofdstuk is er onderzoek gedaan naar het antwoord op de deelvraag: **D2: Hoe kan een orkestratietool verschillende cloud computing platformen beheren om een machine learning pipeline op te zetten?** Het onderzoek is uitgevoerd met behulp van digitale bronnen en een experiment.

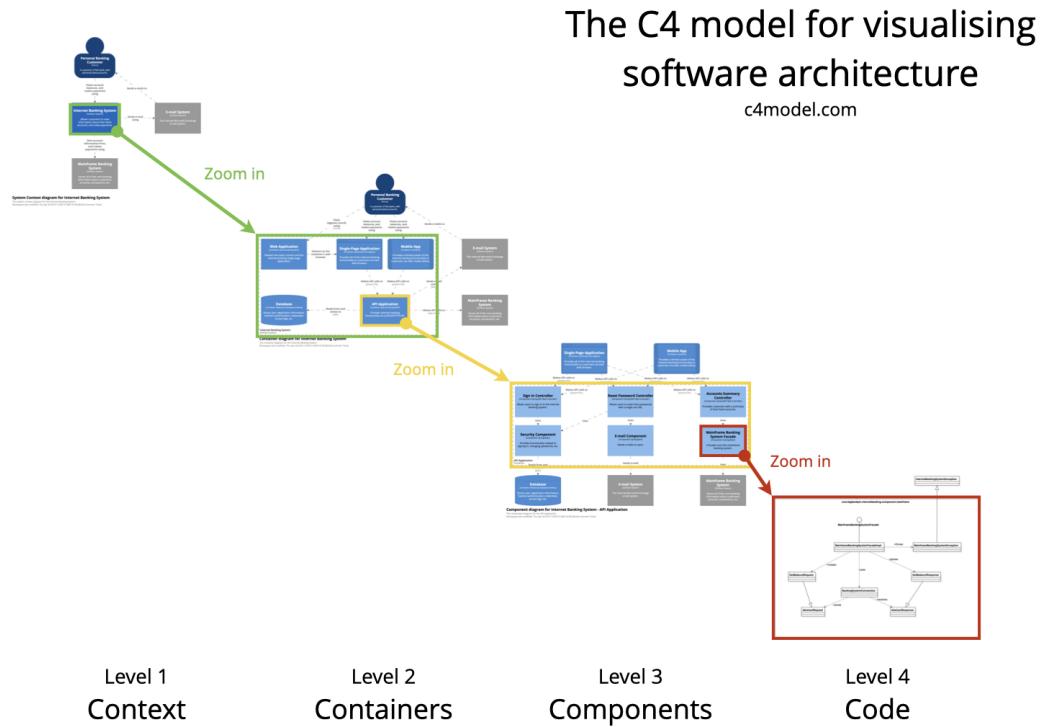
Een orkestratietool is een manier om resources aan te maken, aan te passen en te verwijderen. De tool doet dit zonder het gebruik van een online portaal of interventie van een developer. Het managen van resources wordt gedaan op basis van een plan en de huidige situatie binnen de cloud platform. In het plan staat welke resources moeten bestaan en met welke configuratie. Het plan kan beschreven worden in een YAML bestand of door middel van code. De orkestratietool vergelijkt vervolgens het plan met de huidige situatie en zorgt ervoor dat de situatie voldoet aan het plan.

6 Architecturale ontwerp van de oplossing

Nu het bekend is hoe ML pipelines en orkestratietools werken, kan er gekeken worden naar hoe de PoC architecturaal eruit gaat zien. De technische tekeningen laten zien hoe verschillende componenten in de PoC interacteert met elkaar en met componenten buiten de scope van de PoC. De tekeningen zijn gemaakt volgens het C4 model [38]. In dit hoofdstuk zal worden uitgelegd hoe C4 gelezen moet worden waarna de technische tekeningen wordt uitgelegd.

6.1 Het C4 model

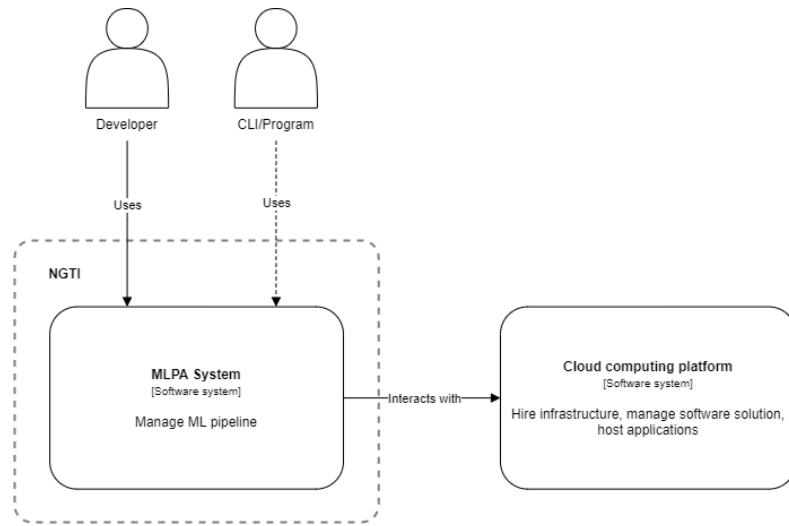
Het C4 model is een notatietechniek om de architectuur van een systeem te communiceren aan andere developers en is bedacht door Simon Brown [38]. Bij het tekenen van de modellen wordt er gedacht in vier niveaus: context, containers, components en code. In Figuur 6.1 is een voorbeeld van een C4 model te zien. De context level is het eerste niveau waarbij wordt gekeken naar het systeem, gebruikers en contexten buiten de scope. Het systeem binnen de scope kan uitgebreid worden door middel van het volgende niveau: containers. Containers laten op een hoog niveau zien waar het systeem uit bestaat. Elk container kan verder verduidelijkt worden met het component niveau. Hierbij worden containers verder opgebroken in components. Het laatste niveau is het code niveau en wordt doorgaans weergegeven via een klassendiagram.



Figuur 6.1: Voorbeeld van een C4 model [38].

6.2 Architecturaal ontwerp

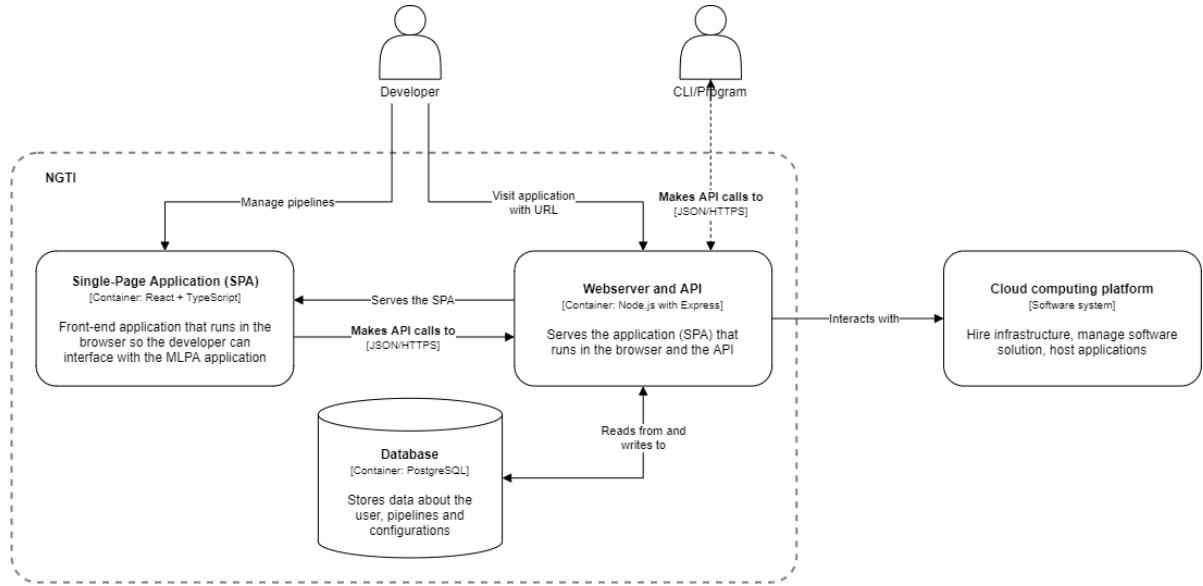
Op context niveau is het vrij simpel met het machine learning pipeline automation (MLPA) systeem binnen de scope waarmee een developer interacteert en cloud computing platformen die buiten de scope vallen (Figuur 6.2). In dit diagram is ook te zien dat developers alleen gebruik maken van het systeem en indirect van cloud platformen. In de context diagram is ook een command line interface (CLI)/program te zien dat gebruik kan maken van de MLPA systeem. Deze valt echter buiten de scope en is daarom aangegeven met een stippellijn.



Figuur 6.2: Context niveau diagram van het architecturaal ontwerp.

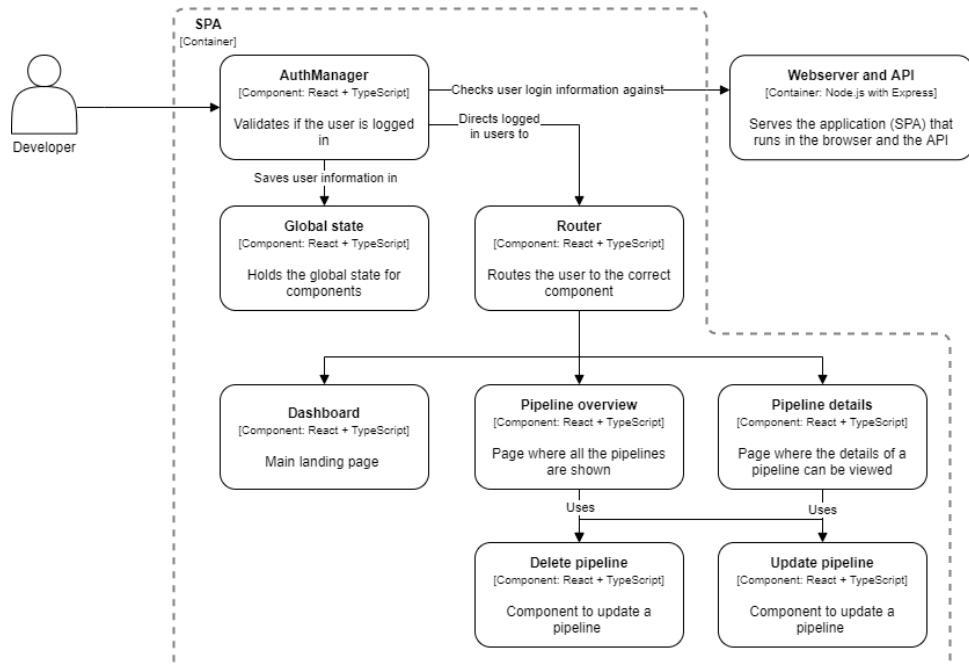
In Figuur 6.3 is het container niveau van het MLPA systeem te zien. Het systeem bestaat uit een single page application (SPA) frontend, een Node.js met Express backend en een PostgreSQL database. Een developer krijgt van de backend de SPA geserveerd waarmee een pipeline beheert kan worden. De frontend stuurt API requests naar de backend die vervolgens interacteert met de database en cloud platformen.

Voor de keuze van frameworks is rekening gehouden met de populariteit en kwaliteit van documentatie. Voor de populariteit is nogmaals de enquête van Stack Overflow geraadpleegd [25]. Volgens de resultaten scoren zowel React.js, Node.js en Express hoog [39] [26]. Daarnaast hebben alle drie frameworks robuuste documentatie met references en tutorials [40] [41] [42].



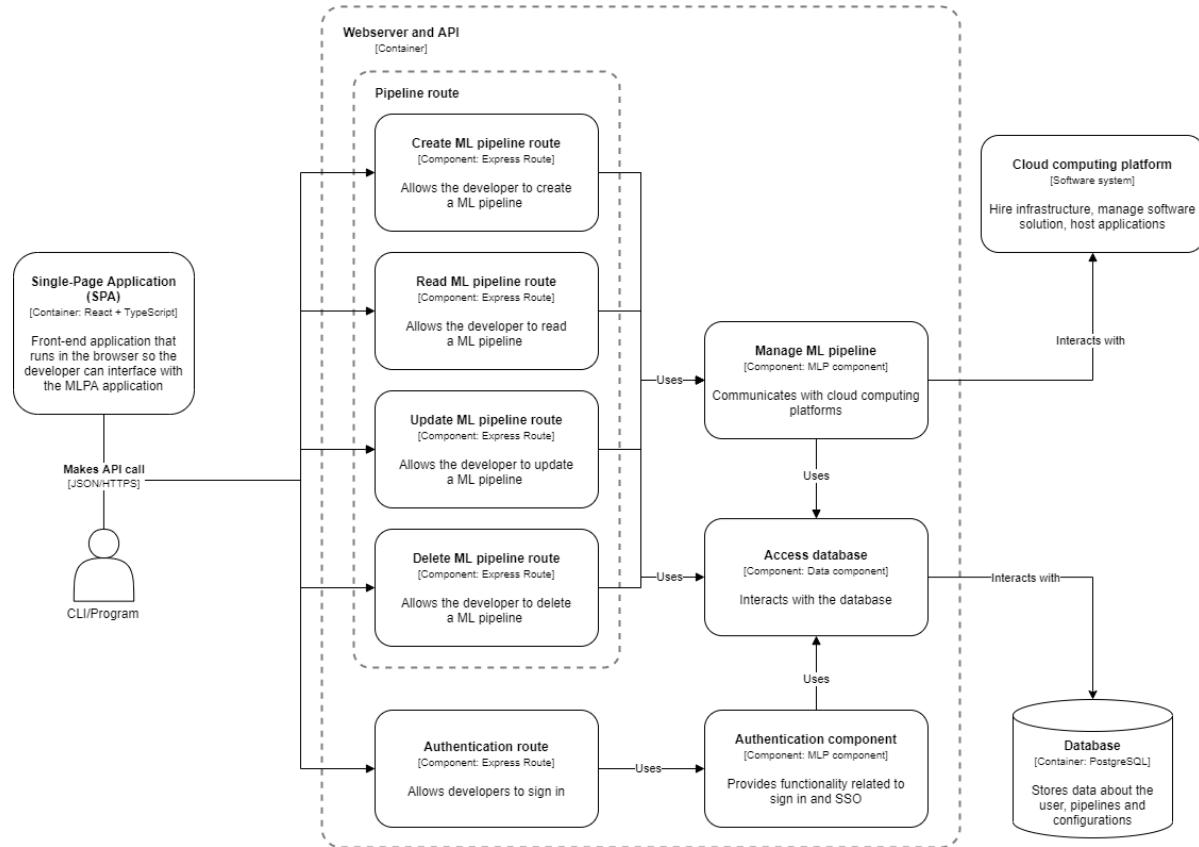
Figuur 6.3: Container niveau diagram van het architecturaal ontwerp.

Als laatste is de component overview van de SPA te zien in Figuur 6.4 en van de webserver met API in Figuur 6.5. De SPA bevat een *AuthManager* component dat controleert of de developer is ingelogd. Als dat het geval is, stuurt de *Router* component de developer door naar de *Dashboard*, *Pipelineoverview* of *Pipelinedetails* pagina.



Figuur 6.4: Single page application (SPA) component niveau diagram van het architecturaal ontwerp.

De component diagram van de Node.js server bevat het complexe gedeelte (Figuur 6.5). Hier komt een API request binnen in een *ExpressRoute*. De route maakt vervolgens gebruik van Pulumi en het database om taken uit te voeren zoals het starten van een pipeline, het uploaden van een dataset of de status van een run bekijken.



Figuur 6.5: Node.js server component niveau diagram van het architecturaal ontwerp.

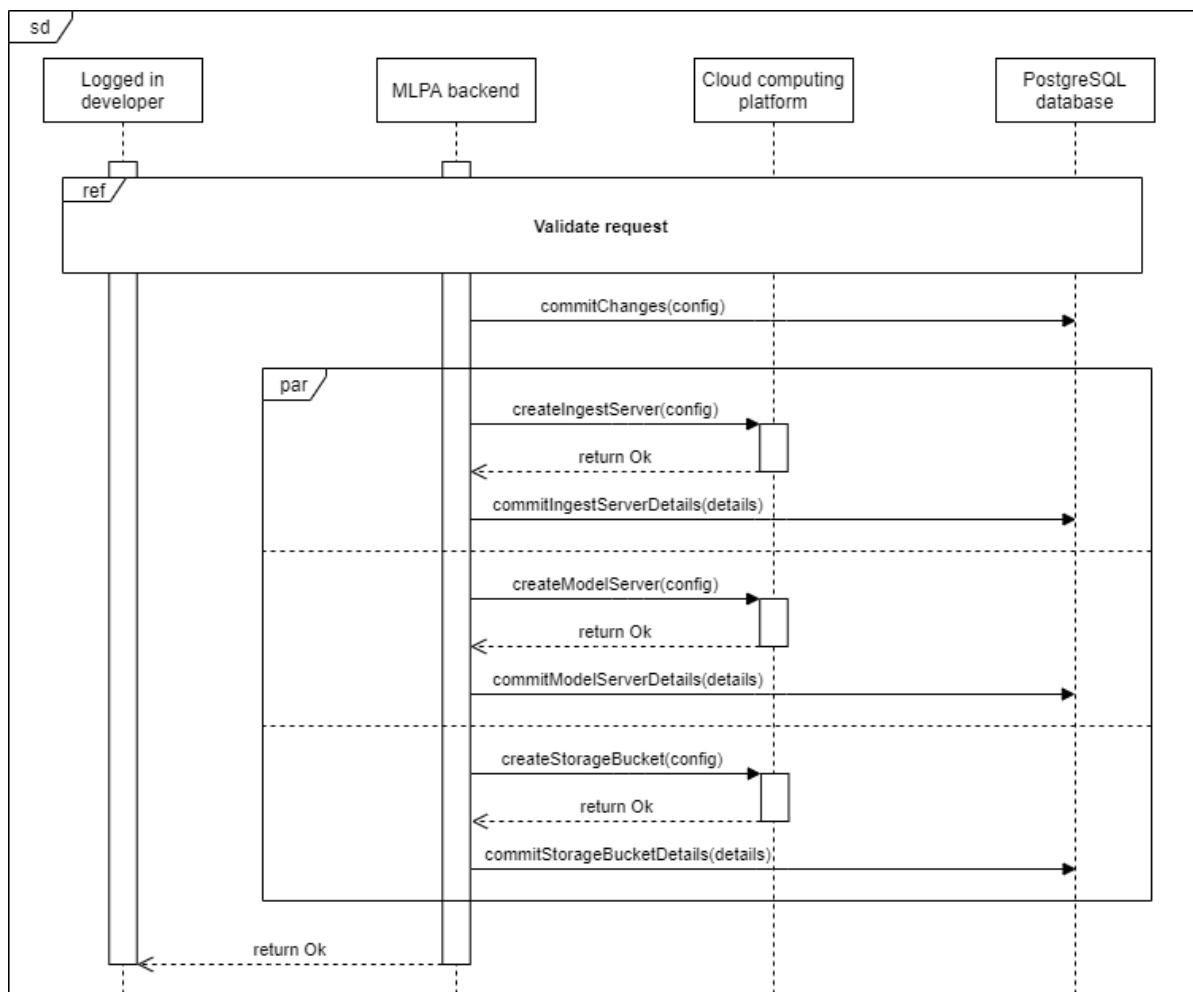
De architectuur van de Node.js server is ontworpen volgens de repository-service pattern [43]. Dit is een pattern dat bepaalde zaken van elkaar scheid zodat het project onderhoudbaar blijft. In paragraaf 7.5 zal dit uitgelegd worden met voorbeelden.

Diagrammen voor het laatste niveau, code, zijn achterwege gelaten door de vereiste tijd om de diagrammen te maken en het feit dat de diagrammen vaak tijdens het programmeerproces zullen veranderen. Brown geeft ook als advies om de code diagrammen niet te maken of automatisch te laten genereren [44].

6.3 Sequence diagrammen

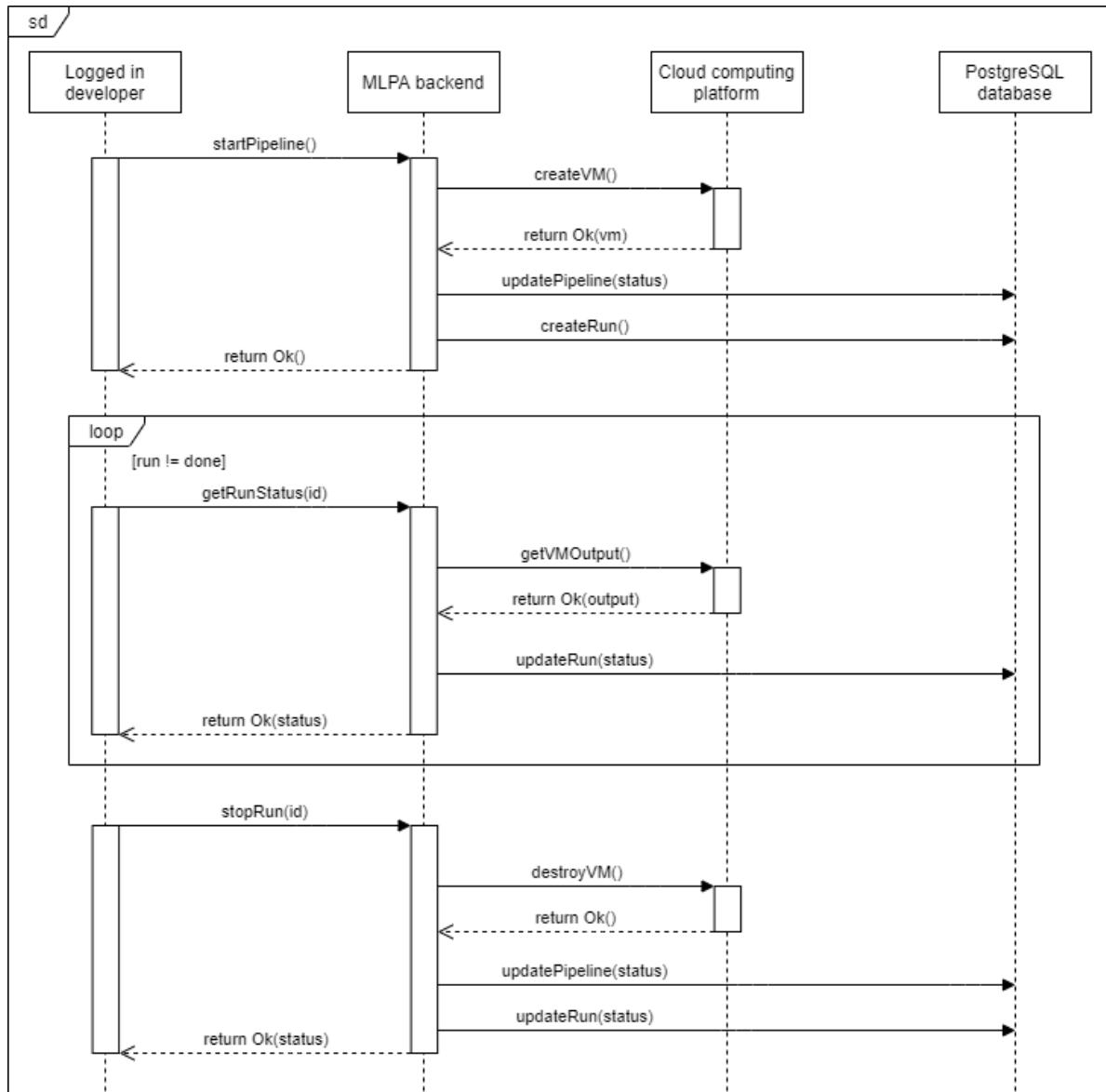
Naast de C4 model diagrammen is het behulpzaam om uit te leggen wat er gebeurd als er een actie wordt uitgevoerd. Dit kan met behulp van sequence diagrammen. Voor belangrijke acties zoals het aanmaken of uitvoeren van een pipeline is een sequence diagram gemaakt. Daarnaast is een entity relation diagram (ERD) gemaakt om de structuur van het database weer te geven. In deze kop wordt door een aantal diagrammen gelopen.

In Figuur 6.6 is te zien wat er gebeurt als een pipeline wordt aangemaakt. De developer stuurt een request naar de backend waar het wordt gevalideerd. De validatie sequence diagram is te vinden in bijlage XI. De backend slaat vervolgens de gegevens op waarna, in parallel, een ingest server, model server en storage bucket worden aangemaakt. De ingest server zorgt voor de intake van datasets, de model server traant het model en in de storage bucket worden alle datasets en modellen opgeslagen. Als deze drie resources zijn aangemaakt, wordt de request voltooid.



Figuur 6.6: Sequence diagram van het aanmaken van een pipeline.

Een sequence diagram is gemaakt om te laten zien wat er gebeurt als een pipeline wordt gestart (Figuur 6.7). De developer zal een request maken om de pipeline te starten. Om dit te doen stuurt de backend een request naar de cloud platform om een virtual machine (VM) te starten. Na het opstarten wordt het trainen van het model automatisch gestart. De backend past de status van de pipeline aan en voltooid de request. De developer weet nu dat een VM is gestart en kan vragen voor de output van de VM. In de output is te zien waar de VM mee bezig is. Dit gebeurt continu totdat het model getraind is en de artefacten veilig zijn opgeslagen. Dit zijn afbeeldingen, modellen en datasets die bewaard moeten worden. Hierna zal de developer een laatste request sturen om de VM te verwijderen en de status in het database aan te passen.



Figuur 6.7: Sequence diagram van het starten van een pipeline.

7 Proof of concept

Om de PoC daadwerkelijk te maken moet er eerst gedefinieerd worden wat de scope is en hoe de PoC eruit gaat zien. De scope brengt in beeld wat er gedaan moet worden en wat de minimal viable product (MVP) is. Op de MVP kan de mock up gebaseerd worden. In dit hoofdstuk wordt de scope bepaald, delen van de mock up doorgelopen, uitleg gegeven over de PoC en kwaliteit van de code. Het hoofdstuk wordt afgesloten met een conclusie en advies

7.1 User requirements verzamelen

Om te begrijpen wat er gebouwd moet worden kunnen de behoeftes opgeschreven worden in de vorm van user stories. Een user story is de behoefte opgeschreven in een specifieke formaat om belangrijke informatie te achterhalen [45]. Een bekende formaat is de *Connextra template* van Mike Cohn [45]: **”As a [role], I can [capability], so that [receive benefit]”**. De template zorgt ervoor dat de rol (role), functie (capability) en reden (benefit) bekend is. In Figuur 7.1 zijn alle user stories opgesteld, gegroepeerd in Epics. Bij elke user story is ook aangegeven of het MVP is, welk prioriteit het heeft en de t-shirt size. De t-shirt size is een manier om op een hoog niveau aan te geven hoeveel werk een user story vereist. Een **S** vereist weinig werk terwijl **XL** veel werk vereist.

ID	Epic	As a/an...	i want to...	so that...	MVP [Y/N]	P [L/M/H]	T-Shirt [S/M/L/XL]	Notes
1	Manage pipeline	Developer	create a pipeline with a name, project and platform	I can create a pipeline configuration and train ML models	Y	H	M	
2	Manage pipeline	Developer	see the status of a pipeline	I know if a pipeline works or not	Y	L	S	
3	Manage pipeline	Developer	switch between platforms	I am not vendor-locked in	N	H	L	
4	Ingest server	Developer	see the status of the ingest server	I know if the ingest server is available and works correctly	N		S	
5	Ingest server	Developer	upload data from an application to the ingest server with specification on how it's saved	I can use the data in a dataset when a new ML model is being trained	N		M	
6	Model deployment server	Developer	see the status of the model deployment server	I know if the model deployment server is available and works correctly	N		S	
7	Model deployment server	Developer	get a prediction from the model deployment server via an endpoint	I can use a ML model without downloading it and using it locally	N		M	
8	Datasets/artifacts storage bucket	Developer	upload datasets	I can use the dataset when training ML models	Y	H	M	
9	Datasets/artifacts storage bucket	Developer	download datasets	I can experiment locally	Y	L	S	
10	Datasets/artifacts storage bucket	Developer	download dataset artifacts	I can experiment and debug locally	Y	L	S	
11	Datasets/artifacts storage bucket	Developer	download model artifacts	I can use a ML model offline	Y	M	S	
12	Pipeline configuration	Developer	be warned with detailed explanation if the configuration is invalid	I can correct the errors	N		M	
13	Pipeline runs	Developer	see the steps that have been performed in a run	I know what the MLPA did	N	H	XL	
14	Pipeline runs	Developer	see the output a step might have generated	I can see the analysis of datasets and models in a run	N	H	L	
15	Pipeline runs	Developer	start a pipeline run	I can train a model	Y	H	L	
16	Platform management	Developer	save platform keys or pats	I can access resources in platforms when using pipelines	N		S	

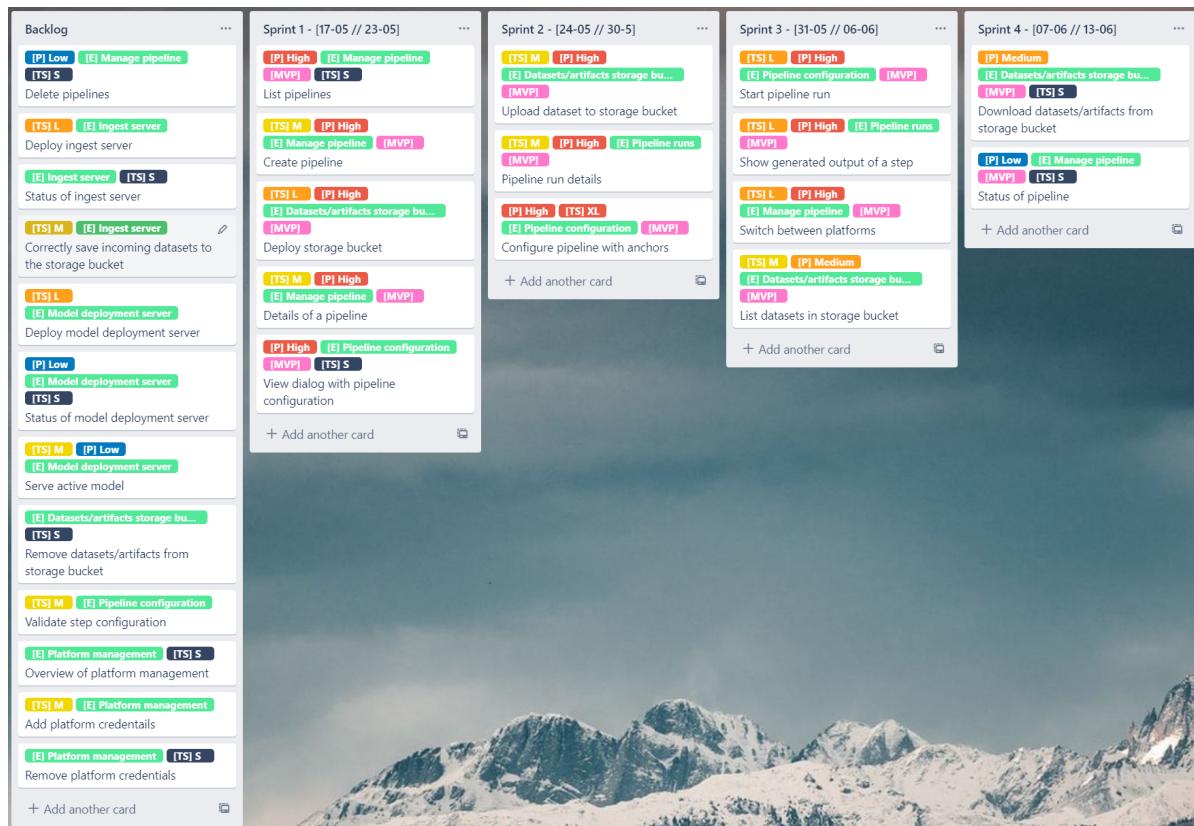
Figuur 7.1: Requirements opgesteld voor de proof of concept.

De user stories zijn opgesteld in samenwerking met de begeleiders van NGTI over het hele afstudeerproces. Nu de requirements zijn opgesteld, kan een Kanban bord opgericht worden om sprints te maken.

7.1.1 Requirements vertalen naar een scrum bord

Een Kanban bord is een werkwijze om visueel aan te geven in welke stadia een taak is in het proces [46]. Taken in een Kanban bord vallen altijd onder een kolom om de status aan te geven. Een kolom kan bijvoorbeeld "todo", "doing", of "done" zijn. Een scrum bord is een vorm van een Kanban bord waarbij taken in een sprint zijn ingepland [47]. Een sprint is een vaste periode van bijvoorbeeld een of twee weken waarin gewerkt wordt aan de ingeplande taken. Dit verschilt met een Kanban bord waarbij taken op elk moment toegevoegd kunnen worden. In software development kan dit niet; sprints zorgen ervoor dat er geen scope creep plaats vindt.

De user stories uit Figuur 7.1 zijn vertaald naar tickets in het scrum bord. Een user story is opgesplitst in meerdere tickets. Dit is het geval met bijvoorbeeld user story "upload datasets" (ID 8). Om deze functionaliteit te bouwen moet een storage bucket beschikbaar zijn en in de frontend de mogelijkheid zijn om dit te doen. In Figuur 7.2 zijn de tickets aangemaakt met relevante labels om aan te geven bij welke epic een ticket hoort, of het MVP is, de prioriteit en t-shirt size.



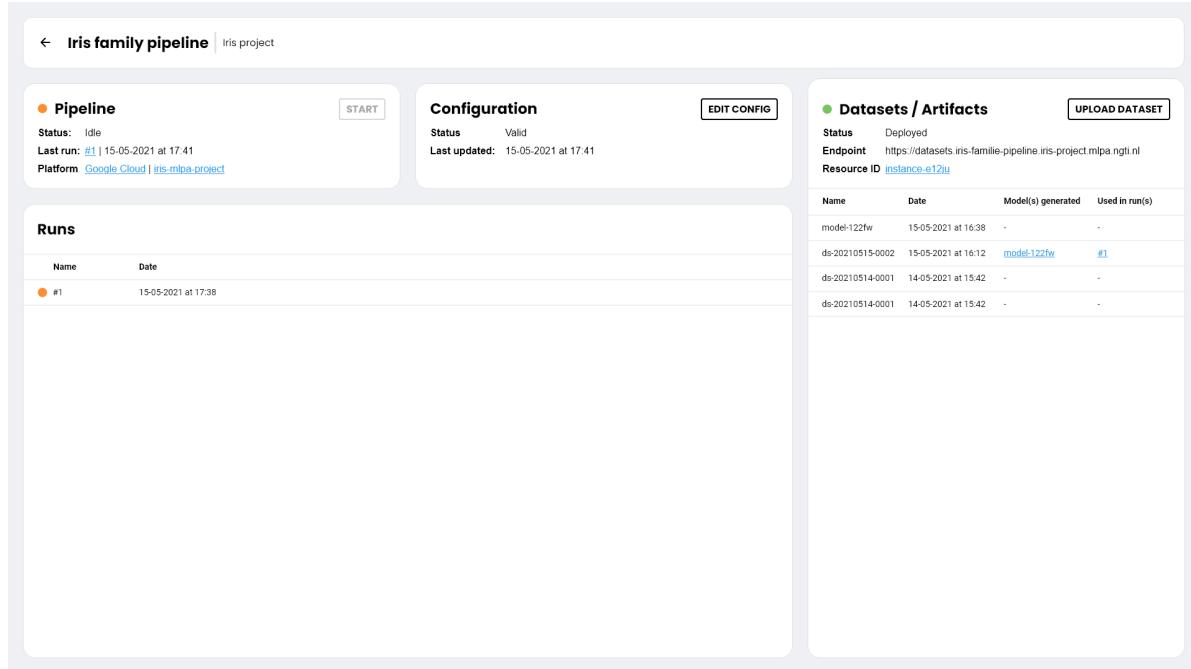
Figuur 7.2: Trello bord in het begin van sprint 1

In het ontwikkelproces is na elke sprint het scrum bord opnieuw bekijken om te reflecteren op de afgelopen sprint en de komende sprints opnieuw in te plannen als dit nodig was. Nu de tickets aangemaakt zijn en de MVP is vastgesteld, kan de mock up gemaakt worden.

7.2 Mock up van de proof of concept

De mock up is bedoeld om te laten zien hoe de frontend van de PoC eruit gaat zien. Dit geeft ook de kans om te controleren of de user interface (UI) logisch in elkaar zit. Om de mock up te ontwerpen is gebruik gemaakt van Adobe XD [48]. De mock up bevat alleen functionaliteiten die MVP zijn.

Een pipeline wordt aangemaakt door het invullen van de naam en project (bijlage IX). Na het aanmaken van een pipeline kan erop geklikt worden. In Figuur 7.3 is de mock up van de details van een pipeline te zien. Op deze pagina is de status van de pipeline, configuration, dataset/artifacts en runs te zien.



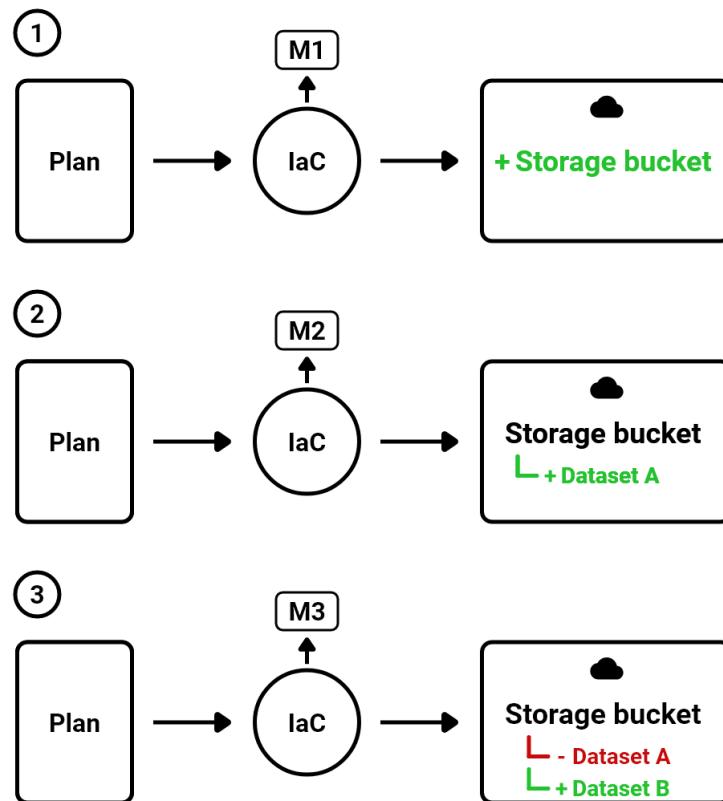
Figuur 7.3: Mock up van de pipeline details pagina.

De configuration is een Python script dat door de developer is gedefinieerd. De script wordt uitgevoerd als de pipeline wordt gestart en bevat de stappen uit Figuur 4.2. Stappen zoals "Model deployment" en "Model feedback" vallen uit de scope. In de datasets/artifacts sectie kan de developer datasets en artefacts up- en downloaden. Als laatste kan de developer de status van de pipeline en run zien. Een run is een individuele run als een pipeline wordt gestart. Mock ups van andere gedeeltes van de PoC is te vinden in bijlage X. Nu het voorwerk is gedaan, kan code voor de PoC geschreven worden.

7.3 Wijzigingen in het architecturaal ontwerp

Tijdens het ontwikkelproces is naar voren gekomen dat het orkestreren van infrastructuur in een cloud platform met een tool niet de juiste werkwijze is. Een orkestratietool werkt zoals uitgelegd in paragraaf 5.1 met een plan. De tool controleert de infrastructuur in een cloud platform aan de hand van het plan en past zo nodig de infrastructuur aan.

Gedurende het experiment kwamen geen problemen naar boven; echter bij het implementeren van een feature vertoonde Pulumi gedrag dat onverwacht was. In Figuur 7.4 is een illustratie te zien ter ondersteuning van de uitleg. In de eerste stap (1) wordt middels code een plan beschreven waarbij een storage bucket aangemaakt moet worden. Pulumi valideert het plan en maakt vervolgens de storage bucket aan. Pulumi houdt intern bij met een manifest (M1) wat is aangemaakt, in dit geval een storage bucket. Bij de tweede stap (2) wordt Dataset A geüpload in de storage bucket. Pulumi houdt weer bij wat er aangemaakt is in de storage bucket met het manifest (M2). Bij de derde stap staat in het plan dat alleen Dataset B geüpload moet worden. Pulumi vergelijkt het plan met de manifest (M2) en ziet dat Dataset A niet in het plan staat. Pulumi verwijderd vervolgens Dataset A en upload Dataset B.



Figuur 7.4: Gedrag van Pulumi met een incorrect plan.

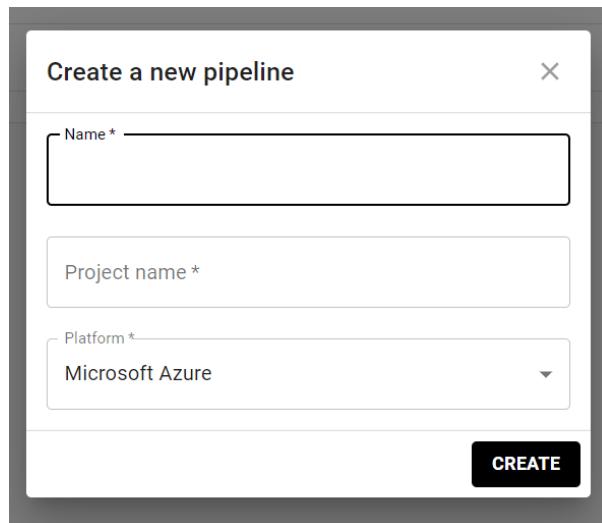
Pulumi verwacht dat de volledige infrastructuur in het plan beschrijft. Het is niet mogelijk om incrementeel de infrastructuur aan te passen. Voor de PoC is gekozen om verder te werken met libraries van de cloud platformen AWS, Google Cloud en Azure zelf. Na het onderzoeken of het mogelijk is om infrastructuur in te richten en naar de kwaliteit van de documentatie is gekozen om te werken met de libraries van Google Cloud en Azure. De documentatie van AWS is onoverzichtelijker en Amazon gebruikt unieke termen voor hun resources en services wat de leercurve hoger legt dan de andere twee libraries [49].

7.4 De proof of concept

In paragraaf 6.2 is kort uitgelegd dat de backend gebouwd wordt met Node.js en Express en de frontend met React.js en TypeScript. Voor de PoC is een GitHub repository aangemaakt waarin de front- en backend geïnitialiseerd is met behulp van de documentatie van de frameworks. In de volgende koppen wordt in een logische volgorde door de applicatie gelopen waarbij uitleg over de werking wordt gegeven met screenshots van de UI en code als ondersteuning.

7.4.1 Aanmaken van pipelines

Om te beginnen moet een pipeline aangemaakt kunnen worden. In de frontend gaat dit met een dialoog (Figuur 7.5). Hierin kan de naam, project en cloud platform aangegeven worden.



Figuur 7.5: Aanmaak dialoog van een pipeline.

Zodra de developer klikt op de **Create** knop, wordt een request verstuurt van de frontend naar de backend. De backend stuurt de request naar een service dat ervoor zorgt dat de juiste acties uitgevoerd wordt. De service in Figuur 7.6 controleert welk platform er gespecificeerd is en spreekt de relevante functie aan. Als het platform bijvoorbeeld Google Cloud is, wordt de functie *gc.createBucket()* op lijn 6 aangeroepen.

```

1 export const createPipeline = async (data: DTO_CreatePipeline) => {
2   try {
3
4     const pipeline = await createPipelineInDb(data)
5     if (data.platform === 'GOOGLE') {
6       await gc_createBucket({ name: data.name, project_id: data.project_id })
7     } else if (data.platform === 'AZURE') {
8       await azure_createResourceGroup({ name: data.name, project: data.project })
9       await azure_createBucket({ name: data.name, project: data.project })
10    }
11
12    return pipeline
13  } catch (e) {
14    throw new Error(e)
15  }
16}

```

Figuur 7.6: Service om een pipeline aan te maken.

Om een pipeline in Google Cloud aan te maken hoeft alleen een storage bucket aangemaakt te worden. De functie die deze actie uitvoert is te zien in Figuur 7.7. De bucket heeft een naam nodig en de locatie. Op lijn 5 van Figuur 7.7 is ”**EUROPE-WEST4**” gespecificeerd. Dit is een locatie in Nederland.

```

1 export const gc_createBucket = async (data: { name: string, project_id: string }) => {
2   const storage = new Storage()
3
4   return await storage.createBucket(createBucketName(data.name), {
5     location: 'EUROPE-WEST4'
6   })
7 }

```

Figuur 7.7: Functie om een storage bucket aan te maken in Google Cloud.

Naast het aanmaken van de storage bucket in Google Cloud wordt de details van de pipeline opgeslagen in de PostgreSQL database (Figuur 7.6, lijn 4). Na het aanmaken van een pipeline kan een dataset geüpload worden.

7.4.2 Datasets uploaden

Tijdens het aanmaken van de pipeline is een opslaglocatie in het cloud platform aangemaakt. In de details weergave van een pipeline is een sectie weergegeven zoals in Figuur 7.8 waar de developer datasets kan uploaden. Na het uploading krijgt de developer een confirmatie te zien en wordt de lijst met bestaande datasets en artefacts ververst.

Name	Size
iris.csv	4.4 KB

Figuur 7.8: Status en upload-mogelijkheid van datasets en artefacten in de details pagina van een pipeline.

De code in de backend voor het uploaden is vergelijkbaar met de code in Figuur 7.6 en Figuur 7.7. In Figuur 7.9 is de service te zien dat wordt gebruikt als de developer een dataset upload in de frontend. De service selecteert de juiste functie om de dataset te uploaden om basis van het gekozen platform.

```

1 export const uploadData = async (id: number, files: fileType[]) => {
2   try {
3     const p = await pipeline(id)
4
5     if (p.platform === 'GOOGLE') {
6       await gc_uploadToBucket({ name: p.name, project_id: p.project_id }, files)
7     } else if (p.platform === 'AZURE') {
8       await azure_uploadToBucket({ name: p.name, project: p.project }, files)
9     }
10
11    return {}
12  } catch (e) {
13    throw new Error(e)
14  }
15 }
```

Figuur 7.9: Service om datasets te uploaden naar een opslaglocatie binnen een cloud platform.

```

1 export const gc_uploadToBucket = async (data: { name: string, project_id: string }, files: fileType[]) => {
2   const gc_bucket = gc_storage.bucket(createBucketName(data.name))
3
4   console.log(`Uploading to bucket: ${gc_bucket.name}`)
5
6   files.map(file => {
7     gc_bucket.upload(file.path, { destination: file.name, public: true })
8       .then(r => console.log('done uploading'))
9       .catch(e => console.log(e))
10  })
11
12  return {}
13 }

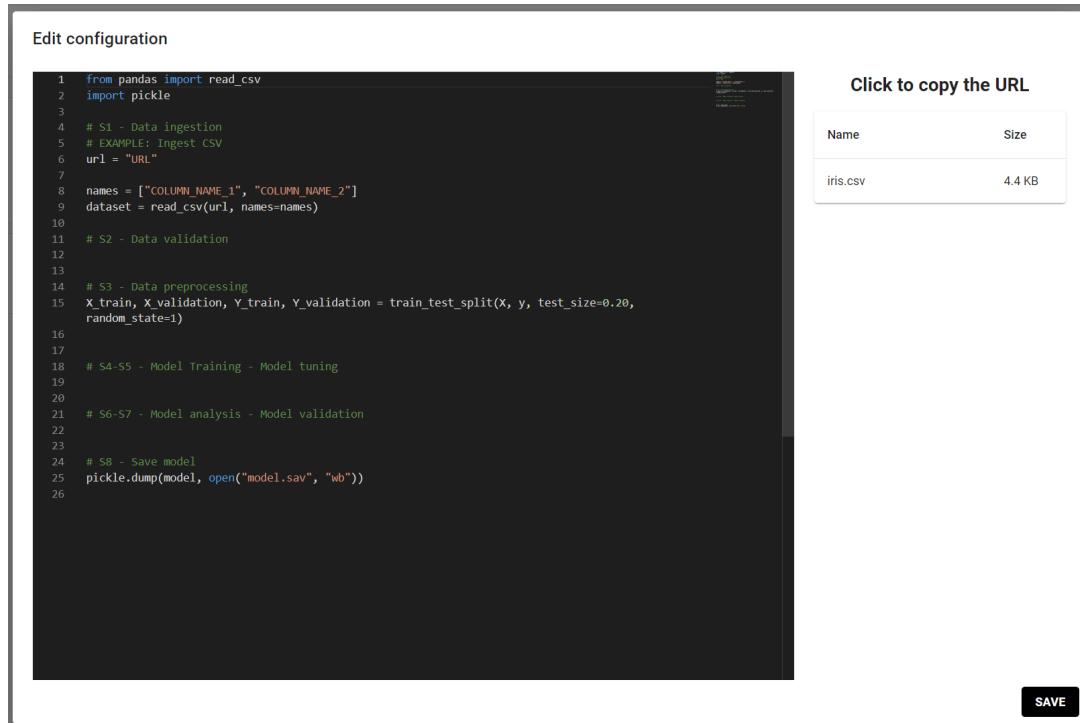
```

Figuur 7.10: Functie om datasets te uploaden naar een storage bucket in Google Cloud.

De backend maakt een instantie van de bucket in Figuur 7.10 op lijn 2 en voert de *upload()* functie uit op lijn 7. De geüploade datasets zijn vervolgens beschikbaar om te gebruiken in de configuratie van de pipeline.

7.4.3 Configuratie definiëren

Het laatste wat de developer moet doen om de pipeline te starten is het opgeven van de configuratie. Zoals aangegeven in paragraaf 7.2 is de configuratie een Python bestand dat uitgevoerd wordt als de pipeline gestart wordt. In de PoC is de configuratie bewerkbaar door middel van een dialoog (Figuur 7.11) met rechts een lijst van de datasets. De standaard configuratie dient als startpunt voor de developer. Een link naar de dataset kan gekopieerd worden door op een dataset te klikken.



Figuur 7.11: Configuratie dialoog van een pipeline.

Zodra de developer op de knop **Save** klikt, wordt een request gedaan naar de backend om de configuratie op te slaan. De code om de configuratie op te slaan is te zien in Figuur 7.12. Hier wordt een nieuwe configuratie aangemaakt en gelinkt aan de pipeline.

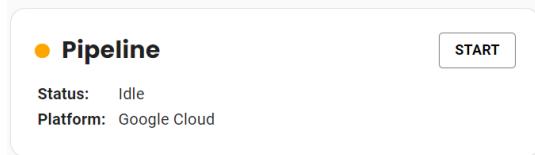
```
1 export const createPipelineConfiguration = async (pipeline_id: number, config: string) => {
2   return await prisma.pipelineConfiguration.create({
3     data: {
4       config: config,
5       pipeline: {
6         connect: {
7           id: pipeline_id
8         }
9       }
10    }
11  })
12}
13
```

Figuur 7.12: Code om de configuratie op te slaan in de database.

Het is mogelijk om code te schrijven in andere talen dan Python om modellen te trainen. Om de complexiteit en scope klein en realistisch te houden is er gekozen voor Python. De taal is een van de populairste talen [27]. Daarnaast heeft Python een complete set van libraries om data te transformeren, plotten in diagrammen en modellen te trainen [50]. Nu de configuratie is opgeslagen zijn alle benodigdheden beschikbaar om de pipeline te starten.

7.4.4 Pipeline starten

Zodra de opslaglocatie en configuratie beschikbaar zijn, krijgt de developer de mogelijkheid om de pipeline te starten Figuur 7.13. Zoals uitgelegd in Figuur 6.7 worden drie stappen uitgevoerd: het aanmaken van een VM, het continu ophalen van de output en vervolgens het verwijderen van de VM.



Figuur 7.13: Status en startknop van een pipeline.

De request van de frontend komt in de backend binnen om een pipeline te starten en komt terecht bij de functie *gc_startPipeline()* in Figuur 7.14. Deze functie maakt een configuratie aan op lijn 7 waarin de besturingssysteem, soort VM in Google Cloud en de start-up script staat gedefinieerd. De configuratie wordt gebruikt als de VM wordt aangemaakt op lijn 27. De status van de pipeline wordt aangepast naar **"RUNNING"** en een nieuwe Run wordt aangemaakt in het database.

```

1 export const gc_startPipeline = async (pipeline_id: number, name: string, run: number) => {
2   const compute = new Compute()
3   const zone = compute.zone('europe-west4-a')
4   const pipeline = await fetchPipeline(pipeline_id)
5   const configuration = await currentPipelineConfiguration(pipeline_id)
6
7   const config = {
8     os: 'ubuntu',
9     http: true,
10    machineType: 'e2-small',
11    metadata: {
12      items: [
13        {
14          key: 'startup-script',
15          value: `...`,
16        }
17      ]
18    }
19  }
20
21  const vm = zone.vm(`${name}-${run}`)
22
23  await updatePipeline(pipeline.id, { ...pipeline, status: 'RUNNING', run: pipeline.run + 1 })
24  const _run = await newRun(pipeline_id, pipeline.platform, `${name}-${run}`, 'PROVISIONING')
25
26  console.log(`Creating VM ${name}-${run} ...`);
27  await vm.create(config)
28
29  return { run_id: _run.id }
30 }

```

Figuur 7.14: Code om een virtual machine (VM) in Google Cloud te starten.

In de configuratie is een start-up script gedefinieerd (Figuur 7.14, lijn 15). De script wordt uitgevoerd zodra de VM beschikbaar is en zorgt ervoor dat het model getraind kan worden en artifacts opgeslagen worden in de opslaglocatie indien dit nodig is. Op lijn 18 in Figuur 7.15 wordt de configuratie die de developer heeft opgeslagen in deelparagraaf 7.4.3 in het script gezet. Na het trainen van het model worden alle artifacts in de *output* map geüpload naar de opslaglocatie. Dit gebeurt met bash commando's op lijn 34 en 35.

```

1 #! /bin/bash
2 apt-get update
3 apt-get install -y python3-pip
4 echo '[MLPA] - INSTALLING PIP PACKAGES'
5
6 sudo pip3 install pandas
7 sudo pip3 install numpy
8 sudo pip3 install matplotlib
9 sudo pip3 install scikit-learn
10
11 sudo mkdir ml
12
13 cd ml
14 sudo mkdir output
15 echo '[MLPA] - CREATED OUTPUT FOLDER'
16
17 sudo cat <<'EOF' >> main.py
18 ${configuration.config}
19 EOF
20 echo '[MLPA] - CREATED PYTHON FILE'
21
22 sudo cat <<'EOF' >> key.json
23 ${JSON.stringify(key, null, 2)}
24 EOF
25 echo '[MLPA] - CREATED KEY FILE'
26
27 sudo python3 main.py
28
29 gcloud auth activate-service-account --key-file key.json
30
31 cd output
32
33 echo '[MLPA] - UPLOADING TO STORAGE BUCKET ... '
34 for FILE in *; do sudo mv $FILE "$HOSTNAME-$FILE"; done
35 for FILE in *; do gsutil cp $FILE gs://test-pipeline-artifacts; done
36
37 echo '[MLPA] - DONE'

```

Figuur 7.15: Start-up script van een virtual machine (VM) in Google Cloud.

Gedurende het uitvoeren van de script in Figuur 7.14 vraagt de frontend continu naar de output van de VM. Deze request wordt afgehandeld door de *gc_getRunStatus()* functie in bijlage XII. De functie haalt de output op (lijn 9) en slaat het op in de database voordat het terug gestuurd wordt naar de frontend. De frontend laat de output zien (bijlage XIII) en controleert of de output '[MLPA] - DONE' bevat. Als de output deze tekst bevat, weet het systeem dat het model is getraind en alle artifacts in de output map geüpload zijn in de opslaglocatie. Als dit niet het geval is, wordt de output opnieuw opgehaald en geüpdatet in de frontend. Dezelfde controle op de output vindt vervolgens weer plaats. Als de output wel '[MLPA] - DONE' bevat, wordt er middels een request aan de backend de VM verwijderd (bijlage XIV). In het verwijderproces wordt de status in de database geüpdatet.

7.4.5 Model downloaden

Nadat het model getraind is, worden artifacts in de *output* map geüpload naar de opslaglocatie. Het is aan de developer om in het Python script te definiëren dat artifacts opgeslagen worden in de *output* map. De lijst met datasets/artifacts wordt geüpdatet zodra de start-up script klaar

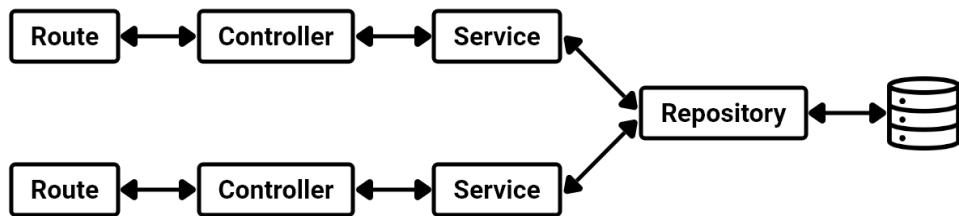
is. De artifacts zijn downloadbaar door erop te klikken.

● Datasets / Artifacts		UPLOAD DATASET
Name	Size	
test-pipeline-27-scatter_matrix.png	88 KB	
test-pipeline-27-model.sav	3.4 KB	
test-pipeline-27-histogram.png	19 KB	
test-pipeline-27-boxplot.png	18 KB	
iris.csv	4.4 KB	

Figuur 7.16: Model en overige artifacts van het trainproces geüpload in een storage bucket.

7.5 Kwaliteit van de code

Gedurende het ontwikkelproces is gebruik gemaakt van de repository-service pattern, verschillende principes en "good practices". In paragraaf 6.2 is de repository-service pattern kort uitgelegd. Het idee achter deze pattern is dat het ophalen van data uit de database en complexe logica gescheiden van elkaar blijft [43]. De repository laag in Figuur 7.17 bevat create, read, update en delete (CRUD) functies die in verschillende services hergebruikt kan worden.



Figuur 7.17: Voorbeeld van de repository-service pattern.

Doordat de architectuur ontworpen is met de repository-service pattern, voldoet de PoC ook aan de don't repeat yourself (DRY) [51]. Volgens deze principe moet er geen code geschreven worden dat al geschreven is. Functies zoals het ophalen van een pipeline kan hergebruikt worden met behulp van de repository laag (Figuur 7.18).

```

1 const getPipeline: RouteFunction = async (req, res, next) => {
2   const id = Number(req.params.id)
3
4   try {
5     const pipeline = await fetchPipeline(id)
6
7     if (pipeline) {
8       res.json(pipeline)
9     } else {
10       res.status(404).end()
11     }
12   } catch (e) {
13     return res.status(500).json(e)
14   }
15 }
16
17 const startPipeline: RouteFunction = async (req, res) => {
18   const id = Number(req.params.id)
19
20   try {
21     const p = await fetchPipeline(id)
22
23     const run_id = await runPipeline(p.id)
24
25     return res.status(200).json(run_id)
26   } catch (e) {
27     return res.status(500).json(e)
28   }
29 }

```

Figuur 7.18: Voorbeeld van de don't repeat yourself (DRY) principe in de backend.

De toepassing van de DRY principe komt ook voor in de code van de frontend. Zoals weergegeven in Figuur 7.19 wordt bijvoorbeeld de container waar elke sectie in zit hergebruikt. Met deze werkwijze kunnen elementen in de frontend sneller en consistenter gebouwd worden.

The screenshot shows a web-based management interface for a pipeline named "test-pipeline". The top navigation bar includes a back arrow, the pipeline name, and a "test-project" label. Below the header, there are three main sections: "Pipeline", "Configuration", and "Datasets / Artifacts".

- Pipeline:** Status: Idle, Platform: Google Cloud. Includes a "START" button.
- Configuration:** Status: Valid, Last updated: 11/06/2021, 15:55:00. Includes an "EDIT CONFIG" button.
- Datasets / Artifacts:** Status: Deployed, Endpoint: gs://test-pipeline-artifacts. Includes an "UPLOAD DATASET" button. A table lists artifacts:

Name	Size
test-pipeline-27-scatter_matrix.png	88 KB
test-pipeline-27-model.sav	3.4 KB
test-pipeline-27-histogram.png	19 KB
test-pipeline-27-boxplot.png	18 KB
iris.csv	4.4 KB

Figuur 7.19: Voorbeeld van de don't repeat yourself (DRY) principe in de frontend.

7.6 Conclusie

In dit hoofdstuk is onderzoek gedaan naar het antwoord op de hoofdvraag: **H: In welke mate kan een machine learning pipeline worden geautomatiseerd onafhankelijk van de onderliggende cloud computing platform?** Het onderzoek is uitgevoerd met het verzamelen van requirements, ontwerpen van een mock up en een PoC bouwen.

Middels het implementeren is duidelijk geworden dat een orkestratietool niet de correcte werkwijze is om infrastructuur aan te passen. De tool verwacht dat alle resources, ook resources die voorheen aangemaakt zijn, in het plan zijn gedefinieerd. Resources worden verwijderd als dit het geval is. Om dit op te lossen is gekozen om te werken met libraries van de cloud platformen zelf. Documentatie van de libraries is aanwezig, echter laat documentatie van een aantal libraries te wensen over. De functionaliteit om resources aan te maken is wel aanwezig en is dus een valide vervanging.

De scope van de PoC bevat de stappen uit Figuur 4.2 om een model te trainen. Het trainen gaat met een Python bestand dat als configuratie wordt opgeslagen. Eerst moet een pipeline aangemaakt worden zodat de PoC de benodigheden klaar kan zetten voor de developer. Dat is de opslaglocatie in de gekozen cloud platform en informatie over de pipeline in de database. Vervolgens kan de developer een dataset uploaden, de configuratie opslaan en de pipeline starten. Na het starten maakt de PoC een VM aan in het cloud platform en voert een script uit zodat het model getraind en opgeslagen kan worden samen met overige artifacts in de opslaglocatie. Tot slot wordt de VM verwijderd en is het model beschikbaar om te gebruiken in een webserver of app om voorspellingen of classificaties te maken.

7.7 Advies

De PoC laat zien dat het automatiseren van een pipeline mogelijk is. Een developer van NGTI heeft alleen kennis nodig van ML om gebruik te maken van de PoC. Een grote verbetering is de manier hoe een developer configuratie opgeeft om het model te trainen.

7.7.1 Configuratie definiëren met stappen

De PoC vraagt aan de developer om Python code op te geven. Daarnaast wordt de mogelijkheid aangeboden om een link te kopiëren van een dataset om te gebruiken in een van de stappen. Dit kan aanzienlijk verbeterd worden door stappen te maken zoals in Figuur 4.2. In Figuur 7.20 is een mock up gemaakt van de configuratie dialoog. Links zijn de stappen weergegeven. Bij elke stap kan de developer een optie kiezen of selecteren wat er gedaan moet worden. De PoC converteert vervolgens de gekozen opties in code waarmee een model getraind kan worden.

Pipeline configuration

1 - Choose algorithm kind

- 2 - Dataset structure
- 3 - Pre-process tasks
- 4 - Algorithm exploration
- 5 - Model validation

What type of ML problem is it?

- Classification
Pick one of N labels
- Regression
Predict numeric values
- Clustering
Group similar examples
- Association rule learning
Infer likely association patterns in data
- Structured output
Create complex output
- Ranking
Identify position on a scale or status

SAVE CHANGES

NEXT

Figuur 7.20: Mock up van een verbeterd configuratie dialoog.

In de éérste stap in Figuur 7.20 is te zien dat de type van het ML probleem wordt gevraagd. Op deze wijze hoeft de developer niet zelf opzoek naar een geschikt algoritme maar kiest welk type ML probleem het is. Op basis hiervan kan de PoC een aantal algoritmes dat het probleem oplost selecteren. Bij stap 5 in Figuur 7.20 kan gespecificeerd worden wanneer een model goed genoeg is om uit te rollen. Over het algemeen probeert de PoC relevante opties te geven zodat de developer niet zelf hoeft te programmeren maar simpelweg kan kiezen wat er moet gebeuren.

8 Discussie

Gedurende de scriptie is onderzoek gedaan naar ML pipeline, orkestratietools en hoe de architectuur er uitziet van een applicatie dat ML pipelines automatiseert. Daarnaast zijn een aantal experimenten uitgevoerd om de potentie van een ML pipeline en orkestratietool te valideren.

De potentie van een ML pipeline is groter dan dat het lijkt. In paragraaf 4.4 is uitgelegd hoe ML versimpeld kan worden voor de developer door extra stappen toe te voegen. Dit kan niet alleen gebruikt worden door NGTI maar ook andere bedrijven die geïnteresseerd zijn in het toepassen van ML. Een stap verder zou zijn dat een standaard wordt geïntroduceerd dat wordt toegepast door industrieleiders zoals Google, Microsoft en Azure. Momenteel hebben ze hun eigen variant wat betekent dat specifieke kennis voor één cloud platform nodig is. Verhuizen naar een ander cloud platform betekent essentieel dat de werking van een ander ML pipeline geleerd moet worden.

In paragraaf 7.7 wordt er gesproken over hoe het definiëren van de configuratie vereenvoudigd kan worden door het te verdelen instappen. Vervolgens wordt er gevraagd wat voor type ML probleem het is en wat, op een hoog niveau, de pipeline moet doen. Dit zou niet alleen de PoC verbeteren, maar kan gebruikt worden als beginpunt voor developers nieuw zijn in het ML domein.

9 Reflectie

Het afstudeerproces is ook een groot leerproces voor mij geweest. De materie behandeld was niet alleen nieuw maar ook zeer complex. Daarnaast heb ik een aantal fouten gemaakt waardoor ik in tijdsnood kwam.

In de eerste helft van de afstudeerstage lag de focus te veel op ML. Ik deed ontzettend veel onderzoek naar ML omdat ik weinig tot niks wist van het onderwerp. Gedurende het onderzoek was ik theoretisch bezig en programmeerde of experimenteerde ik niet; er was geen balans tussen theoretisch en praktisch werk. Het onderzoek was gelukkig niet voor niets. De stappen in een pipeline werden genomen omdat deze nodig zijn om een model te trainen. Door de kennis die ik heb opgedaan werd dat snel duidelijk en begreep ik wat er gebeurde in een pipeline.

Na een stap terug te hebben gedaan werd het duidelijk dat ML pipelines de focus was, en niet ML zelf. Halverwege de afstudeerstage heb ik samen met mijn bedrijfsbegeleiders gezeten om de focus en scope te bepalen. Hierdoor werd het snel duidelijk waarnaar ik onderzoek moest doen en wat ik moest bouwen. Dit heeft enorm geholpen met mijn productiviteit. Door de duidelijke grens van wat wel en niet binnen de scope viel deed ik geen onderzoek naar onderwerpen die niet relevant waren. Dit is achteraf gezien een grote maar niet duidelijke valkuil in het ML domein. Er zijn talloze algoritmes met elk hun eigen theorie en er zijn ontzettend veel manieren om dingen te doen. Terugkijkend heb ik stukken geschreven dat niet geschreven hoefde te worden.

Een andere valkuil was de manier waarop ik aan de scriptie werkte. Doordat ik vooral theoretisch bezig was had ik moeite met de structuur van hoofdstukken en de samenhang tussen koppen. Ook halverwege heb ik besloten met zowel de school- als bedrijfsbegeleiders dat ik praktischer bezig zou zijn en wat minder tijd aan de scriptie zou besteden. Het praktisch bezig zijn hield voor mij in dat er meer en sneller experimenten werden gedaan. In de scriptie zette ik dan steekwoorden/zinnen om de kern van een kop te definiëren. Via deze manier kon ik de structuur van een hoofdstuk goed bepalen en kon er weer theoretisch gewerkt worden.

Iets wat ik vanaf het begin had moeten doen was het maken van een Trello bord om de status van taken bij te houden. Nadat ik een Trello bord had ingericht voor de laatste vier weken om aan de PoC te werken, wist ik dat een Trello bord beter zou werken voor mij. Bij elke meeting heb ik genotuleerd, maar uiteindelijk heb ik niet veel met de notities gedaan. Met een Trello bord zou ik suggesties en andere taken die in een meeting voorbij schieten kunnen organiseren. Ook dient het als een herinnering zodat ik taken niet vergeet te doen. Tijdens de laatste vier weken ben ik ook begonnen met het gebruik van de Pomodorotechniek. Dit is een werkwijze wat helpt om de focus bij het schrijven van de scriptie te houden. Met deze techniek maak je een lijst met taken. Daarna start je een timer en na 25 minuten neem je een pauze van 5 minuten. Dit doe je vier keer waarna je een pauze kan nemen van 15 minuten. Ikzelf heb pomofocus.io gebruikt. Met deze website kan je schatten hoeveel tijd nodig is voor een taak en geef vervolgens de geschatte eindtijd.

Voor een volgende scriptie zou ik een Trello bord inrichten voor taken en gebruik maken van de Pomodorotechniek om de focus bij het werk te houden. Notuleren tijdens meetings heeft niet veel zin en suggesties/taken kunnen gelijk in het Trello bord. Daarnaast is de focus en scope belangrijk voor de scriptie; zonder dit val ik in de diepte en verspil ik tijd.

Bibliografie

-
- [1] URL: <https://www.ngti.nl/diensten/> (bezocht op 22-03-2021).
 - [2] URL: <https://www.ngti.nl/oplossingen/> (bezocht op 29-03-2021).
 - [3] NGTI B.V. „Organogram”.
 - [4] URL: <https://www.swisscom.ch/en/about/beteiligungen-swisscom-uebersicht.html> (bezocht op 29-03-2021).
 - [5] URL: <https://www.ngti.nl/cases/swiss-climate-challenge/> (bezocht op 29-03-2021).
 - [6] URL: <https://www.swissclimatechallenge.ch> (bezocht op 29-03-2021).
 - [7] URL: <https://www.ngti.nl/cases/my-swisscom-app/> (bezocht op 29-03-2021).
 - [8] URL: <https://www.scribbr.nl/scriptie-structuur/methodologie-in-je-scriptie/> (bezocht op 25-03-2021).
 - [9] URL: <https://www.scribbr.nl/onderzoeksmethoden/kwalitatief-vs-kwantitatief-onderzoek/> (bezocht op 25-03-2021).
 - [10] E. Alpaydin. *Introduction to Machine Learning*. 4de ed. ISBN: 9780262043793. (Bezocht op 2020).
 - [11] URL: <https://serokell.io/blog/ai-ml-dl-difference> (bezocht op 01-06-2021).
 - [12] URL: <https://machinelearningmastery.com/think-machine-learning/> (bezocht op 16-03-2021).
 - [13] URL: <https://wiki.pathmind.com/neural-network> (bezocht op 16-03-2021).
 - [14] H. Hapke en C. Nelson. *Building Machine Learning Pipelines*. 1ste ed. ISBN: 9781492053194. (Bezocht op 2020).
 - [15] URL: <https://dvc.org> (bezocht op 06-04-2021).
 - [16] URL: <https://www.pachyderm.com> (bezocht op 06-04-2021).
 - [17] URL: <https://www.kaggle.com/rstatman/data-cleaning-challenge-scale-and-normalize-data> (bezocht op 27-05-2021).
 - [18] URL: https://sebastianraschka.com/Articles/2014_about_feature_scaling.html (bezocht op 27-05-2021).
 - [19] URL: <https://www.kdnuggets.com/2020/04/data-transformation-standardization-normalization.html> (bezocht op 27-05-2021).
 - [20] URL: <https://machinelearningmastery.com/difference-between-algorithm-and-model-in-machine-learning> (bezocht op 28-05-2021).
 - [21] URL: <https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/> (bezocht op 12-03-2021).
 - [22] URL: <https://fortune.com/2020/02/11/a-i-fairness-eye-on-a-i/> (bezocht op 30-05-2021).
 - [23] M Wittig en A Wittig. *Amazon Web Services in Action*. Manning Press. 1ste ed. ISBN: 9781617292880. (Bezocht op 2015).
 - [24] URL: <https://learn.hashicorp.com/tutorials/terraform/azure-build?in=terraform/azure-get-started> (bezocht op 01-06-2021).
 - [25] URL: <https://insights.stackoverflow.com/survey/2020> (bezocht op 03-06-2021).
-

-
- [26] URL: <https://insights.stackoverflow.com/survey/2020#technology-other-frameworks-libraries-and-tools> (bezocht op 03-06-2021).
 - [27] URL: <https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-other-frameworks-libraries-and-tools-loved3> (bezocht op 03-06-2021).
 - [28] URL: <https://www.pulumi.com/> (bezocht op 03-06-2021).
 - [29] URL: <https://www.redhat.com/en/blog/system-administrators-guide-getting-started-ansible-fast> (bezocht op 03-06-2021).
 - [30] URL: https://docs.ansible.com/ansible/latest/scenario_guides/cloud_guides.html (bezocht op 03-06-2021).
 - [31] URL: <https://docs.ansible.com/> (bezocht op 03-06-2021).
 - [32] URL: <https://www.pulumi.com/docs/guides/automation-api/> (bezocht op 03-06-2021).
 - [33] URL: <https://www.pulumi.com/docs/intro/cloud-providers/> (bezocht op 03-06-2021).
 - [34] URL: <https://www.pulumi.com/docs/> (bezocht op 03-06-2021).
 - [35] URL: <https://www.terraform.io/docs/language/index.html> (bezocht op 03-06-2021).
 - [36] URL: <https://registry.terraform.io/browse/providers> (bezocht op 03-06-2021).
 - [37] URL: <https://www.terraform.io/docs/index.html> (bezocht op 03-06-2021).
 - [38] URL: <https://c4model.com/> (bezocht op 07-06-2021).
 - [39] URL: <https://insights.stackoverflow.com/survey/2020#technology-web-frameworks> (bezocht op 07-06-2021).
 - [40] URL: <https://reactjs.org/docs/getting-started.html> (bezocht op 07-06-2021).
 - [41] URL: <https://nodejs.org/en/docs/> (bezocht op 07-06-2021).
 - [42] URL: <https://expressjs.com/en/4x/api.html> (bezocht op 07-06-2021).
 - [43] URL: <https://exceptionnotfound.net/the-repository-service-pattern-with-dependency-injection-and-asp-net-core/> (bezocht op 12-06-2021).
 - [44] URL: <https://c4model.com/#FAQ> (bezocht op 07-06-2021).
 - [45] URL: <https://www.agilealliance.org/glossary/user-story-template/> (bezocht op 11-06-2021).
 - [46] URL: <https://www.atlassian.com/agile/kanban/boards> (bezocht op 11-06-2021).
 - [47] URL: <https://www.forecast.app/blog/kanban-board> (bezocht op 11-06-2021).
 - [48] URL: <https://www.adobe.com/nl/products/xd.html> (bezocht op 11-06-2021).
 - [49] URL: <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/index.html> (bezocht op 12-06-2021).
 - [50] URL: <https://www.geeksforgeeks.org/best-python-libraries-for-machine-learning/> (bezocht op 11-06-2021).
 - [51] R. C. Martin. *Clean Code*. Pearson Education, Inc., 2018. ISBN: 9780132350884. (Bezocht op 12-06-2021).

Bijlagen

I Scope deelvraag 1

D1: Uit welke stappen bestaat een machine learning pipeline?	
Scope	<p>Binnen de scope:</p> <ul style="list-style-type: none">• In kaart brengen uit welke stappen een pipeline bestaat• Acties die in een stap worden uitgevoerd• Of het mogelijk is om stappen te versimpelen / abstracteren voor developers• Of het mogelijk is om stappen en acties te automatiseren <p>Buiten de scope:</p> <ul style="list-style-type: none">• Automatisering van stappen en acties• Een versimpeling van machine learning
Toelichting	Er wordt gekeken naar welke stappen er in een pipeline zitten. De theorie wordt vervolgens toegepast in een experiment. De nadruk ligt vooral op de mogelijkheid er is om stappen en acties te automatiseren en of machine learning versimpeld kan worden, niet dat er een uitwerking is.

Tabel 1: Scope deelvraag 1

II Scope deelvraag 2

D2: Hoe kan een orkestratietool verschillende cloud computing platformen beheren om een machine learning pipeline op te zetten?	
Scope	<p>Binnen de scope:</p> <ul style="list-style-type: none">• High-level uitleg over hoe een orkestratietool werkt• Inventarisatie met de "knock-out" methode• Criteria lijst voor orkestratietools• Opmerkelijke features die relevant zijn voor de PoC• Ervaring opdoen doormiddel van een pipeline te maken op twee cloud computing platformen <p>Buiten de scope:</p> <ul style="list-style-type: none">• Performance en snelheid
Toelichting	Frameworks dat cloud computing platformen beheert worden in kaart gebracht. Met knock-out criteria wordt de lijst verkort. Met een framework wordt een pipeline opgezet.

Tabel 2: Scope deelvraag 2

III Scope deelvraag 3

D3: Hoe ziet de architecturale blauwdruk van een applicatie, waarmee een platform-onafhankelijk machine learning pipeline opgezet kan worden, eruit?	
Scope	<p>Binnen de scope:</p> <ul style="list-style-type: none">• Technische tekeningen <p>Buiten de scope: -</p>
Toelichting	De literatuuronderzoek slaat op of de technische tekeningen gemaakt zijn volgens een standaard zoals UML. Dit komt niet terug als theorie maar de bronnen worden wel vermeld.

Tabel 3: Scope deelvraag 3

IV Scope hoofdvraag

H: In welke mate kan een machine learning pipeline worden geautomatiseerd onafhankelijk van de onderliggende cloud computing platform?	
Scope	De scope wordt bepaald na de requirement analyse.
Toelichting	Onderzoek naar documentatie van gebruikte framework(s).

Tabel 4: Scope hoofdvraag

V Mindmap van machine learning algoritmes



Figuur 1: Mindmap van machine learning algoritmes

VI ML theorie

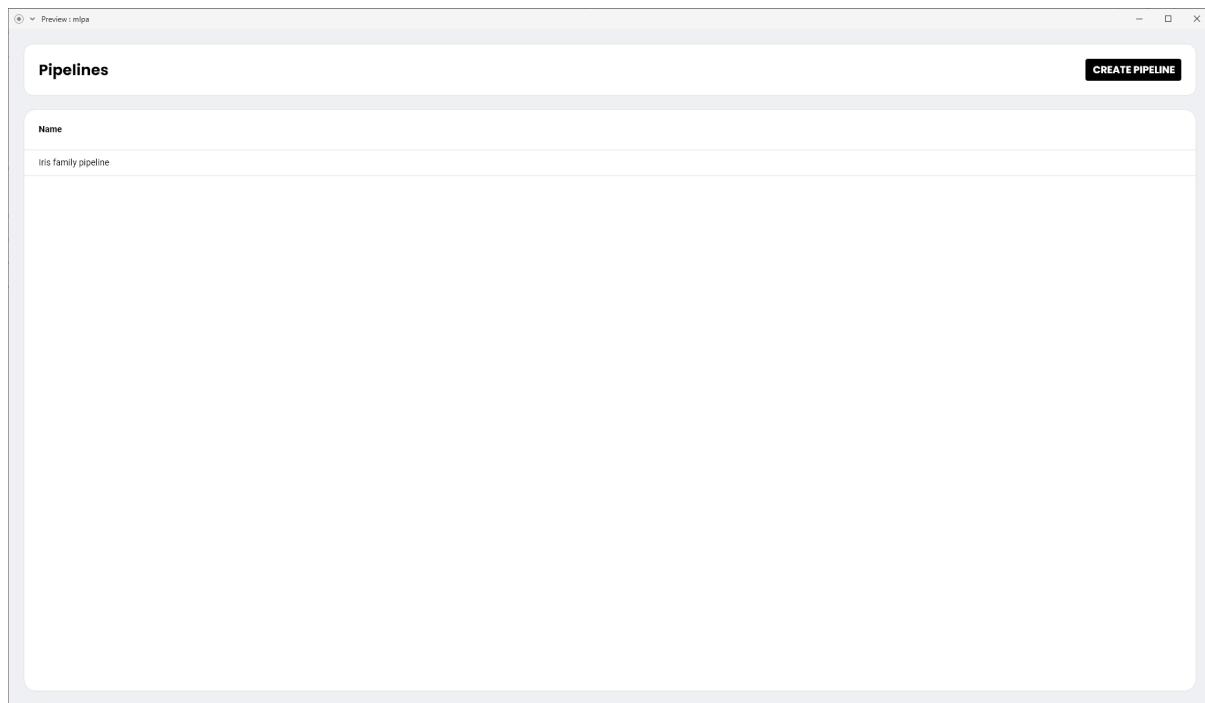
VII Machine learning pipeline experiment

VIII Stack configuratie van het experiment met Pulumi

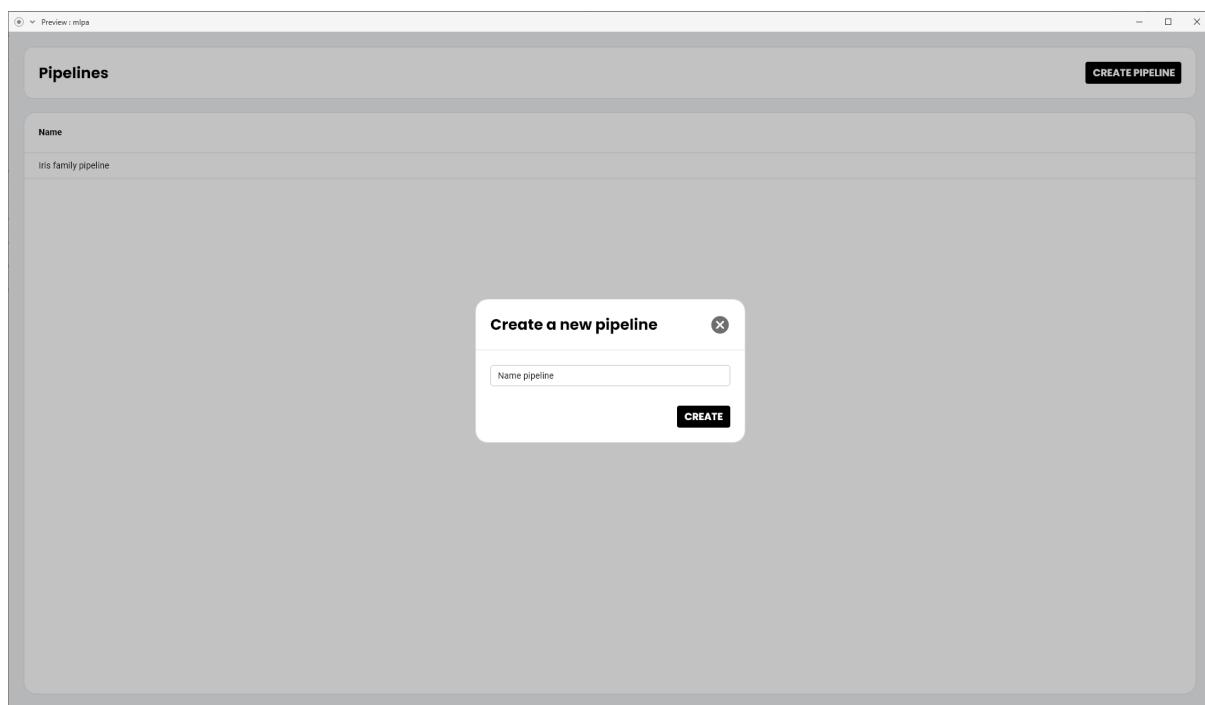
```
1 const CreateResource = (message: string) => async () => {
2   const computeNetwork = new compute.Network("network", {
3     autoCreateSubnetworks: true,
4     project: "disco-sky-312109"
5   })
6
7   const computeFirewall = new compute.Firewall("firewall", {
8     project: "disco-sky-312109",
9     network: computeNetwork.selfLink,
10    allows: [
11      { protocol: "tcp", ports: ["22", "80"] },
12    ],
13  })
14
15
16  const startupScript = `#!/bin/bash
17 echo "${message}" > index.html
18 nohup python -m SimpleHTTPServer 80 &
19
20 const computeInstance = new compute.Instance("instance", {
21   project: "disco-sky-312109",
22   machineType: "f1-micro",
23   metadataStartupScript: startupScript,
24   bootDisk: {
25     initializeParams: {
26       image: "debian-cloud/debian-9-stretch-v20181210",
27     },
28   },
29   networkInterfaces: [
30     { network: computeNetwork.id,
31       accessConfigs: [{}], // must be empty to request an ephemeral IP
32     },
33     serviceAccount: {
34       scopes: ["https://www.googleapis.com/auth/cloud-platform"],
35     },
36     zone: "europe-west1-c"
37   }, { dependsOn: [computeFirewall] })
38
39 const ip = computeInstance.networkInterfaces.apply(ni => ni[0].accessConfigs![0].natIp);
40
41 return { name: computeInstance.name, ip: ip }
42 }
```

Figuur 2: Stack configuratie van het experiment met Pulumi

IX Mock up - Overzicht pipeline en pipeline aanmaken

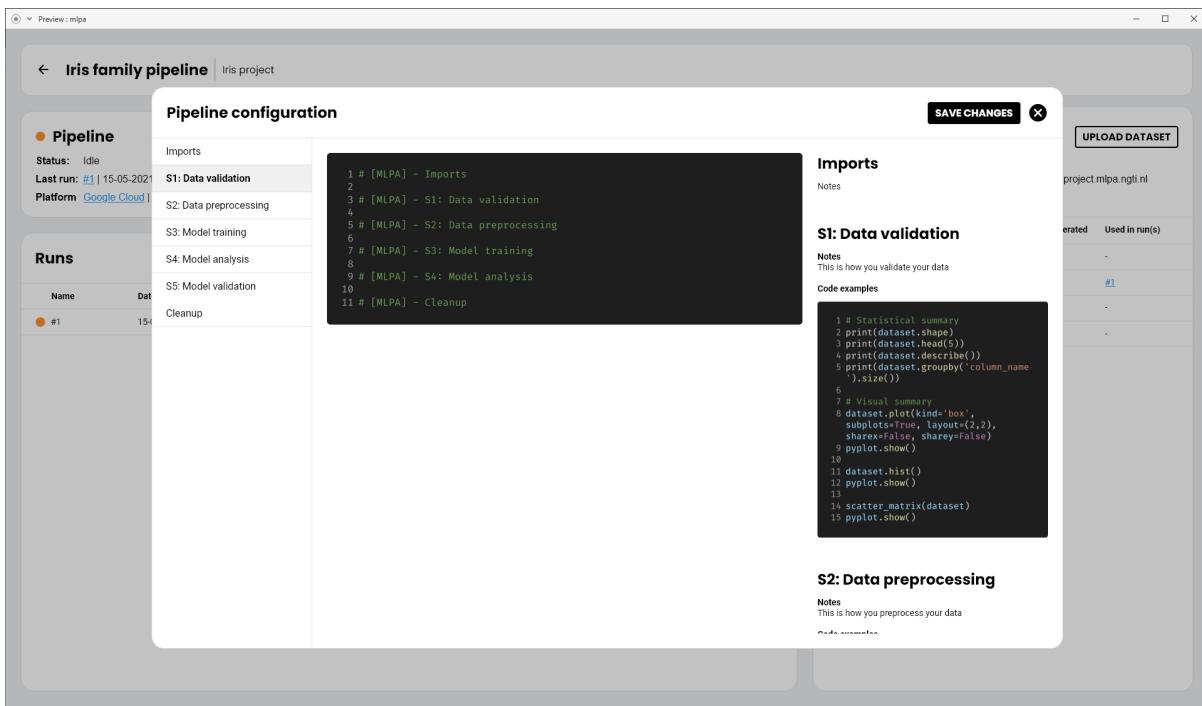


Figuur 3: Mock up van het pipeline overzicht



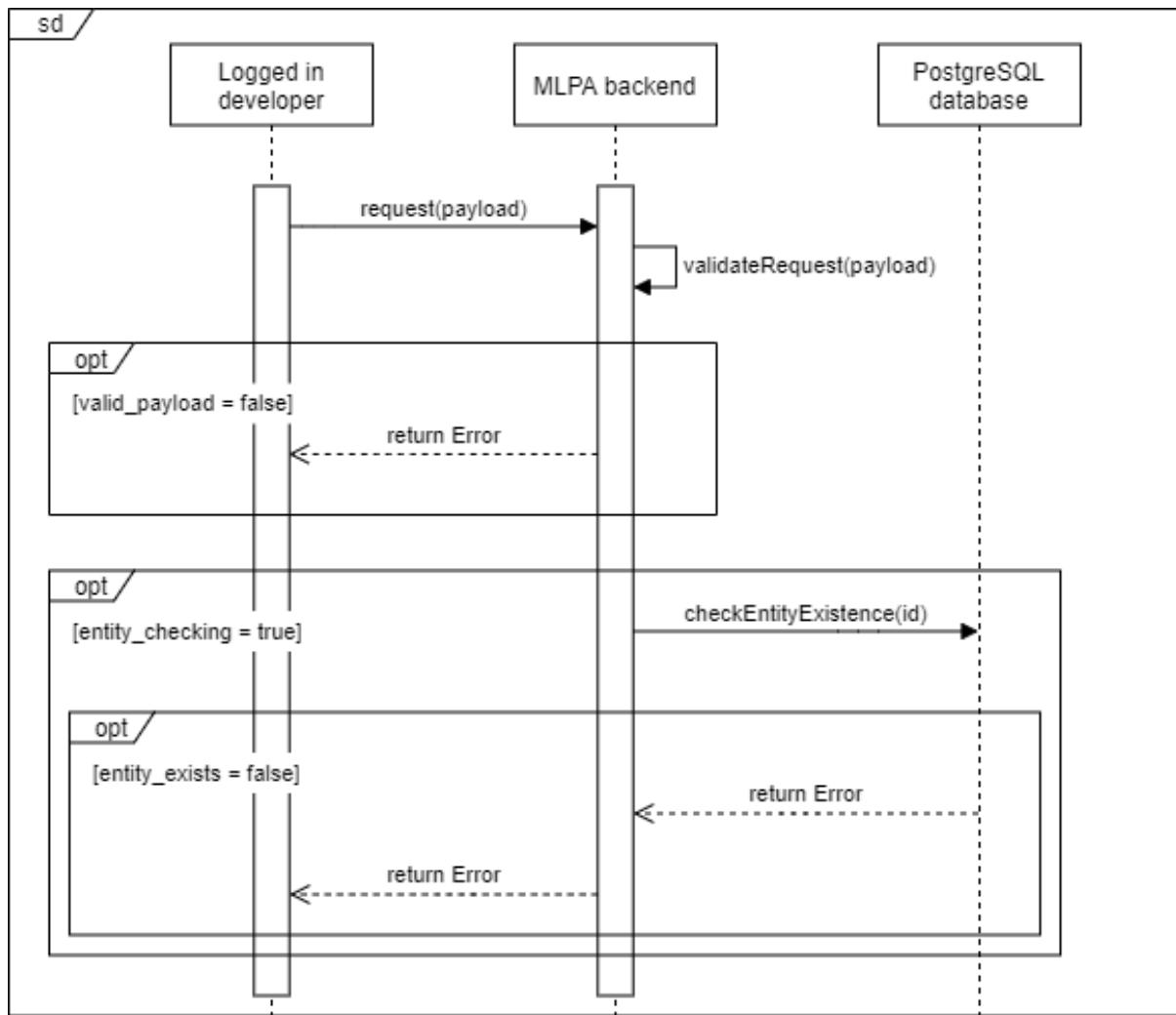
Figuur 4: Mock up van het aanmaken van een pipeline

X Mock up - Configuration dialoog



Figuur 5: Mock up van het configuratie dialoog

XI Validation sequence diagram



Figuur 6: Validation sequence diagram

XII Proof of concept code - *gc_getRunStatus()*

```
1 export const gc_getRunStatus = async (name: string, run: number, run_id: number) => {
2   const compute = new Compute()
3   const zone = compute.zone('europe-west4-a')
4   const vm = zone.vm(`${name}-${run}`)
5
6   const metadata = await vm.getMetadata()
7   const output = await vm.getSerialPortOutput()
8
9   await updateRun(run_id, { status: metadata[0].status, output: output[0] })
10
11   return await output[0]
12 }
```

Figuur 7: Code om de status van een VM op te halen in Google Cloud

XIII Proof of concept code - VM output in de frontend

Run

```
6.900000 2.500000
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: class
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: Iris-setosa 50
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: Iris-versicolor 50
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: Iris-virginica 50
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: dtype: int64
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: 0.9666666666666667
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: [[1 0 0]
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: [0 12 1]
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: [0 0 6]]
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: precision recall f1-score
support
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script:
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: Iris-setosa 1.00 1.00 1.00
11
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: Iris-versicolor 1.00 0.92 0.96
13
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: Iris-virginica 0.86 1.00 0.92
6
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script:
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: accuracy 0.97 30
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: macro avg 0.95 0.97 0.96
30
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script: weighted avg 0.97 0.97 0.97
30
Jun 12 13:13:32 test-pipeline-27 GCEMetadataScripts[2498]: 2021/06/12 13:13:32 GCEMetadataScripts: startup-script:
Jun 12 13:13:33 test-pipeline-27 google_metadata_script_runner[2498]: 2021/06/12 13:13:33 logging client: rpc error: code = Unauthenticated desc = transport: metadata: GCE metadata "instance/service-accounts/default/token?scopes=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Flogging.write" not defined
[ 174.666435] google_metadata_script_runner[2498]: 2021/06/12 13:13:33 logging client: rpc error: code = Unauthenticated desc = transport: metadata: GCE metadata "instance/service-accounts/default/token?scopes=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Flogging.write" not defined
[0;32m OK [0m] Started snap.google-cloud-sdk.gcloud...5a0b-4ff3-ad34-ca3b15b1681b.scope.
Jun 12 13:13:34 test-pipeline-27 systemd[1]: Started snap.google-cloud-sdk.gcloud.11babab84-5a0b-4ff3-ad34-ca3b15b1681b.scope.
```

Figuur 8: virtual machine (VM) output in de frontend

XIV Proof of concept code - *gc_stopRun*

```
1 export const gc_stopRun = async (name: string, run: number, run_id: number) => {
2   const compute = new Compute()
3   const zone = compute.zone('europe-west4-a')
4   const vm = zone.vm(`${name}-${run}`)
5
6   await vm.delete()
7
8   const _run = await fetchRun(run_id)
9
10  await updateRun(run_id, { status: 'TERMINATED' })
11  await updatePipeline(_run.pipeline.id, { status: 'IDLE' })
12 }
```

Figuur 9: Code om een virtual machine (VM) te verwijderen in Google Cloud