Onderzoek naar de compilatie en bind van een Coolgen programma via een Azure DevOps pipeline binnen de mainframe omgeving van ArcelorMittal Gent met proof of concept.

Dylan Vermeersch.

Scriptie voorgedragen tot het bekomen van de graad van
Professionele bachelor in de toegepaste informatica

Promotor: Dhr. L. Blondeel
Co-promotor: Dhr. D. Marichal
Academiejaar: 2023–2024
Eerste examenperiode

Departement IT en Digitale Innovatie .

HO
GENT

Woord vooraf

Voor u ligt mijn bachelorproef getiteld „Onderzoek naar de compilatie en bind van een Coolgen programma via een Azure DevOps pipeline binnen de mainframe omgeving van ArcelorMittal Gent met proof of concept". Deze bachelorproef is geschreven als een vereiste voor het behalen van mijn afstudeerdiploma in de richting „Toegepaste Informatica" aan HoGent. Gedurende de periode van februari tot mei 2024 heb ik me beziggehouden met het onderzoeken en het schrijven van deze bachelorproef.

Gedurende mijn tijd aan HoGent heb ik lang getwijfeld waar ik later mijn job van wou maken, er was niets dat mijn interesse voor de volle 100% kon wekken. Dit allemaal veranderde toen er op het einde van mijn tweede jaar in de opleiding Toegepaste Informatica de term mainframe naar mijn hoofd werd geslingerd. Ik was meteen heel erg geïnteresseerd in deze nieuwe term en vroeg mijn medestudenten Muhammed en Mehmet om meer informatie over het onderwerp. Het bleek al snel dat het een hot topic was binnen de wereld van big IT, we hebben toen ook de vraag gesteld aan Dhr. Leendert Blondeel of het mogelijk was om meer duiding te geven bij de specialisatierichting Mainframe Expert. Dit verzoek werd met veel enthousiasme onthaald en vrijwel meteen werd een virtueel gesprek ingepland met mijzelf, Mehmet en Muhammed om ons meer te vertellen over mainframe en hoe een jaar van een mainframe student op HoGent eruitziet. Meteen tijdens het gesprek wist ik dat ik mijn afstudeerrichting gevonden had, de passie was en is nog altijd heel erg groot bij Dhr. Leendert Blondeel en trok je direct aan zoals normaal alleen een magneet dat kan.

Het werd tijdens dat derde jaar alleen maar duidelijker dat ik de juiste weg ingeslagen was, we ontmoetten enorm veel mensen die net zoals Dhr. Leendert Blondeel heel erg gepassioneerd zijn over mainframe en alles daaromtrent. Veel van deze mensen zijn nog altijd goed bereikbaar via mail ondanks het feit dat ze bij heel grote bedrijven zitten, kan ik ze nog altijd bereiken voor allerhande vragen over bijeenkomsten, mainframe topics of zelfs om de ondervindingen van mijn bachelorproef mee te delen. Het heeft me duidelijk geen windeieren gelegd om de keuze voor mainframe te maken.

Graag wil ik mijn begeleider, Dhr. Leendert Blondeel, bedanken voor zijn uitstekende begeleiding en ondersteuning gedurende dit onderzoek. Zijn begeleiding

heeft mijn leermogelijkheden gemaximaliseerd, waarvoor ik hem zeer dankbaar ben. Ook wil ik Dhr. Didier Marichal van ArcelorMittal Gent bedanken voor zijn bijdrage aan dit onderzoek als co-promotor.

Tot slot wil ik ook het mainframe systeembeheer team van ArcelorMittal Gent bedanken, ik kon elk moment van de dag bij hun terecht met allerhande vragen of zelfs om een gezellig babbeltje te slaan. Ze zorgden ervoor dat ik me heel erg thuis voelde binnen het team. Graag wil ik ook nog mijn familie en vrienden bedanken voor hun steun gedurende mijn onderzoeksproces.

Ik wens u veel leesplezier.

Dylan Vermeersch
Oostakker, 24 mei 2024

Samenvatting

Deze scriptie onderzoekt op welke manier er een Azure DevOps pipeline geïntegreerd kan worden om de compile/bind van bestaande en toekomstige Coolgen programma's uit te voeren en wat de gevolgen van dit nieuw systeem zijn op de werking van het huidige systeem binnen de mainframe omgeving van ArcelorMittal Gent.

Momenteel gebruikt ArcelorMittal Gent Rocket Software MSP (Manager Products) in samenwerking met PANAPT meta dictionaries om de compile en bind uit te voeren van hun Coolgen programma's. Ze willen dat er onderzocht wordt of dit te vervangen valt met een modernere oplossing zoals een pipeline gebaseerd systeem in samenwerking met Git.

Het doel van dit onderzoek is dan ook om een duidelijk beeld te scheppen over welke technologieën er gebruikt kunnen/moeten worden om een Azure DevOps pipeline op te zetten die als taak heeft om de compile/bind van Coolgen programma's te verzorgen. Het doel is om dit ook aan te tonen met behulp van een proof of concept.

De methodologie die opgesteld is om dit onderzoek uit te voeren bestaat uit zeven fases. Zo is fase één voorbereidend werk, in deze fase wordt zoals de naam zegt voorbereidingen getroffen voor het onderzoek. Dit bestaat onder meer uit het opmaken van een literatuurstudie om de expertise in het onderzoek te verhogen. In fase twee van de methodologie wordt er gekeken om de volledige werking van het huidige systeem in kaart te brengen en te onderzoeken. De derde fase is de configuratie van IBM DBB, hierin wordt IBM DBB volledig klaargestoomd om te kunnen werken binnen de proof of concept. Deze fase bestaat vooral uit het opzoeken van datasets en met behulp van de vorige fase zou dit vrij makkelijk moeten gaan doordat het huidige systeem in kaart is gebracht. In fase vier wordt de proof of concept opgezet en gerealiseerd, zo wordt er hierin aanpassingen gemaakt aan de groovy scripts en aan het build proces van IBM DBB. De vijfde fase focust zich op een analyse van de resultaten van de proof of concept en in de zesde fase worden er aan de hand van de resultatenanalyse conclusies getrokken. De zevende en laatste fase is om de scriptie af te werken zodat deze correct en volledig is.

De resultaten van de proof of concept tonen aan dat het mogelijk is om met behulp

van IBM DBB en Azure DevOps een naadloos werkend systeem te ontwikkelen om simultaan te werken met het huidige systeem. De proof of concept bewees dat het ontwikkelde systeem op basis van IBM DBB en Azure DevOps ook helemaal aangepast kan worden naar de manier van werken binnen de mainframe omgeving van ArcelorMittal Gent. Deze resultaten zijn vooral van belang voor ArcelorMittal Gent aangezien hun mainframe omgeving volledig aangepast is naar hun manier van werken met zelfgemaakte software, desondanks kunnen andere bedrijven zeker en vast baten bij het lezen van deze scriptie door de concepten en ideeën die gebruikt zijn binnen dit onderzoek zouden heel veel fouten en problemen vermeden kunnen worden.

Uit de resultaten kan de conclusie getrokken worden dat Azure DevOps en IBM DBB een goed team vormden binnen dit onderzoek en dat deze zeker en vast de capaciteiten hebben om op lange termijn weleens het huidige systeem te vervangen. Al is hiervoor wel nog extra onderzoek nodig om de complexiteit van de volledige mainframe omgeving van ArcelorMittal Gent in kaart te brengen.

Inhoudsopgave

Lijst van figuren

1

Inleiding

Deze bachelorproef gaat over op welke manier er een Azure DevOps pipeline kan geïntegreerd worden binnen ArcelorMittal Gent om hun bestaande en toekomstige Coolgen programma's te compilen/binden. Dit onderzoek is begonnen omdat ArcelorMittal Gent graag hun mainframe omgeving wil moderniseren en automatiseren, meer bepaald gaat het in dit onderzoek over het automatiseren van de compile/bind van Coolgen programma's.

1.1. Probleemstelling

Bij ArcelorMittal Gent wordt er gekeken hoe men hun mainframe omgeving kan moderniseren, op die manier kunnen ze efficiënt gebruikmaken van de nieuwe technologieën die de laatste jaren in opmars zijn zoals automatische DevOps pipelines. Door zaken zoals het compileren/binden van programma's automatisch te starten via een pipeline wordt er aan tijd gewonnen wat er dus voor zorgt dat een taak efficiënter wordt. Bij ArcelorMittal Gent zijn ze daarom opzoek naar een manier om het compile/bind proces voor hun Coolgen programma's te automatiseren met behulp van een Azure DevOps pipeline. Momenteel is het nog niet duidelijk wat ervoor nodig is om zo'n Azure DevOps pipeline werkende te krijgen voor bestaande en toekomstige Coolgen programma's binnen ArcelorMittal Gent.

1.2. Onderzoeksvraag

Op welke manier kan er een Azure DevOps pipeline geïntegreerd worden om de compile/bind van bestaande en toekomstige Coolgen programma's uit te voeren. Wat zijn de gevolgen voor het huidige compile/bind systeem voor Coolgen programma's na het integreren van de bekomen pipeline met de mainframe omgeving van ArcelorMittal Gent.

1.3. Onderzoeksdoelstelling

Het onderzoek moet een duidelijk beeld geven over welke technologieën er gebruikt worden om een Azure DevOps pipeline op te zetten die als taak heeft om de compile/bind van Coolgen programma's te verzorgen. Een proof of concept wordt opgesteld om zo'n Azure DevOps pipeline uit te werken die de compile/bind van bestaande en toekomstige Coolgen programma's kan uitvoeren.

1.4. Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:
In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie.
In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.
In Hoofdstuk 4 wordt de werking van het huidige systeem in verband met de compilatie en bind van een programma toegelicht.
In Hoofdstuk 5 wordt de uitvoering van de Proof Of Concept toegelicht en welke aanpassingen moesten gebeuren aan het product DBB van IBM.
In Hoofdstuk 6, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2

Stand van zaken

In de snel evoluerende wereld van technologie speelt de mainframe nog altijd een essentiële rol als krachtige en betrouwbare computerinfrastructuur. Het is de ruggengraat van talloze organisaties, variërend van grote bedrijven tot overheidsinstanties, die vertrouwen op mainframes voor het verwerken van bedrijfskritieke workloads. Met de opkomst van moderne applicatie-ontwikkeling en de behoefte aan automatisering van het software ontwikkelproces, hebben mainframes zich aangepast aan het veranderende landschap. Dit doen ze door het aanbieden van de DevOps werkwijze en geavanceerde technologieën zoals Git, Azure DevOps en IBM Dependency Based Build ook wel bekend als IBM DBB.

Deze literatuurstudie onderzoekt de samenstelling van mainframe-technologie, IBM Dependency Based Build, Azure DevOps en Git workflows als een krachtige combinatie om een brug te slaan tussen het legendarische mainframe-erfgoed en het moderne automatisatie-gedreven software ontwikkelproces. Het onderzoek verkent de essentie van mainframes en hun evolutie door de jaren heen, evenals de cruciale rol die ze blijven spelen in het ondersteunen van kritieke bedrijfsprocessen.

Daarnaast wordt er dieper ingegaan op IBM Dependency Based Build, een krachtige tool ontwikkeld door IBM om mainframes te verbinden met moderne DevOps tools zoals Azure DevOps en Git. De studie onderzoekt de kenmerken en voordelen van IBM DBB en hoe dat het mogelijk maakt om een mainframe landschap open te stellen tot de DevOps werkwijze om op die manier software te ontwikkelen op de wijze waarop veel moderne platformen dat ook doen.

Verder wordt de focus gelegd op Azure DevOps, een krachtige tool-set die onder andere Azure Repos en Azure Pipelines bevat. Beide zijn noodzakelijk voor een goede DevOps werkwijze op te zetten. De studie verkent de mogelijkheden van de

tool-set, zoals de verschillende onderdelen ervan en hoe deze bijdragen aan een DevOps werkomgeving.

Tot slot komt ook Git workflows aan bod, dit is een manier van werken om zo de flow van het software ontwikkelproces te bepalen. Dit is noodzakelijk voor de integriteit van de software die ontwikkeld wordt te bewaren. Er wordt in de studie onderzocht wat een Git workflow is en welke het best aanleunt tegen de mainframe waarden die zo hoog aangeschreven staan zoals betrouwbaarheid, integriteit en veiligheid.

2.1. De mainframe

2.1.1. Etymologie van de term mainframe

De term mainframe is afgeleid van het Engelse woord „frame", dat in dit geval verwijst naar de behuizing of structuur van de computer. Het woord „main" duidt op de centrale rol van deze computer in een gegevensverwerkingsomgeving. Een mainframe was bedoeld als het belangrijkste systeem in een computerinstallatie, waar andere randapparatuur en terminals op waren aangesloten (IBM, 2024e).

De oorsprong van de term mainframe kan worden toegeschreven aan de evolutie van computersystemen van die tijd. In de beginjaren van mainframe werden ze meestal aangeduid als grote computers of elektronische rekenmachines. Naarmate de technologie vorderde en de computers krachtiger en complexer werden, ontstond de behoefte aan een specifieke term om deze geavanceerde systemen te beschrijven (IBM, 2024e).

2.1.2. Historie van de IBM mainframe

De IBM mainframe heeft een rijke geschiedenis die teruggaat tot de vroege dagen van de computertechnologie.

IBM 701 Electronic Data Processing machine (1952)

In 1952 werd de IBM 701 gelanceerd als een geavanceerde elektronische gegevensverwerkingsmachine. Het was ontworpen om wetenschappelijke berekeningen en gegevensverwerkingstaken uit te voeren en bood meer rekenkracht en geheugencapaciteit dan eerdere computersystemen aan (IBM, 2024a).

De IBM 701 maakte gebruik van vacuümbuizen als belangrijkste elektronische com-

ponenten en kon ongeveer 10.000 optellingen per seconde uitvoeren. Het systeem werd voornamelijk gebruikt door overheidsinstanties, laboratoria en grote bedrijven voor complexe wetenschappelijke en technische berekeningen (IBM, 2024a).

De IBM 701 markeerde een belangrijke ontwikkeling in de computertechnologie, aangezien het een van de eerste commercieel succesvolle computers was die specifiek werd ontworpen voor gegevensverwerking. Het opende de deur naar geavanceerdere computersystemen en legde de basis voor toekomstige mainframes (IBM, 2024a).

IBM System/360 (1964)

De ontwikkeling van de IBM/360 begon in de late jaren 1950 als een ambitieus project binnen IBM om een nieuwe generatie computersystemen te creëren die compatibiliteit zouden bieden tussen verschillende modellen. Het doel was om een reeks computers te ontwikkelen die zowel kleinere als grotere organisaties kon bedienen (IBM, 2024b).

Op 7 april 1964 werd de IBM/360 officieel geïntroduceerd en werd het de eerste commercieel succesvolle mainframe computerreeks. Het systeem bood een breed scala aan modellen met verschillende prestatieniveaus en configuraties, waardoor het aan de behoeften van verschillende organisaties kon voldoen (IBM, 2024b).

De IBM/360 was revolutionair omdat het een gemeenschappelijke architectuur introduceerde die compatibiliteit bood tussen de verschillende modellen. Hierdoor konden klanten hun investeringen in software en hardware beschermen, omdat programma's op meerdere systemen konden draaien. Het was ook een van de eerste computersystemen die gebruikmaakte van geïntegreerde schakelingen (IBM, 2024b).

De IBM/360 had een enorme impact op de computerindustrie en droeg bij aan de standaardisatie van computerarchitectuur. Het systeem werd breed geadopteerd door bedrijven, overheden en academische instellingen over de hele wereld en legde de basis voor latere ontwikkelingen in de mainframe-technologie (IBM, 2024b).

IBM System/370 (1970)

De IBM/370 werd gelanceerd als opvolger van de IBM/360 en introduceerde belangrijke technologische verbeteringen, waaronder een uitgebreidere instructieset, ver-

beterde virtualisatiemogelijkheden en grotere geheugencapaciteit. Deze verbeteringen maakten het systeem krachtiger en veelzijdiger (IBM, 2024c).

Een belangrijk kenmerk van de IBM/370 was de ondersteuning voor virtual memory, waardoor meerdere programma's tegelijkertijd konden uitgevoerd worden en gebruik maken van het beschikbare geheugen. Dit leidde tot verbeterde systeemprestaties en efficiënter gebruik van resources (IBM, 2024c).

De IBM/370 mainframe werd breed gebruikt door bedrijven en overheidsinstanties voor diverse toepassingen, waaronder gegevensverwerking, transactionele systemen, wetenschappelijke berekeningen en databasebeheer. Het bood hogere prestaties en schaalbaarheid, waardoor het kon voldoen aan de groeiende behoeften van organisaties (IBM, 2024c).

IBM System ZSeries (2000)

De IBM zSeries, gelanceerd in het jaar 2000, was een belangrijke mijlpaal voor de mainframe-industrie. Het bood verschillende baanbrekende kenmerken en voordelen die een grote impact hadden.

Een van de belangrijkste aspecten van de zSeries was de aanzienlijke verbetering in prestaties en schaalbaarheid. Het systeem was in staat om enorme werklasten te verwerken en te voldoen aan de groeiende behoeften van bedrijven. Dit maakte het een krachtige keuze voor organisaties die behoefte hadden aan grote rekenkracht en verwerkingscapaciteit (IBM, 2024d).

Naast prestatieverbeteringen stond de zSeries bekend om zijn ongeëvenaarde betrouwbaarheid en beschikbaarheid. Het bevatte geavanceerde functies zoals redundantie, fouttolerantie en hot-swappable componenten. Deze kenmerken minimaliseerden ongeplande downtime en waarborgden een hoge beschikbaarheid van systemen, wat van cruciaal belang was voor bedrijfskritieke toepassingen (IBM, 2024d).

Een ander belangrijk aspect van de zSeries was de ondersteuning voor moderne technologieën. Het introduceerde onder andere Linux op mainframes, waardoor organisaties zowel mainframe- als open source-technologieën op één platform konden gebruiken. Dit opende de deur voor een breed scala aan toepassingen en bood flexibiliteit in de ontwikkeling en implementatie van software (IBM, 2024d).

Beveiliging is altijd een cruciale factor geweest in de mainframe-wereld, en de zSeries stelde op dit gebied niet teleur. Het bood geavanceerde beveiligingsfuncties,

zoals ingebouwde encryptie, toegangscontrolemechanismen en auditmogelijkheden. Deze functies waarborgden de integriteit en vertrouwelijkheid van gegevens, wat essentieel is in omgevingen waar gevoelige informatie wordt verwerkt (IBM, 2024d).

Een ander belangrijk voordeel van de zSeries was de mogelijkheid om naadloos te integreren met bestaande legacy-systemen en applicaties. Hierdoor konden organisaties waardevolle bedrijfsactiva behouden en moderniseren zonder de noodzaak van grootschalige herontwikkeling. Dit zorgde voor een soepele overgang naar de nieuwe technologie en minimaliseerde de verstoring van bestaande processen (IBM, 2024d).

Tijdlijn van de grootste IBM mainframes

1952 IBM 701-de eerste commerciële mainframe van IBM.

1964 IBM System/360-de eerste mainframe-reeks die compatibiliteit bood over meerdere modellen.

1970 IBM System/370-deze mainframe bood nieuwe mogelijkheden zoals virtueel geheugen en verbeterde instructiesets.

1980 IBM System/38-mainframe met microprocessoren en een nieuwe programmeertaal genaamd „CPF" (Control Program Facility).

1985 IBM ES/9000-ondersteuning voor geavanceerde besturingssystemen zoals MVS/ESA en VM/ESA.

1990 IBM System/390-opvolger van System/370, met de focus op verbeterde prestaties, beveiliging en schaalbaarheid.

2000 IBM zSeries-deze mainframe had betere ondersteuning voor internet en webgebaseerde toepassingen, en verbeterde virtualisatie- en partitioneringsmogelijkheden.

2005 IBM System z9-de opvolger van de zSeries, met een focus op betere prestaties en beveiliging, en verbeterde virtualisatie- en partitioneringsmogelijkheden.

2010 IBM zEnterprise System-een hybride mainframe dat zowel traditionele mainframe- als BladeCenter-technologie combineerde.

2015	IBM z13-de opvolger van zEnterprise System, met verbeterde prestaties, beveiliging en betere ondersteuning voor cloud computing en big-data analyse.
2021	IBM z15-deze mainframe kreeg betere ondersteuning voor hybride cloud-omgevingen en AI-workloads.
2023	IBM z16-de nieuwste generatie mainframe van IBM, het biedt real-time AI-inferentie en is het eerste kwantumveilige systeem op de markt.

(Elliot, 2015) (IBM, g.d.-c) (IBM, g.d.-b)

2.1.3. Concurrentie op het mainframe platform

1960s

In de jaren 1960 had IBM een dominante positie op de mainframe-markt. Ze waren de toonaangevende leverancier van mainframes en hun System/360-serie was een belangrijke mijlpaal in de computerindustrie. Echter, waren er ook andere belangrijke concurrenten die IBM uitdaagden. Burroughs Corporation was daar een van, met hun B5000-serie die bekend stond om zijn geavanceerde architectuur en programmeertaal. Control Data Corporation (CDC) was ook een belangrijke concurrent, met hun CDC 6600 die destijds bekend stond als 's werelds snelste computer (CH, 2024).

1970s

In de jaren 1970 bleef IBM een dominante positie behouden op de mainframe-markt, maar er waren nieuwe concurrenten die uitdagingen boden. Een belangrijke concurrent was Digital Equipment Corporation (DEC), een bedrijf dat bekend stond om zijn minicomputers. DEC bracht echter ook mainframes op de markt, zoals de DECsystem-10 en DECsystem-20, die aantrekkelijk waren voor verschillende organisaties (Society, 2017).

Een andere belangrijke speler was Honeywell, dat zijn eigen reeks mainframes aanbood, zoals de Honeywell 6000-serie. Deze systemen waren populair in sectoren zoals banken en overheidsinstanties (Society, 2017).

Bovendien begonnen in de jaren 1970 ook nieuwe bedrijven, zoals Amdahl Corporation en Hitachi, de mainframe-markt te betreden. Amdahl Corporation, opgericht

door een voormalige IBM-manager, bood mainframes aan die compatibel waren met IBM-systemen, maar tegen lagere prijzen (Society, 2017).

1980s

Doorheen de jaren 1980 werd de concurrentie op de mainframe-markt intenser, met verschillende spelers die de dominante positie van IBM probeerden uit te dagen. Een van de grootste concurrenten was Digital Equipment Corporation (DEC), dat in deze periode zijn VAX-computers lanceerde. De VAX-systemen waren krachtige machines die in staat waren om complexe taken uit te voeren en werden populair in bedrijfsomgevingen (Society, 2017).

Een andere opkomende concurrent was Amdahl Corporation, dat IBM-compatibele mainframes aanbood tegen lagere prijzen. Amdahl wist een aanzienlijk marktaandeel te veroveren en werd een belangrijke speler in de mainframe-industrie (Society, 2017).

Ook Hewlett-Packard (HP) betrad de mainframe-markt met zijn HP 3000-serie. Deze systemen waren gericht op kleinere organisaties en boden een combinatie van mainframe-functionaliteit met de gebruiksvriendelijkheid van minicomputers (Society, 2017).

Naast deze concurrenten zette IBM zelf ook belangrijke ontwikkelingen voort. In 1980 introduceerde IBM de IBM 3081-mainframeserie, die verbeterde prestaties bood ten opzichte van eerdere modellen. Later in het decennium lanceerde IBM de 3090-serie, die geavanceerde mogelijkheden bood, zoals verbeterde geheugencapaciteit en verwerkingssnelheid (Society, 2017).

1990s

Gedurende de jaren ’90 was IBM nog altijd leider in de markt, maar er waren ook concurrenten die uitdagende alternatieven aanboden. Zoals Amdahl, dat bracht in deze periode de 9000-serie mainframes uit, die concurrerende prestaties en betrouwbaarheid boden (Ceruzzi, 2003).

Een andere concurrent die ook al in de jaren ’80 aanwezig was is Hitachi met zijn Hitachi Mainframe Systems. Deze systemen waren populair in de Aziatische markt en boden krachtige verwerkingsmogelijkheden en schaalbaarheid (Ceruzzi, 2003).

Een opvallende ontwikkeling in de jaren 1990 was de opkomst van open systemen

en de Unix-besturingssystemen. Sun Microsystems was een belangrijke speler met zijn Sun Enterprise-servers, die draaiden op het Solaris-besturingssysteem. Deze systemen werden vaak gebruikt voor zware reken- en databasetoepassingen (Ceruzzi, 2003).

Daarnaast begon IBM zelf ook met het aanbieden van open-systemen op basis van de IBM RS/6000-architectuur, die het AIX-besturingssysteem draaiden. Deze systemen combineerden de kracht van mainframes met de flexibiliteit van open systemen (Ceruzzi, 2003).

2000s

Bij het begin van het nieuwe millennium bleef IBM veruit de grootste speler in de mainframe-wereld. Toch waren er nog altijd geduchte concurrenten zoals Fujitsu, een Japans technologiebedrijf. Fujitsu bood zijn eigen lijn van mainframes aan, zoals de Fujitsu BS2000-serie, die bekend stond om zijn betrouwbaarheid en schaalbaarheid (LaMonica, 2004).

Een andere uitdager was Hewlett-Packard (HP), dat de non-stop servers aanbood. Deze waren gericht op transactionele verwerking en waren populair in sectoren zoals banken en financiële dienstverlening (LaMonica, 2004).

Naast deze gevestigde spelers begon de opkomst van cloud computing in de jaren 2000 de dynamiek in de mainframe-markt te veranderen. Bedrijven zoals Amazon Web Services (AWS) en Google Cloud Platform (GCP) boden schaalbare en flexibele cloudinfrastructuur aan, waardoor organisaties een alternatief kregen voor het beheren van hun eigen mainframes (AWS, g.d.) (Google, g.d.).

IBM speelde ook in op de opkomst van cloud computing en introduceerde zijn eigen mainframe-gebaseerde cloudoplossingen, zoals IBM Cloud en IBM Z als een Service. Deze diensten boden klanten de mogelijkheid om mainframe-functionaliteit te benutten in een cloudomgeving (IBM, g.d.-a).

2010s

In de jaren 2010 bleef IBM zijn dominante positie voort zetten in de mainframe-markt, met zijn IBM Z-systemen die bekend stonden om hun schaalbaarheid, beveiliging en betrouwbaarheid. IBM investeerde voortdurend in de ontwikkeling van nieuwe mainframe-technologieën en introduceerde regelmatig verbeterde versies van zijn mainframe-systemen (IBM, 2024d).

Naast IBM waren er enkele andere spelers die zich in de mainframe-markt begaven. Een belangrijke concurrent was Oracle Corporation, dat zijn Engineered Systems-lijn aanbood, waaronder de Oracle SuperCluster en de Oracle Exadata Database Machine. Deze systemen combineerden high-performance hardware met geoptimaliseerde software en waren specifiek gericht op gegevensverwerking en databasebeheer (Oracle, g.d.).

Een andere opkomende trend in die periode was de verschuiving naar gevirtualiseerde en software gedefinieerde infrastructuren. Bedrijven zoals VMware, met zijn virtualisatieoplossingen, en OpenStack, met zijn open-source cloudbeheerplatform, begonnen aan populariteit te winnen. Hoewel deze technologieën niet rechtstreeks mainframe-gericht waren, boden ze alternatieve manieren om IT-infrastructuur te schalen en beheren. Bovendien speelden cloudproviders zoals Amazon WebServices (AWS), Microsoft Azure en Google Cloud Platform (GCP) een steeds grotere rol in de IT-industrie (Google, g.d.) (AWS, g.d.) (VMWare, g.d.).

Heden

In de hedendaagse markt is IBM nog altijd heer en meester met zijn IBM Z-systemen. Deze systemen zijn geoptimaliseerd voor high-performance computing, beveiliging, schaalbaarheid en worden nog steeds gebruikt door organisaties over de hele wereld voor kritieke workloads en bedrijfsprocessen.

Naast IBM hebben andere technologiebedrijven, zoals Fujitsu en Unisys, nog steeds een aanwezigheid in de mainframe-markt. Fujitsu biedt zijn BS2000-mainframes aan, die zich richten op betrouwbaarheid en schaalbaarheid. Unisys heeft zijn ClearPath Libra- en Dorado-systemen die geschikt zijn voor bedrijfskritieke applicaties (Fujitsu, g.d.) (Unisys, g.d.).

Een opvallende trend is de verschuiving naar hybride cloudarchitecturen en de opkomst van cloud-native technologieën. Bedrijven zijn op zoek naar manieren om mainframe-technologie te integreren met cloudoplossingen, zoals IBM Cloud, Amazon Web Services (AWS), Microsoft Azure en Google Cloud Platform (GCP). Dit stelt organisaties in staat om de schaalbaarheid, flexibiliteit en kostenefficiëntie van de cloud te benutten, terwijl ze nog steeds kunnen profiteren van de kracht en betrouwbaarheid van mainframes voor hun kritieke workloads (Google, g.d.) (AWS, g.d.).

Een andere belangrijke ontwikkeling in de mainframe-markt is de focus op beveiliging. Met de groeiende dreiging van cyberaanvallen en gegevensinbreuken

is dit een topprioriteit geworden voor organisaties. IBM Z-systemen hebben ingebouwde beveiligingsfuncties zoals IBM Secure Service Container en Secure Execution voor het beschermen van gevoelige gegevens en het voorkomen van ongeautoriseerde toegang (IBM, g.d.-c).

2.2. IBM Dependency Based Build

In dit hoofdstuk wordt er beschreven wat IBM Dependency Based Build is en waarvoor het in dit onderzoek zal gebruikt worden. Er wordt ook besproken hoe IBM DBB werkt zowel op de voorgrond als op de achtergrond.

2.2.1. Introductie tot IBM Dependency Based Build?

Geschiedenis

Een van de eerste versies van DBB is versie 1.0.1, deze versie was uitgekomen in juni 2018 en had als nieuwe features over de initiële 1.0.0 versie dat het JCL kon submitten en een dat het object beheer door middel van eigenaarsrollen verbeterd is. Bij versie 1.0.1 werden ook nog eens 2 extra sample programma's geïntroduceerd namelijk PL/I Helloworld en DB2 Bind Sample in Mortgage Application.

Terwijl de versie van DBB in 2018 nog maar heel pril is worden er aan sneltempo versies met bijhorende extra features en verbeteringen uitgebracht. Zo is er op het einde van 2018 al een versie 1.0.3 uitgebracht die extra features zoals:

- Versie 1.0.2
  - Multi-thread build support
  - Binary en load module kopie support
  - ISPF interactieve gateway support voor TSOExec en ISPFExec
  - Opvangen en opslaan van indirecte dependencies
  - Automatische Groovy script caching
  - Build properties kunnen opgebouwd zijn uit andere build properties
  - DBB configuratie properties
  - SMF record generatie
  - DBB Build manager introductie
- Versie 1.0.3
  - SMF record generatie

  - DBB Build manager introductie

In 2019 blijven er op hetzelfde tempo nieuwe versies uitkomen, elk kwartaal wordt er een nieuwe versie uitgerold van DBB zo zit IBM tegen eind 2019 al aan versie 1.0.7. De belangrijkste features zijn onder andere:

- Versie 1.0.4
  - zFS directories in DD statements
  - FIPS 140-2 compliance
  - Error en warning messages zijn nu te vinden in het knowledge center
  -
- Versie 1.0.5
  - Tar/gzip file support voor dependencies
  - Non-roundtrippable character detectie door de migratie tool
  - Linux on IBM Z support
- Versie 1.0.6
  - Introducering van Z Open Automation Utilities (ZOAU)
  - Support voor aanpassingen aan database schema
  - Support voor Db2 voor z/OS
  - YAML bestanden voor build configuraties
  - Swagger API documentatie
- Versie 1.0.7
  - Toevoeging van nieuwe ZOAU functionaliteit
  - File tagging support voor CopyToHFS commando

Doorheen 2020 zwakte het aantal versie-updates af naar slechts 2, zo werd er een update in maart en in juni uitgebracht voor DBB. Respectievelijk versies 1.0.8 en 1.0.9 hiervan zijn er een aantal nieuwe features opgelijst:

- Versie 1.0.8
  - File tagging support voor het CopyToHFS commando bij non-ASCII en UTF-8 gecodeerde bestanden
  - Nieuwe copy mode toegevoegd bij het CopyToHFS commando dat ASA carriage control characters behoudt
  - Support om het CopyToPDS commando als een build stap te registreren in de build report

- Versie 1.0.9
  - Personal daemon beschikbaar als toevoeging op de bestaande shared daemon

Het bleef wat betreft versie-updates heel stil in het jaar 2021 en 2022 al was er in 2022 nog een laatste versie-update voor versie 1.0x van DBB. Die update werd doorgevoerd in maart 2022 en bepaald tot op heden de huidige versie van DBB 1.0x:

- Versie 1.0.10
  - Apache Groovy 4.0 upgrade

(IBM, 2022)

In oktober 2021 komt er een nieuwe versie uit van DBB namelijk de versie 1.1x, deze versie heeft 4 updates gekregen sinds zijn ontstaan, de updates zijn niet zo talrijk als in versie 1.0x. Met de laatste update van versie 1.1x in maart 2023 zit men aan de huidige versie van DBB 1.1x, er zijn nog updates uitgekomen in juni 2021, oktober 2021 en maart 2022. De belangrijkste features per versie zijn de volgende:

- Versie 1.1.0
  - DBB Web Applicatie beschikbaar als een Red Hat OpenShift Container Platform (OCP) cluster
  - Integratie met z/OS Automated Unit Testing Framework (zUnit)
  - Introducering van simpelere dependency resolution en impact analyse API's
- Versie 1.1.1
  - DBB Web Applicatie kan geïnstalleerd worden op een Red Hat OpenShift Cotainer Platform (OCP cluster) door middel van een operator
  - Support voor „report only" mode tijdens het uitvoeren van DBB z/OS commando API's
  - DBB source code scanner kan programma's met IBM MQ call statements herkennen
- Versie 1.1.2
  - DBB Web Applicatie User Interface
  - Introducering van de SearchPathDependencyResolver en SearchPathImpactFinder klassen
- Versie 1.1.3
  - Apache Groovy 4.0 upgrade

- Versie 1.1.4
  - JSON Web Token (JWT) eenmalige inlog authenticatie
  - Statisch build report in HTML bestand
  - APAR verbeteringen

(IBM, 2023c)

De versie die gebruikt zal worden voor dit onderzoek is de versie 2.0.0, die maakt deel uit van DBB 2.0x en werd geïntroduceerd in oktober 2022 en kreeg zijn eerste en voorlopig recentste update in mei 2023 (2.0.1). Zoals vermeld zal dit onderzoek gebruik maken van DBB 2.0.0, de belangrijkste feature updates voor DBB 2.0x zijn als volgt:

- Versie 2.0.0 (versie onderzoek)
  - DBB toolkit geïnstalleerd op z/OS UNIX verbind direct met db2 databases
  - DBB toolkit heeft support voor zowel Java 8 als Java 11
  - DBB 2.0 toolkit heeft de Apache Log4J logging technologie vervangen door SLF4J
- Versie 2.0.1
  - Toevoeging van het JobExec commando
  - Toevoeging RACF Group Configuratie

(IBM, 2023b)

Definitie

IBM Dependency Based Build biedt de mogelijkheid aan om traditionele z/OS applicaties die ontwikkeld zijn in programmeer talen zoals Cobol, PL/I en Assembler te builden als onderdeel van een moderne DevOps pipeline. Het biedt een moderne, op scripttaal gebaseerde, automatisatie mogelijkheid dat kan gebruikt worden op z/OS. DBB is gebouwd als een stand-alone product waarvoor geen specifieke source code manager of pipeline automation tool nodig is (IBM, 2023a).

2.2.2. Technische aspecten van DBB

Hoe het werkt

DBB bevat een Java Application Programming Interface, kortweg API, die het mogelijk maakt om taken op z/OS te ondersteunen en afhankelijkheidsinformatie te

creëren en te gebruiken voor de broncode die wordt verwerkt. DBB bestaat uit een z/OS-gebaseerde toolkit die de API's, een afhankelijkheidsscanner en Apache Groovy bevat. Er zijn ook afzonderlijk verkrijgbare componenten, waaronder een webapplicatie die de afhankelijkheidsinformatie en bouwrapporten opslaat en beheert, en een set Apache Groovy-templates om het gebruik van de API's voor het bouwen van applicaties te demonstreren (IBM, 2021c).

Architectuur van DBB

De architectuur van DBB bestaat volgens IBM (2021a) uit acht onderdelen die elk hun eigen taak hebben en absoluut nodig zijn om de DBB build uit te voeren. Elk van deze onderdelen kan vervangen worden door een alternatief product, DBB is niet afhankelijk van eender welk merk of product.

Component 1: MVS bestandssysteem

Het MVS bestandssysteem is nodig omdat het resultaat van de compile/link stap nog altijd PDS gebaseerd is en daardoor is zo'n bestandssysteem onmisbaar.

Component 2: DBB

Vanzelfsprekend is het DBB product nodig om een DBB build uit te voeren, het product komt met een DBB toolkit, bijhorende Groovy build scripts en z/OS build API's die bijvoorbeeld de compile/link in goede banen leidt.

Component 3: DBB WebApp

In de DBB WebApp worden vooral de metadata van afhankelijkheden en de build resultaten bijgehouden in een web server. Deze web server is verbonden met een database, dit kan op eender welke database maar er wordt aangeraden om in productie gebruik te maken van Db2.

Component 4: Rocket Git

Rocket Git is nodig om de communicatie met een Git server te verzorgen en heeft ook de functionaliteit om aan code conversie te doen van ASCII naar EBCDIC.

Component 5: Enterprise Git server

In deze enterprise Git server zal de main repository komen en daarin zitten alle bestanden die nodig zijn om een DBB build uit te kunnen voeren. Dit gaat dan vooral over properties bestanden en source code voor applicatie ontwikkeling taken en om de development pipeline te kunnen uitvoeren. In dit onderzoek zal er gebruikgemaakt worden van Azure Repos als enterprise Git server maar een alternatief is bijvoorbeeld om een eigen Git server op te zetten op een eigen Linux omgeving.

Component 6: Pipeline orchestrator
Een DBB build pipeline heeft ook een pipeline orchestrator nodig, dit is gedistribueerde server gebaseerde software die ervoor zorgt dat de pipeline stappen uitgevoerd kunnen worden. Dit kan gaan over automatische testing, cloning van de repository, DBB build en andere geautomatiseerde taken. Deze orchestrator houdt ook informatie over de builds bij zoals build status en logs. Er wordt in dit onderzoek gebruik gemaakt van Azure Pipelines als pipeline orchestrator maar een alternatief is bijvoorbeeld Jenkins.

Component 7: Pipeline agent
Deze agent is nodig om de pipeline orchestrator te laten runnen en communiceren met z/OS.

Component 8: App dev IDE
Dit is de plaats waarin de applicatie ontwikkelaar zijn aanpassingen aan source code zal aanbrengen. Dit kan elke Git gebaseerde IDE zijn zoals IBM Developer for zOS (IDz) of Visual Studio Code. In dit onderzoek wordt er gebruik gemaakt van beide IDE's.

De architectuur heeft ook een workflow die het doorloopt, IBM (2021a) geeft aan dat in een standaard geval dit neerkomt op negen stappen van begin tot einde. De stappen worden in chronologische volgorde hieronder opgesomd.

De applicatie ontwikkelaar maat wijzigingen aan de source code van een applicatie en maakt gebruik van DBB user build om te verifiëren dat de wijzigingen goed compileren en om eventueel een aantal testen uit te voeren.

De applicatie ontwikkelaar gaat nadien de wijzigingen commiten/pushen naar de enterprise Git server. De Git server ontvangt de wijzigingen nadat een push of merge is gebeurd en gaat dan, automatisch of manueel, het signaal geven aan de pipeline orchestrator dat er een pipeline mag gestart worden.

Deze start op zijn beurt dan een job die wordt doorgegeven aan de agent in z/OS, de communicatie tussen de pipeline orchestrator en de pipeline agent kan gebeuren via SSH of Personal Acces Token (PAT). Dit onderzoek zal gebruik maken van Personal Acces Tokens om de communicatie tussen pipeline agent en pipeline orchestrator te verzorgen.

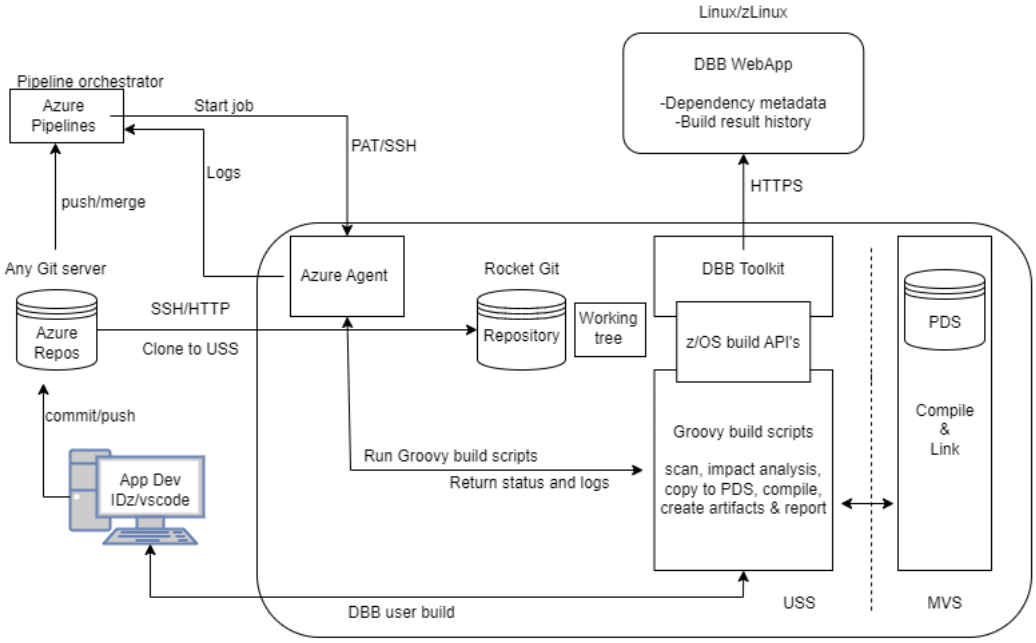
De eerste stap die de agent zal doen nadat die het start job signaal krijgt is om de source code/repository van de Git server te clonen op z/OS naar een working tree repository. Deze working tree repository zal dan gebruikt worden in de volgende

stap door de Groovy build scripts die in werking treden doordat de agent de DBB build start.

Deze build scripts voeren een compile/link uit via de z/OS build API's en zorgen ervoor dat de verkregen of aangepaste load modules op de juiste plek terechtkomen in het MVS bestandssysteem.

Eenmaal dat de build compleet is worden de gegevens zoals de metadata van afhankelijkheden en het resultaat van de build verstuurd naar de DBB Web App die deze gegevens dan opslaat in zijn databank.

Tot slot krijgt ook de agent de logs en resultaten binnen en die stuurt die op zijn beurt dan weer door naar de pipeline orchestrator om ook daar bepaalde logs en gegevens bij te houden over het resultaat. Dit alles is ook visueel terug te vinden in figuur 2.1.



Figuur (2.1)
DBB architectuur van dit onderzoek

2.3. Azure DevOps

2.3.1. Wat is Azure DevOps?

Volgens Microsoft (2024a) ondersteunt Azure DevOps de samenwerking van ontwikkelaars, projectmanagers en medewerkers door een reeks processen aan te bieden die gebruikt worden om software te ontwikkelen. Het stelt organisaties in staat om sneller producten te maken en te verbeteren dan mogelijk is met traditionele softwareontwikkelingsmethoden.

Azure DevOps biedt geïntegreerde functies waartoe je toegang hebt via je webbrowser of IDE-client. Je kunt alle services gebruiken die bij Azure DevOps worden geleverd of alleen die services kiezen die je nodig hebt om je bestaande workflows aan te vullen.

Azure producten

Azure DevOps is een verzameling van vijf individuele services die samen de DevOps ontwikkel filosofie omarmt. Omdat de services onder de Azure DevOps suite zitten is communicatie tussen de services heel snel en heel eenvoudig op te zetten zonder gecompliceerde methodes om de communicatie tot stand te brengen.

Onder de Microsoft (2024a) Azure DevOps suite zitten de volgende services:

- Azure Boards: levert een suite van Agile tools om werk, code defecten en problemen te plannen en op te volgen, gebruikmakend van Kanban en Scrum methodes.
- Azure Repos: voorziet Git repositories of Team Foundation Version Control (TFVC) voor de source control van code.
- Azure Pipelines: voorziet build en release services om Continuous Integration (CI) en Continuous Delivery (CD) van applicaties te ondersteunen.
- Azure Test Plans: set van tools om applicaties te testen, inclusief manueel/exploratieve testing en continuous testing.
- Azure Artifacts: teams kunnen hier gebruik van maken om het delen van packages zoals Maven, npm, NuGet en andere packages van private- of publieke sources te integreren in pipelines.

Voor dit onderzoek zal er enkel gebruik gemaakt worden van de Azure Repos en Azure Pipelines services.

2.3.2. Azure Repos als Source Code Control

In Azure Repos zijn er twee manieren om aan Source Code Control te doen, via Team Foundation Version Control (TFVC) of via Git (gedistribueerd). TFVC is een gecentraliseerd client-server systeem. In zowel Git als TFVC kan er gebruik gemaakt worden van folders, branches en repositories om bestanden te organiseren. Voor dit onderzoek zal er gebruik gemaakt worden van een gedistribueerd Source Code Control (Git) (Microsoft, 2022).

Via Azure Repos kan je alle repositories waartoe je toegang hebt beheren en eventueel aanpassen. Zo kunnen er branches, tags (versies), commits, pushes, merges en pull requests aangemaakt worden. Er kunnen ook handmatig, via Azure Repos, bestanden of folders geüpload worden (Microsoft, 2022).

Azure Repos heeft ook de functionaliteit om automatisch of handmatig applicaties een nieuwe versie te bezorgen na een push. Zo kan er ook aan versiebeheer gedaan worden binnen Azure Repos, er kan ten allen tijden teruggegaan worden naar een vorige versie of commit. Op die manier hoef je niet expliciet meerdere versies bij te houden van eenzelfde applicatie en kan er bij problemen altijd teruggegaan worden naar een vorige werkende versie (Microsoft, 2022).

Een ontwikkelaar die werkt met Azure Repos heeft een eigen kopie van de source repository op zijn computer. De source repository komt inclusief met alle branches en vorige versies van de repository. De ontwikkelaar werkt rechtstreeks op zijn eigen lokale repository en de aanpassingen worden gedeeld naar de centrale repository door een push naar de main source repository. Op die manier kan de ontwikkelaar zelf aan versiebeheer doen door de vorige versies te vergelijken met elkaar zonder netwerk connectie (Microsoft, 2022).

Wanneer de ontwikkelaar een nieuw onderdeel wil toevoegen aan de applicatie kan die ook zijn eigen lokale branch maken om daarop verder te werken zonder andere ontwikkelaars te dwarsbomen. Doordat de branches lichtgewichten zijn op vlak van opslag kan er snel en makkelijk gewisseld worden tussen branches. Indien de ontwikkelaar klaar is met zijn onderdeel kan die door middel van een push/merge de main branch updaten met zijn nieuwe code, zo kan zijn lokale branch opgeruimd worden na de push/merge (Microsoft, 2022).

2.3.3. Azure Pipelines als pipeline orchestrator

Azure Pipelines zorgt ervoor dat het mogelijk is om meerdere taken opeenvolgend te automatiseren om op die manier het builden en deployen van applicaties mak-

kelijker en gestroomlijnder te maken. Met Azure Pipelines kan je taken zoals bestanden aanmaken en scripts of commando's uitvoeren, automatiseren in één of meerdere pipelines. Zo kan er met Azure Pipelines een systeem opgezet worden dat na afloop van een pipeline er opnieuw een pipeline zal gestart worden. Dit kan automatisch of manueel ingesteld worden. Het is ook mogelijk om Azure Pipelines je Git repository in Azure Repos te laten observeren en wanneer die een verandering detecteert in de source code, wordt er automatisch een pipeline gestart. Dit maakt het mogelijk om aan Continuous Integration (CI) te doen en doordat de mogelijkheid bestaat om meerdere pipelines/taken na elkaar uit te voeren kan er ook gedaan worden aan Continuous Delivery (CD). (Microsoft, 2023)

Een ontwikkelaar die werkt met Azure Pipelines hoeft, indien de optie CI aanstaat voor de repository en pipeline, enkel maar een push uit te voeren naar de repository en dan activeert die push de pipeline die daaraan gekoppeld is. De taken van de pipeline worden dan zelfstandig uitgevoerd zonder dat de ontwikkelaar zelf iets moet selecteren of uitvoeren. Er kan ook manueel een pipeline gestart worden, eenmaal gestart werkt die op dezelfde manier als de pipeline die geactiveerd werd door een push naar de gelinkte repository. (Microsoft, 2023)

Standaard is er ondersteuning om controle mechanismen in te schakelen binnenin de pipeline. Zo kan er voordat de pipeline effectief verandering brengt in bijvoorbeeld een productie applicatie eerst een review gevraagd worden aan een of meer leidinggevenden. Pipelines kunnen ook afgeschermd worden van bepaalde personen of groepen om zo confidentialiteit te waarborgen. (Microsoft, 2023)

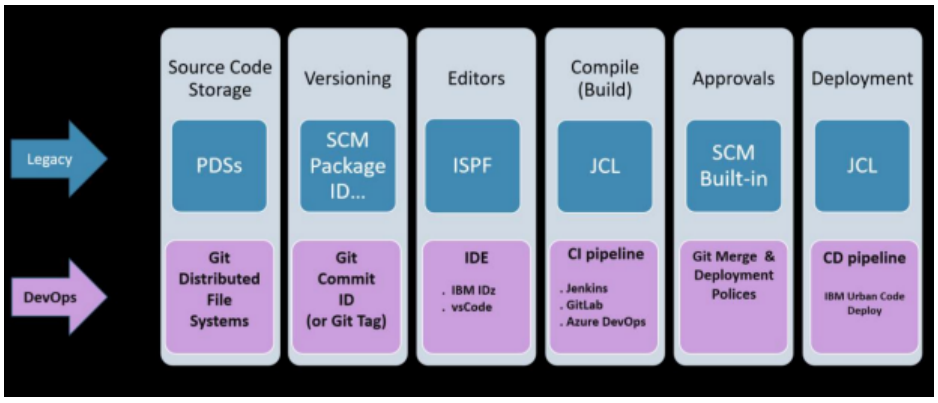
Azure Pipelines hebben ook de mogelijkheid om automatische testing in gang te zetten bij het uitvoeren van de pipeline. Zo kan er naargelang het resultaat van de testen alsnog de wijziging van een applicatie afgebroken worden. Indien de pipeline succesvol beëindigd wordt kan er ook een optie aangevinkt worden om automatische versie controle aan te zetten. Zo kan er bij een applicatie van versie 1.0.1 na een succesvolle uitvoering van de pipeline de versie 1.0.2 meegegeven worden. (Microsoft, 2023)

2.4. Git workflows

2.4.1. Software Development Lifecycle

Een van de belangrijkste aspecten van mainframe software ontwikkeling is dat het betrouwbaar, beschikbaar en bruikbaar is. Deze workflow zit nagenoeg standaard ingebakken in mainframe softwareontwikkeling, maar hoe kan die workflow gerecreëerd of zelfs verbeterd worden door middel van gebruik te maken van Git om de softwareontwikkeling workflow op te stellen. Volgens Nelson Lopez (2023) is een van de beste strategieën om dezelfde eigenschappen van een mainframe workflow te bekomen om de Software Development Lifecycle te recreëren met behulp van Git, een Git ondersteunende IDE, CI pipeline, deployment policies en een CD pipeline.

Op de manier zoals voorgesteld door Lopez (2023) zou de source code storage niet meer in PDS's opgeslagen worden maar in een Git gedistribueerd bestandssysteem, versiebeheer zal niet meer verlopen via een SCM package ID maar via Git commit ID of Git tags. Indien er wijzigingen moeten aangebracht worden in de source code wordt er geen ISPF maar een IDE zoals VS-code of IDz, de compile/link van een programma zal dan weer verzorgd worden door een CI pipeline en niet door een JCL. Toestemmingen en het aanvaarden van wijzigingen aan applicaties zal je dan weer kunnen doen door middel van Git merge en deployment policies in plaats van de SCM. Als laatste zal het uiteindelijk deployen van een programma niet via JCL gebeuren maar via een CD pipeline.



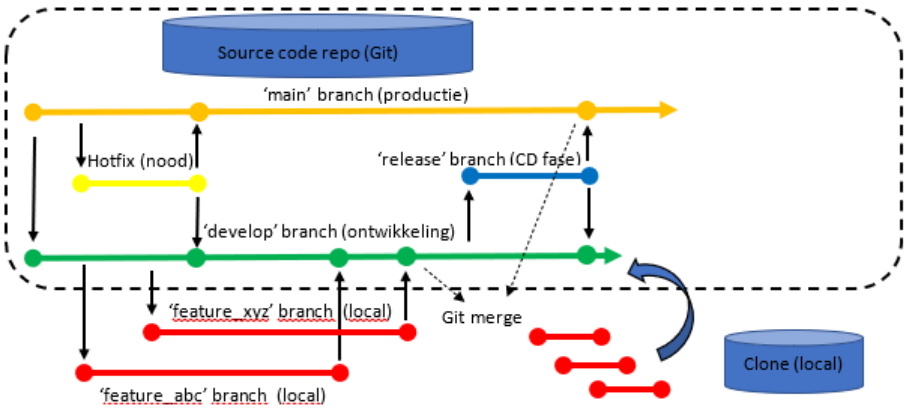
Figuur (2.2)
Software Development Lifecycle: Legacy vs DevOps (Lopez, 2023)

2.4.2. Git branching strategie

Elk bedrijf regelt zijn software ontwikkeling op een manier zodat er een duidelijk onderscheid en doel is voor elke omgeving. Zo is er een productieomgeving, de lopende versie van een applicatie zit daar te draaien. Er is ook een development omgeving waarin nieuwe features of verbeteringen voor die applicatie worden ontwikkeld. Tot slot is er ook nog de QA omgeving (Quality Assesment) deze omgeving wordt gebruikt om applicaties te testen op functioneel vlak vaak tegenover de productie database.

Om een goede branching strategie te bekomen is het belangrijk volgens Lopez (2023) dat tenminste die drie omgevingen worden geïmplementeerd als een branch in de branching strategie. Dit komt neer op een main (productie) branch, een develop (development/ontwikkeling) branch en een release (QA) branch.

Om nog beter te werk te gaan moeten er nog een aantal zaken toegevoegd worden aan de branching strategie zo heeft Nelson Lopez het onder andere over een feature en hotfix/nood branch. Die laatste is voor noodgevallen waarin er heel snel iets veranderd moet worden aan de applicatie die in productie draait. De feature branch zou dan weer een lokale branch worden waarin de ontwikkelaar werkt aan zijn opdracht voor die applicatie. De workflow die wordt beschreven door Nelson Lopez is visueel te vinden in figuur 2.3.



Figuur (2.3)
Git branching strategie beschreven door Nelson Lopez

De weg die een applicatie aflegt binnen de strategie die afgebeeld wordt op 2.3 begint bij het binnenhalen van de applicatie versie die in de ontwikkelomgeving draait. Die is te vinden op de develop branch, vandaar wordt er een clone gemaakt naar een lokaal werkstation en wordt allereerst een nieuwe feature branch aangemaakt. De naamgeving van die branch kan als volgt zijn: feature-array-toevoegen,

dit is afhankelijk van de policy binnen de werkomgeving omtrent naamgeving van branches.

Eenmaal de feature branch aangemaakt is kan de ontwikkelaar aan het werk gaan om de feature te coderen. In de tussentijd kan er een andere ontwikkelaar aan hetzelfde programma werken op nog een andere feature branch. Die ontwikkelaar moet bijvoorbeeld foutmeldingen aanpassen dus zijn feature branch krijgt de naam feature-update-foutmeldingen. Als een ontwikkelaar klaar is met zijn feature dan kan die een merge uitvoeren naar de develop branch om de versie die daar draait te updaten met zijn aangepaste versie. Doordat Git een merge functie heeft gaat Git enkel hetgeen aanpassen dat de ontwikkelaar zelf heeft gewijzigd. Zo kan er tussendoor iemand anders al een merge uitgevoerd hebben naar diezelfde branch zonder dat er conflicten zijn.

Nadat een applicatie in de ontwikkelomgeving een of meerdere updates heeft gekregen kan er gekozen worden om die versie over te zetten naar de productie omgeving. Voordat effectief plaatsvindt zal er eerst een release branch aangemaakt worden om de functionele testen uit te voeren en indien nodig source code aan te passen. Pas als de testen vlot verlopen dan wordt er zowel met de main branch (productie) als met de develop branch een merge uitgevoerd. Zo zijn beide versies up to date met de versie die doorheen de functionele test fase is geraakt (QA).

In geval dat er een situatie is waarin de productie versie niet meer draait dan wordt er een hotfix branch gemaakt vanaf de versie die draait op productie. Source code wordt aangepast en testing wordt opnieuw uitgevoerd zoals bij de release branch. Indien die testen succesvol zijn dan wordt de aangepaste versie zowel met de main branch als met de develop branch gemerged. Hierdoor wordt de fout die in de productie versie zat zowel in de ontwikkelomgeving als in de productieomgeving opgelost.

3

Methodologie

3.1. Voorbereidend werk

Om een zo goed mogelijk onderzoek te kunnen verrichten is het van belang dat er een bepaalde expertise aanwezig is van hetgeen onderzocht wordt. In het geval van deze thesis gaat dit over mainframe, IBM Dependency Based Build (DBB), Azure DevOps, de fundamentele principes van pipelines en werken met Git. Deze expertise is in de vorm van een literatuurstudie ook aangetoond in dit onderzoek, door het opzoekingswerk is er een grotere expertise opgebouwd dan wanneer er blindelings zou gestart worden zonder een goed literatuuronderzoek gevoerd te hebben. De literatuurstudie begint bij het onderwerp mainframe en in dat hoofdstuk wordt er gekeken naar wat een mainframe is, waarvoor het gebruikt wordt en wat de geschiedenis ervan is.

In het stuk over IBM Dependency Based Build is er beschreven wat IBM DBB is, wat het doet en hoe het werkt. Verder is er ook te vinden wat een aantal van de valkuilen kunnen zijn waarmee rekening gehouden moet worden.

Het voorlaatste deel van de literatuurstudie bevat een toelichting en onderzoek van een aantal Git workflows, hierin is er gezocht naar een workflow die zowel flexibel, veilig en robuust is. Allemaal eigenschappen die synoniem staan aan een mainframe en dus moet ook aangetoond worden dat dit nog altijd mogelijk is met een Git workflow.

In het laatste deel van de literatuurstudie is er beschreven wat Azure DevOps is, wat het kan doen en waarvoor het in dit onderzoek is gebruikt. Een aantal onderwerpen die aan bod komen in dit hoofdstuk hebben betrekking op onderdelen van de Azure DevOps suite meer bepaald Azure Repos en Azure Pipelines. Waarvoor

die zaken in het onderzoek gebruikt kunnen worden en wat voor meerwaarde dat geeft aan het onderzoek.

Deze eerste fase die anderhalve week in beslag heeft genomen, kent als resultaat een diepgaande en volledige literatuurstudie die de rode draad is wat betreft informatie in het onderzoek. De literatuurstudie moet antwoord kunnen bieden op de volgende vraag. Hoe kan er een DevOps workflow gerealiseerd worden op een mainframe met behulp van IBM Dependency Based Build?

3.2. Huidig systeem in kaart brengen

Nadat er een zekere expertise is omtrent de theorie van de verschillende factoren in dit onderzoek moet er ook een duidelijk overzicht komen van de huidige situatie binnen de mainframe omgeving van ArcelorMittal Gent. Dit gebeurt aan de hand van een verkenning van de compilatie en bind van programma's binnen het huidige systeem dat werkt met MSP Panapt.

Eenmaal dat de huidige werking in kaart is kan er overgegaan worden naar het werken en experimenteren op een nieuw systeem dat minstens evenveel moet kunnen als het oude systeem. Deze verkenning heeft 1 week geduurd en moet een duidelijk antwoord kunnen geven op de vraag hoe een Cobol programma gecompileerd en gebind wordt binnen de mainframe omgeving van ArcelorMittal Gent.

3.3. Configuratie IBM DBB

Het doel van deze fase is om IBM Dependency Based Build volledig te configureren op de omgeving van ArcelorMittal Gent, zo zijn er onder andere een aantal properties bestanden die aangemaakt en aangevuld zijn en hierbij zijn er ook extra properties toegevoegd aan deze bestanden. Door deze properties aan te passen zal IBM DBB de volledige toegang hebben tot de nodige datasets en bestanden op zowel z/OS als op Unix System Services (USS) om een goede werking van de scripts horende bij de software te kunnen garanderen.

Dit gebeurt aan de hand van de informatie die is gewonnen bij de vorige fase in verband met het in kaart brengen van het huidige systeem. Dit vergde wat opzoekingswerk en heeft een halve week geduurd om de volledige configuratie van IBM DBB te vervolledigen.

3.4. Realiseren Proof Of Concept

Na de configuratie van IBM DBB kan er overgegaan worden naar de grootste fase van het onderzoek namelijk het uitzetten en realiseren van de proof of concept.

Deze fase is een heel belangrijke omwille van het feit dat hier de resultaten worden bekomen om een conclusie te trekken aan het einde van dit onderzoek. De opdracht bestaat erin om de volledige ontwikkeling van een vooraf gekozen Coolgen applicatie van het mainframe te halen als case study, dat wil zeggen dat de source aangemaakt is in een Git ondersteunende IDE zoals bijvoorbeeld Visual Studio Code of IBM Developer for zOS (IDz) en vanuit diezelfde IDE een pipeline gestart wordt om het programma dan te compileren, binden en te plaatsen in de juiste load libraries op z/OS.

Dit gebeurt onder meer via een push vanuit de IDE naar de Azure Repo die dan de aangepaste source code opslaat en daarna een pipeline start om de code te compileren en binden. Het doel hiervan is om de bekomen load module na de bind stap in de juiste load library te plaatsen.

Verder moet er ook bij elke succesvolle aanpassing van het programma een nieuwe versie ontstaan zo zal een programma CGTST met initiële versie 1.0 na een succesvolle aanpassing bijvoorbeeld de versie 1.1 krijgen toegewezen. Zodat er op een duidelijke en overzichtelijke manier wordt bijgehouden wat de huidige versie is van het programma, de vorige versies en eventueel de mogelijkheid om naar een vorige versie terug te gaan.

Na deze fase die zes weken heeft geduurd is er genoeg informatie om een goede en volledige conclusie te trekken op de vraag of het zin heeft om volledig over te schakelen naar een DevOps gebaseerde manier van werken binnen de mainframe omgeving van ArcelorMittal Gent.

3.5. Resultatenanalyse

De volgende fase van dit onderzoek is om de resultaten die erdoorheen zijn bekomen te analyseren. Deze analyse gebeurt in samenspraak met ArcelorMittal en een aantal ontwikkelaars om ook te polsen naar hun mening over het onderzoek en de resultaten ervan. Op het einde van deze fase is er een antwoord op de vraag of de ontwikkelaars dit onderzoek verder willen zetten voor andere zaken of dat er meer overleg nodig is. De fase waarin deze analyse gebeurt heeft 1 week in beslag genomen.

3.6. Conclusies trekken

Nadat de resultaten zijn geanalyseerd is het nodig om na te gaan of het onderzoek daadwerkelijk een succes kan genoemd worden. Dit wordt gedaan aan de hand van een aantal vereisten waaraan moest voldaan worden. Vereisten die zeker moesten voldaan zijn, zijn als volgt.

- Kan het programma gecompileerd worden?
- Kan het programma gebind worden?
- Is de compilatie volgens de juiste compilatie parameters gebeurd?
- Is de bind volgens de juiste bind parameters gebeurd?
- Wordt het resultaat van de compilatie/bind op de juiste plaats en manier opgeslagen?
- Is het proces automatisch of moet er nog ergens manueel ingegrepen worden?

Indien al deze vereisten zijn voldaan kan er gesproken worden van een succesvol onderzoek. Voor het beoordelen van deze vereisten is er een focusgroep gemaakt om het al dan niet behalen van de vereisten te bespreken en meningen uit te wisselen met mensen binnen de mainframe omgeving van ArcelorMittal Gent. Dit heeft opnieuw 1 week in beslag genomen.

3.7. Afwerken scriptie

Doordat er weinig tot geen vertragingen zijn opgelopen was er nog een volledige week over om de scriptie af te werken en alle puntjes op de i te plaatsen. Hierdoor kan er nog eens extra gekeken worden naar eventuele fouten, vergissingen of spelfouten binnen de scriptie. Het spreekt voor zich maar op het einde van al deze fasen is er een afgewerkte scriptie.

4

Werking huidig systeem

4.1. Inleiding

Dit hoofdstuk zal de werking van het huidige systeem om Coolgen applicaties te ontwikkelen van ArcelorMittal Gent toelichten en visualiseren. De stappen die in het huidige systeem aanwezig zijn zullen ook terug te vinden zijn in het nieuwe systeem met behulp van een pipeline en IBM DBB. Het is uiterst noodzakelijk om kennis te hebben van het huidige systeem alvorens een nieuw systeem kan gemaakt/geïmplementeerd worden. Dit komt doordat het systeem dat momenteel gebruikt uitermate veel is gepersonaliseerd door ArcelorMittal Gent om aan hun eisen en gebruiken te voldoen.

4.2. Compileren

De huidige geïnstalleerde versie van de Cobol compiler binnen ArcelorMittal Gent is versie 6.20, deze versie van Cobol is geïntroduceerd in 2017 op 8 september en heeft geen support meer vanaf 30 september 2024.

De JCL die uitgevoerd wordt wanneer men een compilatie van een programma wil uitvoeren wordt opgeroepen via een product van Rocket software namelijk MSP (Manager Products). Dit product zorgt ervoor dat een ontwikkelaar kan meegeven welk programma hij/zij wil compileren en in welke omgeving die dat wil doen, dit kan bijvoorbeeld productie of ontwikkeling zijn. Dat product zorgt er dan voor dat de JCL voor het uitvoeren van de compile wordt gestart met de correcte parameters en de juiste libraries die moeten meegegeven worden tijdens de compilatie zoals SYSLIB en SYSIN DD's.

Heel belangrijk binnen het huidig systeem is het gebruik van metadata, hierdoor weet in dit geval MSP welke parameters die moet aanvoeren aan de compiler, welke

libraries die moet alloceren en of er extra zaken moeten gestart worden zoals een Db2 package bind of Db2 plan bind. Deze metadata is dan ook een van de eerste zaken dat gecontroleerd wordt als men een compilatie wil starten. Indien er geen metadata te vinden is dan zal de compile geannuleerd worden.

Deze metadata bevat onder andere informatie over de afdeling waar het gemaakt is, programmeur van de applicatie, wat voor soort applicatie het is. Zo heb je 3 verschillende soorten programma's binnen ArcelorMittal Gent, je hebt de main programma's, die kunnen ofwel volledig onafhankelijk draaien of ze roepen sub programma's op. Deze main applicaties kunnen zelf niet opgeroepen worden door andere applicaties. Als laatste zijn er ook nog 2 soorten sub programma's, de fsub en de sub. In theorie is er niet veel verschil buiten de manier waarop ze opgeroepen kunnen worden. Zo wordt een sub statisch gebind aan een programma dat hem oproept en een fsub wordt dynamisch opgeroepen tijdens de run time. Naast die 3 soorten van applicaties is er ook nog het feit of er gebruikgemaakt wordt van subsystemen zoals IMS, Db2 of MQ. Indien hiervan gebruikgemaakt wordt dan kan de ontwikkelaar ook deze zaken aanduiden binnen het metadata scherm van zijn/haar applicatie.

Eenmaal alle metadata aanwezig is zal MSP met behulp van skeletons de juiste JCL opmaken om die applicatie te compileren met de juiste libraries en parameters. De compile parameters zijn heel gelijkaardig voor alle soorten programma's maar kunnen toch nog ergens licht afwijken. Zoals wanneer er gebruikgemaakt wordt van een subsysteem zoals IMS of Db2.

4.3. Bind

Er wordt gebruikgemaakt van de IEWL Binder voor z/OS 2.5 om de applicaties van ArcelorMittal te binden. Het belangrijkste verschil tussen een programma dat gebind is met IEWL en een dat gelinkedit is, is het feit dat bij de linkedit een groot uitvoerbaar bestand gemaakt wordt van de verschillende objectbestanden van het hoofdprogramma, de subprogramma's en de subroutines die opgeroepen worden. Hierdoor is het programma dat gelinkedit wordt statisch aangemaakt, dit wil zeggen dat indien er een sub programma verandert er dus een nieuwe linkedit moet gebeuren van alle programma's die dat sub programma gebruiken. Met een binder kan je dynamisch een programma aan een ander programma koppelen/binden, op die manier is het zo dat een sub programma opgeroepen wordt op run time en niet tijdens de compile. Hierdoor hoeft er geen herlink meer te gebeuren van de applicaties die dat programma gebruiken.

Er zijn net zoals bij de compilatie ook parameters die meegegeven worden aan de binder om op de juiste manier de programma's te kunnen binden. In tegenstelling als bij de compilatie moet er niet voor elk soort programma (main, Db2, IMS, sub, ...) een aparte parameter lijst gemaakt worden. De parameters zijn hetzelfde voor main- en fsub programma's aangezien die worden opgeslagen als DLL, voor de sub programma's is er een andere parameter lijst die ervoor zorgt dat de sub geen DLL wordt maar statisch blijft. Hierdoor zullen main- en fsub programma's wel dynamisch opgeroepen kunnen worden tijdens run time en zal er voor sub programma's opnieuw met herlink en hercompile moeten gewerkt worden.

De binder wordt net zoals de compiler opgeroepen door het product MSP, het gaat op dezelfde manier te werk als bij de compile en maakt gebruik van een applicatie zijn metadata om zo de juiste libraries en parameters mee te geven.

5

Proof Of Concept

5.1. Inleiding

Het hoofdstuk van de Proof Of Concept zal gaan over welke stappen en producten er allemaal nodig waren om de proefopstelling succesvol te laten werken. Deze stappen zullen overeenkomen met die van het huidige systeem besproken in het vorige hoofdstuk.

5.2. Installatie & configuratie IBM DBB

De eerste stap in het opzetten van de proefopstelling is om de IBM DBB installatie af te ronden en de configuratie te starten. De installatie op de mainframe van ArcelorMittal werd gedaan door een bedrijf genaamd NRB, zij zijn verantwoordelijk voor installatie van software op de mainframe van ArcelorMittal Gent.

Aangezien IBM DBB al geïnstalleerd is voor deze Proof Of Concept kan er direct begonnen worden aan de configuratie van de bestanden die gebruikt worden door het product DBB. Deze bestanden zijn onder andere properties bestanden die ervoor zorgen dat bepaalde zaken zoals de binder en compiler worden gedeclareerd. Deze zijn niet standaard meegeleverd en moeten afgehaald worden vanaf een git repository namelijk de dbb-zappbuild git repository van IBM.

5.2.1. Installatie dbb-zappbuild

Die repository kan rechtstreeks gecloned worden of via file transfer kunnen die ook op de juiste plaats in de USS belanden. De aanbevolen plaats om die repository te

stockeren is in de hoofdfolder van DBB, dat ziet eruit zoals /dbb/dbb-zappbuild.

5.2.2. Configuratie dbb-zappbuild

Nu dat de dbb-zappbuild repository aanwezig is op de USS kan er begonnen worden aan de configuratie van DBB. Deze configuratie op de USS bestaat hoofdzakelijk uit properties en .groovy bestanden. De properties bestanden zijn te vinden in de folder build-conf en zijn ook de eerste die moeten veranderd of ingevuld worden. Verder zijn er de .groovy bestanden die voor de verwerking van die properties zorgen, dit is in een hoofdprogramma (build.groovy), language specifieke programma's zoals Cobol.groovy en utility programma's zoals BuildUtilities.groovy.

Deze bestanden hebben elk hun eigen taken en nut in de werking van het systeem, zo zijn de properties bestanden noodzakelijk om datasets en PDS'en mee te geven om zo de juiste binder, compiler, load libraries en dergelijke te hebben. De taak van een language specifiek groovy programma zoals Cobol.groovy is om de compilatie en bind van een programma dat in die taal geschreven is uit te voeren. Dit kan heel erg verschillen tussen talen door, zo is de compilatie van een PL/I programma een stuk anders dan de compilatie van een Cobol programma. De zaken die nodig zijn om die compilatie en bind goed uit te voeren kunnen gevonden worden in zo'n language specifiek groovy programma. Deze language specifieke programma's gebruiken heel vaak utility programma's om zaken die hetzelfde zijn in elke taal te bundelen zo is er een BindUtilities.groovy dat een DB2 package bind en/of plan bind kan doen. Dit is onafhankelijk van welke taal je gebruikt en kan dus in zo'n utility programma. Als laatste om de cyclus te vervolledigen is er de build.groovy, dit is het hoofdprogramma dat alle andere programma's op de juiste moment oproept.

Properties bestanden	Language groovy bestanden	Utility groovy bestanden	build.groovy
Dataset en PDS declaratie	Oproepen van utility's en language specifieke werking bepalen	Werking van language onafhankelijke procedures bepalen	Oproepen van de juiste groovy sub programma's

Figuur (5.1)
Soorten bestanden en hun taak

Properties bestanden

Er zijn in totaal 16 properties bestanden waarin aanpassingen moeten aangebracht worden deze zijn onderverdeeld in de verschillende talen en gebruiken. Zo zijn er properties bestanden voor Cobol, PL/I, PSB, ACB en nog een aantal anderen. Er is ook een datasets.properties bestand waarin veel algemene PDS'en en datasets worden gedeclareerd. Zo kan je bijvoorbeeld de z/OS macro library, Cobol compiler, PL/I compiler en DB2 load library vinden en declareren in de datasets.properties.

In de build.properties zitten er vooral algemene instellingen en dus geen datasets, zo kan je aanduiden welke properties bestanden ingeladen kunnen worden als het systeem geactiveerd wordt. Deze zit net zoals de datasets.properties en de overige taal- en gebruiksspecifieke properties bestanden in de build-conf folder die te vinden is binnen de dbb-zappbuild map.

De taal specifieke properties worden aangepast in onder andere Cobol.properties en de PLI.properties bestanden. Hierin worden onder andere instellingen zoals load libraries, vereiste properties en compiler opties meegegeven die algemeen gelden voor alle programma's van die taal. De volledige properties bestanden zijn te vinden in appendix B.

Language groovy bestanden

De language groovy bestanden zorgen ervoor dat de programma's van die taal succesvol gecompileerd en gebind kunnen worden. Naast de compile en bind wordt er ook gezorgd dat de load module van de applicatie op de juiste plaats beland, dat er een DB2 package bind en/of plan bind uitgevoerd wordt indien nodig. Zo wordt ervoor gezorgd dat het programma niet alleen goed wordt gecompileerd en gebind maar ook uitvoerbaar is op het mainframe van ArcelorMittal Gent. Binnen dit onderzoek is het enige language groovy programma de Cobol.groovy, de belangrijkste aangepaste methodes zijn te vinden in appendix C.

Utility groovy bestanden

De utility groovy programma's worden door zowel de language groovy bestanden als de build.groovy opgeroepen om language onafhankelijke procedures op te roepen. Zo is er een groovy utility programma dat ervoor zorgt dat onder andere de build lijst wordt aangemaakt, metadata wordt ingevuld, logicalfiles worden aangemaakt en ervoor zorgt dat datasets kunnen aangemaakt worden. Dit is maar een kleine opsomming van de mogelijkheden van de BuildUtility.groovy, hiernaast zijn er nog 7 andere groovy utility programma's. De enige programma's die aangepast

zijn in het kader van het onderzoek zijn BuildUtilities.groovy en BindUtilities.groovy, de belangrijkste aangepaste methodes zijn te vinden in appendix D.

build.groovy

Dit groovy programma roept andere programma's op zoals de utility programma's en de language specifieke programma's. Er wordt een build lijst gemaakt op basis van de rangschikking die meegegeven wordt binnen de properties van de build. Er worden dus groovy bestanden aangeroepen en de properties worden ingeladen zodat die gebruikt kunnen worden door de build.groovy en de andere groovy programma's.

Hieronder een opsomming van alle bestanden en programma's die ervoor zorgen dat DBB de programma's en bestanden kan compileren, binden en opslaan in een z/OS omgeving.

- **Properties bestanden**
  - ACBgen.properties
  - Assembler.properties
  - build.properties
  - Cobol.properties
  - datasets.properties
  - DBDgen.properties
  - defaultzAppBuildConf.properties
  - dependencyReport.properties
  - LinkEdit.properties
  - MFS.properties
  - PLI.properties
  - PSBgen.properties
  - Transfer.properties
  - ZunitConfig.properties
- **Language groovy bestanden**
  - Assembler.groovy
  - Cobol.groovy
  - DBDgen.groovy
  - LinkEdit.groovy
  - MFS.groovy
  - PLI.groovy
  - PSBgen.groovy
  - Transfer.groovy
  - ZunitConfig.groovy
- **Utility groovy bestanden**
  - BindUtilities.groovy
  - BuildUtilities.groovy
  - BuildReportUtilities.groovy
  - DependencyScannerUtilities.groovy
  - FilePropUtilities.groovy
  - GitUtilities.groovy
  - ImpactUtilities.groovy
  - ReportingUtilities.groovy
- **build.groovy**

5.2.3. Configuratie DBB buiten USS

Naast de configuratie van DBB in de zappbuild op USS is er ook een configuratie die in elke repository meegegeven moet worden bij het starten van DBB. Dat zijn properties bestanden die in de repository meegeleverd moeten worden om de compilatie en bind in goed uit te voeren. Deze properties bestanden hebben altijd een application.properties en een file.properties, hierin wordt er onder meer een build lijst rangschikking meegegeven om te beslissen in welke orde programma's gecompileerd zullen worden. Naast de application- en file.properties moet er ook per taal een properties bestand meegegeven worden met extra instellingen die toegevoegd worden aan de al beschikbare instellingen die in de properties op USS zijn gedeclareerd. Zo moet er bij de verwerking van een Cobol programma de file-, application- en Cobol.properties bestanden aanwezig zijn binnen de repository.

Er zijn in totaal voor ArcelorMittal 10 properties bestanden die meegegeven kunnen worden binnen een repository. Deze worden hieronder opgesomd.

- application.properties
- Assembler.properties
- Cobol.properties
- file.properties
- languageConfigurationMapping
- LinkEdit.properties
- PLI.properties
- reports.properties
- Transfer.properties
- ZunitConfig.properties

Naast de properties bestanden is er nog een .gitattributes bestand dat heel belangrijk is in verband met het omzetten van de encodering van z/OS naar git en omgekeerd. Op de mainframe van ArcelorMittal Gent wordt er gebruikgemaakt van de EBCIDIC IBM 1148 encodering en die is niet standaard ondersteund door git. Daardoor moet er in een .gitattributes bestand aangegeven worden dat de input in die EBCIDIC encodering staat en dus nog moet omgezet worden naar UTF-8. De properties bestanden en .gitattributes die gebruikt worden binnen de repository van dit onderzoek zijn volledig aangepast terug te vinden in appendix E.

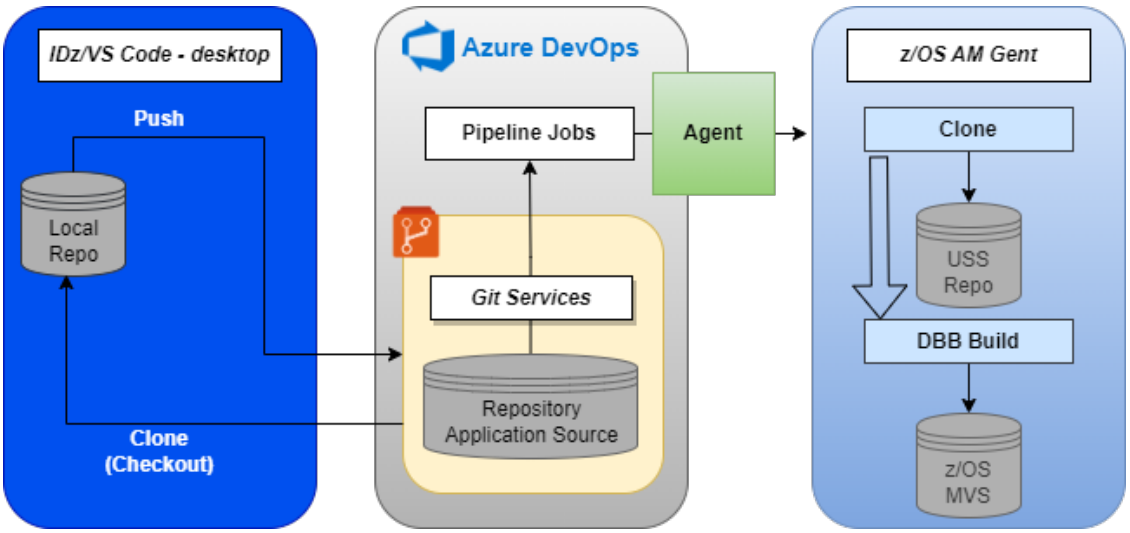
5.3. Opzetten pipeline Azure DevOps

Om de compilatie en bind van programma's te voltooien wordt er gebruikgemaakt van pipelines en pipeline software om dat proces te starten. In dit onderzoek zal er gebruikgemaakt worden van Azure Pipelines, een product uit de Azure DevOps suite. De reden dat er gebruikgemaakt wordt van Azure Pipelines en niet van een andere pipeline software zoals Jenkins is door het feit dat ArcelorMittal Gent al een licentie had voor Azure DevOps. Zo wordt de software stack van ArcelorMittal niet onnodig uitgebreid.

5.3.1. Azure DevOps workflow

Voordat er gestart kan worden moet er bekeken worden welke workflow er gebruikt zal worden binnen de pipeline en Azure DevOps. De workflow die gebruikt wordt in dit onderzoek is heel standaard en zal gaan van een desktop met IDz of Visual Studio Code naar het eindstation z/OS. Dit gaat als volgt te werk, er wordt begonnen vanaf niks dus er is nog geen enkele repository die op de desktop staat.

De eerste stap is dus om een repository te clonen naar een lokale map, dit kan rechtstreeks vanuit IDz of Visual Studio Code. Dan wordt die repository lokaal aangepast door bijvoorbeeld een nieuwe branch toe te voegen, nieuwe commits of nieuwe code. Als de repository klaar is om doorgestuurd te worden naar z/OS dan wordt er via een "git push" een signaal verstuurd naar de centrale repository en die zal dan op zijn beurt een of meerdere pipeline jobs lanceren die uitgevoerd worden door een agent die ofwel op Linux of Windows draait. Die pipeline job zal de repository gaan clonen op z/OS, daarna wordt die repository met behulp van IBM DBB gebuild en de bekomen load modules en source files worden daarna op z/OS opgeslagen. De visuele weergave van deze workflow is te vinden op figuur 5.2.



Figuur (5.2)
Azure DevOps workflow

5.3.2. Opzetten van een organisatie, agent pool en agent in Azure DevOps

Er wordt gewerkt binnen Azure DevOps met organisaties zo kan er op een makkelijke en overzichtelijke manier gescheiden gewerkt worden van andere takken en/of teams binnen een bedrijf. Voordat er kan begonnen worden aan het proces om een organisatie en agent op te zetten moet er toegang zijn tot Azure DevOps. Dit wil zeggen dat er een geregistreerd account ter beschikking moet zijn met de nodige privileges om een organisatie te mogen aanmaken. Indien dat ter beschikking is zouden er geen problemen mogen zijn met het opzetten van een organisatie en agent.

1. Inloggen op Azure DevOps.
2. Aanmaken van een full acces **P**ersonal **A**cces **T**okens.
3. Creëer een nieuwe organisatie.
4. Toevoegen van een self hosted agent pool.
5. Maken van een nieuwe agent op een Windows machine.
6. Configureren van de Windows agent.
7. Starten van de agent.

(Microsoft, 2024d) (Microsoft, 2024c) (Microsoft, 2024b)

Het nut van een agent pool en agents is om de pipelines uit te laten voeren. Elke agent kan 1 pipeline uitvoeren. Indien er twee pipelines zijn die uitgevoerd moeten worden, zal er een van de twee in een queue geplaatst worden totdat de agent terug beschikbaar is. Binnen een agent pool kunnen er meerdere agents geïnstalleerd zijn. Hierdoor is het mogelijk om meerdere pipelines tegelijk te laten uitvoeren binnen dezelfde agent pool.

Het is mogelijk om restricties te plaatsen op agents of agent pools in verband met welke pipelines ze mogen/kunnen uitvoeren. Zo kan er een eigen agent pool zijn voor bijvoorbeeld een migratie pipeline en een voor de compilatie/bind van programma's.

5.3.3. Azure DevOps project en connectie met de mainframe

Binnen een organisaties kunnen er meerdere projecten aangemaakt worden, zo kan er op een overzichtelijke manier gewerkt worden binnen Azure DevOps. Elk project heeft zijn eigen resources, zo kunnen er bijvoorbeeld meerdere Azure Repo's en Azure Pipelines aangemaakt worden per project.

Naast de resources die al aan bod gekomen zijn is er ook de mogelijkheid om service connecties te leggen, in deze proof of concept werd dat gedaan naar de mainframe van ArcelorMittal Gent. Dit gebeurt binnen de project instellingen en dan service connections, op die manier kan er een SSH connectie gemaakt worden met een mainframe of bij uitbreiding een andere server/computer waarmee verbinding wenst gemaakt te worden.

In het geval van dit onderzoek wordt er gebruikgemaakt van één project waarin geconnecteerd wordt naar de mainframe van ArcelorMittal Gent met behulp van de SSH service connectie. Voor de SSH verbinding moeten de juiste waarden ingevuld worden zoals host name en poort nummer. De authenticatie gebeurt via een RACF user en paswoord, zo is bij het onderzoek gebruikgemaakt van een persoonlijke user. Indien het paswoord van die user verandert zal ook de service connectie moeten aangepast worden naar het nieuwe paswoord.

5.3.4. Azure Repo toewijzen aan een project

Voordat er kan gewerkt worden op een remote repository moet er eerst een aangemaakt worden, dit gebeurt via het online portaal van Azure DevOps binnen het project waarin de repo aangemaakt dient te worden.

Er kan gekozen worden om een nieuwe repo aan te maken of om te starten van een bestaande repository via een import, dit onderzoek zal starten vanaf een lege repository.

5.3.5. Azure Pipeline toewijzen aan een project

De verbinding van de repository naar de mainframe wordt behandeld door een pipeline. Dit kan letterlijk gezien worden zoals de naam het aangeeft, het is een pijplijn waardoor allerhande informatie en bestanden wordt doorgegeven aan de repository, de user en de mainframe.

Een pipeline wordt aangemaakt binnen een project en werkt door middel van bash scripts, de bash scripts die nodig zijn voor het werken met IBM DBB worden aangeboden door IBM in een document genaamd „Azure DevOps and IBM Dependency Based Build Integration”, hierin staan de voorgestelde scripts in vermeld.

Het is aangeraden om te beginnen met een basisscript dat bijvoorbeeld de mappenstructuur toont of de aangemelde user alvorens te springen op de DBB scripts. De scripts van IBM werden aangepast naar de eisen van ArcelorMittal en wijken dus licht af ten opzichte van de originele scripts. Zowel de aangepaste als de originele scripts zijn te vinden in appendix F.

5.3.6. Testen werking pipeline

Om er zeker van te zijn dat er over kan gegaan worden naar het gebruik van de IBM DBB bash scripts moet er zekerheid zijn dat de pipeline de goeie connectie maakt naar de mainframe en moet er ook gecloned kunnen worden binnen de USS. Met andere woorden Git moet daar goed geïnstalleerd zijn. Deze zaken kunnen makkelijk getest worden, het eerste wat getest is in dit onderzoek is of er wel degelijk een connectie was. Dat werd getest door een pipeline met het commando „whoami", als dat de juiste user weergaf zoals opgegeven bij het aanmaken van de service connectie dan was er geen probleem. Indien die niet werd weergegeven dan is er iets fout gelopen bij de service connectie en moet die stap opnieuw gedaan/bekeken worden.

Als er vastgesteld is dat er een connectie kan gemaakt worden met de mainframe is er in dit onderzoek gekeken naar Git op de USS. Daarvoor zijn er twee stappen namelijk zorgen dat Git geïnstalleerd is op de user die gebruikt wordt voor de service connectie en zorgen dat die user ook een Azure Repo kan clonen.

Om te zien of Git geïnstalleerd is op de user zijn USS wordt het „git –version" commando gebruikt. Als dat een versie van git weergeeft wil dat zeggen dat Git geïnstalleerd is op die user zijn USS, indien er een boodschap weergegeven wordt dat het commando niet gevonden werd, dan is Git nog niet (goed) geïnstalleerd.

Nadat vastgesteld is dat Git goed is geïnstalleerd kan de configuratie voor het clonen beginnen. De clone zal gebeuren aan de hand van een Personal Acces Token (PAT), met PAT kan er connectie gemaakt worden via HTTPS en niet via SSH. Eerste stap in dit proces is om een PAT aan te maken, dit kan via het Azure Repos dashboard door op de knop clone te klikken en dan git credentials aan te laten maken. Het paswoord dat gecreëerd wordt is uitermate belangrijk want deze wordt nergens bijgehouden door Azure DevOps om later terug op te vragen. Dat paswoord moet dan enkel nog omgezet worden naar een Base 64 encoding, dit kan via Powershell op Windows via de volgende commando's:

1. $MyPat='gekopieerd paswoord'
2. $B64Pat=[Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes("':$MyPat"))
3. echo $B64Pat

Na deze commando's verkrijg je de Base 64 geëncodeerde versie van het paswoord en diegene die gebruikt dient te worden door Git op het USS. Om de clone succesvol uit te voeren is het nodig dat het Base 64 geëncodeerd paswoord wordt opgeslagen in de git config. Dit kan ingesteld worden door het „git config –global http.extraheader "Authorization: Basic GEKOPIEERD B64 PASWOORD"" commando

in te geven binnen een terminal zoals dat kan op IDz. Hierna kan het git clone commando uitgevoerd worden zonder problemen.

Als beide testen vlekkeloos verlopen zijn dan pas kan er overgegaan worden op de IBM DBB bash scripts.

5.3.7. Pipeline taken, variabelen, opties & triggers

Het gebruik van de IBM DBB bash scripts kan enkel en alleen als er ook een aantal variabelen gedefinieerd zijn. Zo moet er een nieuwe SSH taak toegevoegd worden aan de pipeline met daarin het commando om het bash script te starten en daarna de script parameters. In het kader van dit onderzoek ziet het commando voor het AzRocketGit-init.sh script eruit als „./AzRocketGit-init.sh $(MyRepo) $(MyWorkDir) $(Build.SourceBranch)" en het AzDBB-build.sh script ziet eruit als „./AzDBB-build.sh $(MyWorkDir) $(MyRepo) $(MyApp) –impactBuild".

De taken gebruiken variabelen, op die manier kan een pipeline bij meerdere repositories ingezet worden. Azure Pipeline komt inclusief een hele hoop standaard variabelen maar een aantal zijn zelf moeten aangemaakt worden voor dit onderzoek. Diegene die rechtstreeks gebruikt worden in de taken zijn de volgende.

Naam	Waarde	Uitleg
git	Pad naar Rocket Git client op USS.	Zorgt ervoor dat de Git client gevonden wordt door de pipeline
MyApp	source	De plaats binnen de repository waar het script de source code kan vinden.
MyRepo	$(Build.Repository.Uri)	De link van de repository.
MyWorkDir	$HOME/Azure-WorkSpace/ $(System.TeamProject)-$(Build.buildid)	De plaats waar de repository gekloond wordt en de logs aangemaakt.
system.teamProject	Projectnaam	De projectnaam waarin de pipeline is gemaakt.

Figuur (5.3)
Pipeline variabelen en hun taak

De opties zijn voor dit onderzoek op de standaardwaarden gehouden, bij de pipe-

line opties zit onder andere hoelang een pipeline maximaal mag duren en indien die gecanceld wordt hoelang die maximum mag nemen om de job te cancelen.

Als laatste zijn er nog triggers die kunnen ingesteld worden voor de pipeline. Hiermee kan continuous integration geactiveerd worden voor bepaald branches of net gedeactiveerd. Zo kan er voor de branch develop continuous integration geactiveerd zijn en voor de main branch niet.

Na al deze zaken overlopen te hebben en waar nodig aangepast te hebben is de pipeline klaar voor gebruik en kan er overgegaan worden op het aanpassen van de groovy programma's.

5.4. Aanpassen nieuw systeem voor Cobol compilatie & bind

De groovy programma's zijn als het ware het brein achter de hele compile, bind en alles eromheen. Ze maken achterliggend gebruik van Java klassen op die manier is het een soort API om via groovy met het mainframe systeem te communiceren. In dit onderzoek is er gebruikgemaakt van de Cobol.groovy, BuildUtilities.groovy en de BindUtilities.groovy programma's, deze zijn in de loop van het onderzoek volledig aangepast aan het systeem van ArcelorMittal Gent. De belangrijkste aangepaste stukken code zijn te vinden in appendix C en appendix D. Er is ook gebruikgemaakt van het standaard build.groovy programma, deze had voor dit onderzoek geen aanpassingen nodig.

5.4.1. Cobol.groovy aanpassen

De grootste aanpassingen die aangebracht moesten worden aan het Cobol.groovy script was inzake het gebruik van de gepersonaliseerde properties om de subsystemen die gebruikt worden te bepalen aan de hand van de metadata die ingevuld wordt door de programmeur. Deze moeten kloppen met de manier waarop subsystemen zoals IMS of DB2 gebruik worden gecompileerd en gebind. Verder is er nog de metadata die gaat over het soort programma, een main, sub of fsub programma. Als een programma een fsub is moet er een DLL aangemaakt worden en moet deze op de juiste plaats in het systeem opgeslagen worden.

Binnen ArcelorMittal Gent worden fsub's opgeslagen op 2 manieren, ze worden opgeslagen in een RRS library en een IMS library. Dit was vroeger omdat er met een andere module moest gebind worden. Tegenwoordig kan er gebruikgemaakt worden van de module DSNULI binnen de SYSLIB concatenatie om zo de juiste zaken mee te nemen. Er moet bij fsub programma's ook altijd de DB2 load library mee gebind worden, deze hoeven wel geen plan- en package bind uit te voeren,

tenzij deze ook expliciet DB2 gebruiken.

De standaard Cobol.groovy is terug te vinden op de Github van IBM via de volgende link: IBM DBB zappbuild Github, hierop zijn alle bestanden die nodig zijn terug te vinden in hun standaard vorm. Aangezien deze bestanden aangepast zijn in het kader van dit onderzoek worden de bestanden meegegeven en verschillen ze in filosofie niet van elkaar maar de uitvoering van de scripts en de inhoud ervan verschilt wel duidelijk. Het standaard Cobol.groovy is amper 405 lijntjes code terwijl het aangepaste Cobol.groovy programma van dit onderzoek 765 lijntjes code bevat. Dat komt uit op bijna 90% extra lijnen code om de aanpassingen voor het systeem van ArcelorMittal Gent toe te passen.

5.4.2. BuildUtilities.groovy aanpassen

In de BuildUtilities.groovy worden er hoofdzakelijk processen gedefinieerd die de hoofdprogramma's, zoals Cobol.groovy, helpen. Dit kan door bijvoorbeeld een buildlist te maken, een LogicalFile aan te maken, variabelen opvullen, ...

De grootste veranderingen die er voor dit onderzoek zijn doorgevoerd zijn die in verband met de LogicalFile properties. Daarin wordt er gebruikgemaakt van de metadata dat een ontwikkelaar meegeeft via de properties bestanden. Aan de hand van die properties zal de LogicalFile aangemaakt en ingevuld worden.

Nog een verschil met het standaard BuildUtilities.groovy programma, dat te vinden is op dezelfde Github repository als eerder vermeld in 5.4.1. Is het maken van de buildlist, deze werd voorheen enkel maar gesorteerd op programma taal maar in dit onderzoek wordt er gebruikgemaakt van de ArcelorMittal filosofie en daarin staat dat ze niet enkel op taal moeten gesorteerd worden maar ook op type. Zo is de volgorde zo dat de sub programma's eerst moeten verwerkt worden en daarna volgen de fsub's en pas op het einde worden de main programma's verwerkt. Dit is zodat indien er een sub programma gebruikt wordt binnen een fsub of main dat deze direct de goeie versie meekrijgt. Zo zou er in het geval dat een main voor een sub wordt verwerkt de kans bestaan dat die main het sub programma dat aangepast werd, gebruikt en daardoor met de verkeerde versie van dat sub programma wordt verwerkt.

Er is veel aangepast in de BuildUtilities.groovy maar niet veel toegevoegd dit is te zien aan het aantal lijntjes dat gecodeerd is. In de aangepaste versie zitten er 1036 lijntjes code terwijl dat in de standaard versie 983 lijntjes zijn. Dit komt neer op een toename van amper 5%, veel minder dan als de toename die er was bij het Cobol.groovy programma.

5.4.3. BindUtilities.groovy aanpassen

Net zoals de BuildUtilities.groovy is de BindUtilities.groovy er vooral om het hoofdprogramma te sparen van ellenlange code die hergebruikt kan worden voor vele scripts. De inhoud van de BindUtilities.groovy gaat zoals de naam als aangeeft gaan over de processen die moeten gebeuren om in DB2 een bind uit te voeren.

De standaard versie van dit programma had als enige functie een die een package bind uitvoerde op DB2. Dit is binnen ArcelorMittal Gent niet genoeg, daar moet namelijk ook een plan bind uitgevoerd worden voor elk programma en er wordt gebruikgemaakt van een bepaald versienummer dat meegegeven wordt die op precies de juiste manier moet zijn opgesteld om compatibel te zijn met het systeem van ArcelorMittal Gent.

Er zijn heel wat zaken moeten aangepast worden, zo wordt er in plaats van gebruik te maken van de TSOExec klasse geopteerd om de commando's die nodig zijn om zowel een package bind als plan bind uit te voeren. Op te slaan in een MVS bestand dat dan later via een JCL gelanceerd wordt met de JCLExec klasse. Dit omdat de TSOExec niet werkte binnen de huidige versie van DBB. Net zoals de andere groovy code is ook deze code te vinden in de appendices, meer bepaald in appendix D.

Er is in vergelijking met de ander groovy programma's het meeste aangepast in de BindUtilities.groovy maar doordat deze heel efficiënt zijn gemaakt en een aantal zaken overbodig waren in de standaard versie is er in plaats van een toename, een afname in lijntjes code. Zo is er in de aangepaste versie amper 91 lijntjes code nodig om de logica uit te laten voeren en is dat in de standaard versie 111 lijntjes. Dat is een afname van 18% ten opzichte van de standaard versie.

5.4.4. Properties bestanden aanpassen (repo kant)

Binnen ArcelorMittal Gent is de metadata van een programma heel belangrijk en essentieel voor de goede werking van het systeem. Om die reden moest er dus een oplossing zijn om ook via de pipeline werking aan die metadata te komen.

De meta data wordt in het huidige systeem handmatig door de ontwikkelaar ingevuld en heeft als nut dat er voor bijvoorbeeld problemen bij het juiste team wordt aangeklopt, op dezelfde manier worden ook subsystemen meegeven bij de compilatie en bind. Enkel de subsystemen die gedefinieerd zijn binnen de metadata zullen worden meegenomen.

Om dit systeem na te bootsen wordt er gebruikgemaakt van properties bestanden, zo is er een file.properties die een overkoepelende rol heeft binnen de repository en waarin dus gekozen is om die nieuwe properties ook te definiëren. De property

om bijvoorbeeld aan te geven dat een programma een fsub is, ziet er als volgt uit: isFSUB = true :: **/tstdb2.cbl. isFSUB = true geeft aan dat het gaat over een fsub en de **/tstdb2.cbl geeft aan dat het gaat over het programma tstdb2.cbl dat in elke map mag zitten. Stel dat het programma enkel maar een fsub mag zijn als het in de map fsub zit dan kan de volgende lijn gebruikt worden als metadata: isFSUB = true :: **/fsub/tstdb2.cbl. Als er nog een stap verder moet gegaan worden en alle Cobol programma's binnen de map fsub tot de categorie fsub horen dan kan de volgende lijn meegegeven worden: isFSUB = true :: **/fsub/*.cbl.

Deze manier geeft heel erg veel mogelijkheden om wildcards te gebruiken om op die manier zo weinig mogelijk te moeten schrijven/typen. In principe kunnen er policies gemaakt worden dat alle programma's die in een bepaalde map zitten of met een bepaalde letter combinatie beginnen tot een categorie behoren. Dit is buiten de scope van het onderzoek maar toch ook niet onbelangrijk voor eventuele toekomstplannen.

6

Conclusie

In deze bachelorproef wou men onderzoeken op welke manier er een Azure DevOps pipeline geïntegreerd kon worden om de compile/bind van bestaande en toekomstige Coolgen programma's uit te voeren. Men wou ook onderzoeken wat de gevolgen zijn voor het huidige compile/bind systeem voor Coolgen programma's na het integreren van de bekomen pipeline met de mainframe omgeving van ArcelorMittal Gent. Dit is onderzocht aan de hand van een proof of concept waarin er geëxperimenteerd is met een aantal manieren om een Azure DevOps pipeline te integreren binnen de mainframe omgeving van ArcelorMittal Gent.

Uit het onderzoek is gebleken dat zo'n pipeline met Azure DevOps opzetten niet iets is dat even snel kan gebeuren. Als er moet gezorgd worden dat deze pipeline naadloos werkt en geïntegreerd is met de mainframe van ArcelorMittal Gent dan moet er heel wat voorbereiding gebeuren en moeten er ook heel wat veiligheid ingebouwd zijn. Zo moet er als voorbereiding een enorm goede kennis zijn van de werking van het huidige systeem (zonder pipeline) en de processen die dienen te gebeuren alvorens een programma kan/mag gecompileerd of gebind worden. Er moet verder ook een goede kennis zijn van de verschillende load libraries en van de werking van Git en het omgaan met Git repositories en bij voorkeur is er ook een voorkennis van een moderne programmeertaal zoals bijvoorbeeld Java, .NET of python. Dit laatste is vooral om de code, concepten en ideeën van de build scripts te begrijpen en aan te passen.

Het onderzoek toonde dus aan dat er heel wat voorbereidend werk nodig is om nog maar te beginnen aan deze opdracht maar dat was niet alles. Het onderzoek toonde ook dat zo'n pipeline heel goed en consistent kan draaien zo kan er, mits de nodige aanpassingen aangebracht zijn, naadloos samengewerkt worden met het huidige systeem. Dit zorgt ervoor dat het pipeline gebaseerde systeem een goede

kandidaat is om in de toekomst in aanmerking te komen om een vaste waarde te worden in de software release management van ArcelorMittal Gent.

Doordat dit onderzoek volledig is aangepast naar de mainframe omgeving van ArcelorMittal zouden andere bedrijven of personen die dit systeem willen gebruiken weinig nut hebben aan de geschreven code binnen het onderzoek. Dit doordat de mainframe omgeving van ArcelorMittal Gent en bij uitbreiding zo goed als elk bedrijf dat een mainframe onderhoud/heeft volledig anders is/kan zijn, door de vele aanpassingen en zelfgeschreven procedures binnen hun mainframe omgeving. Desondanks is dit onderzoek niet onbruikbaar voor andere geïnteresseerden, de concepten/ideeën die gebruikt zijn in dit onderzoek zijn van goudwaarde voor bedrijven die een soortgelijk project willen starten. Met dit onderzoek zouden er heel wat fouten/problemen vermeden kunnen worden bij een soortgelijk onderzoek/project. De grootste meerwaarde van dit onderzoek is dus voor ArcelorMittal Gent die ook vragende partij waren om dit onderzoek te starten.

De manier waarop dit allemaal werkt is met behulp van een aantal producten namelijk Azure DevOps (Pipeline, Repos, ...), IBM Dependency Based Build (DBB), (Rocket)-Git, Microsoft Visual Studio Code, IBM Developer for z/OS (IDz) en IBM DBB zAppbuild. Deze producten zijn samen de ruggengraat van dit onderzoek en die zou niet gelukt zijn zonder de opgesomde softwareproducten. Er kunnen zeker alternatieven gebruikt worden voor bepaalde software, zo kan er bijvoorbeeld een eigen Git server opgesteld worden in samenwerking met Jenkins om zo Azure DevOps te vervangen. Dit was niet de bedoeling van het onderzoek, het onderzoek moest kunnen aantonen dat het mogelijk was om een werkend systeem te krijgen met behulp van een pipeline software. In het geval van dit onderzoek was dat Azure DevOps om de simpele reden dat dit al aangekocht was voordien.

In de toekomst kan het pipeline gebaseerd systeem op termijn het huidige systeem vervangen maar door de vele aanpassingen binnen de mainframe omgeving van ArcelorMittal Gent moet dit nog zorgvuldig onderzocht worden. Het feit dat zo'n systeem met IBM DBB en pipeline software kan is doordat IBM DBB heel erg aangepast kan worden, hierdoor is de kans dat dit systeem weleens de norm wordt binnen ArcelorMittal Gent reëel.

A

Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

Samenvatting

Dit onderzoek zal gaan over het uitwerken van een werkende Azure DevOps pipeline om Coolgen programma's te compileren en binden met behulp van IBM Dependency Based Build (DBB) en zijn ingebouwd framework. Er werd gezocht naar een oplossing om de mainframe omgeving van ArcelorMittal Gent te moderniseren om zo aantrekkelijker te zijn voor afgestudeerden en om gebruik te maken van de nieuwere technologieën binnen het mainframe landschap. In de proof of concept is er een proefopstelling opgezet zodat de Coolgen applicaties kunnen worden gecompileerd en gebind via een Azure pipeline. Op die manier hoeft de ontwikkelaar niet zelf de compilatie en/of bind te starten. Het verwachte resultaat is dat de pipeline een correcte compilatie en bind kan uitvoeren zonder dat de ontwikkelaar zelf iets moet uitvoeren op de mainframe omgeving en dat het versiebeheer volledig kan beheerd worden door Azure DevOps. Zo zal er een einde komen aan de vele stappen die nodig zijn om een compilatie en bind van een Coolgen programma uit te voeren ook zal er voortaan in een Gitondersteunende IDE gewerkt kunnen worden.

A.1. Introductie

Mainframe is een van de oudste en meest gebruikte computertechnologieën ooit, maar na al die jaren dat er getwijfeld werd over het wel of niet afschaffen van de technologie. Is er een groot tekort aan pas afgestudeerden ontstaan, die verse krachten zouden het voortouw kunnen nemen in de vernieuwing van de main-

frame. Volgens Broadcom (2024) is de mainframe meer dan ooit toegankelijk en compatibel in projecten die zich niet enkel op het mainframe platform situeren. Dit komt volgens hun door de mogelijkheid om de DevOps workflow toe te passen waardoor de flexibiliteit en automatisatie mogelijkheden toeneemt. Verder maakt het ook de samenwerking met teamleden, zowel binnen een mainframe team als er buiten, makkelijker en meer gestroomlijnd. Hierdoor is IBM Dependency Based Build een goede manier om een mainframe te moderniseren omdat deze tool van IBM ervoor zorgt dat je op een DevOps manier kan werken met een mainframe. Deze workflow bestaat uit een pipeline en een repository, in het geval van dit onderzoek zal de pipeline(s) verzorgd worden door Azure Pipelines en de repositories zullen aangeboden worden via Azure Repos. Beide services maken deel uit van het Azure DevOps pakket dat nog heel vaak zal vermeld worden doorheen het onderzoek en de proof of concept.

De leveranciers van mainframe software hebben het belang van modernisering in het landschap opgemerkt en zijn dus al enkele jaren volop tijd en geld aan het pompen in nieuwe gebruiksvriendelijke en meer hedendaagse tools. Die tools zouden het werken op zo'n omgeving vereenvoudigen voor zowel gebruikers, ontwikkelaars en administrators.

Dit onderzoek werd in samenspraak met het mainframe team van ArcelorMittal Gent opgestart om een efficiëntere manier te vinden om de compilatie en bind uit te voeren van een Coolgen programma. Het onderzoek zal vooral impact hebben bij de mainframe systeem administrators en bij uitbreiding ook de ontwikkelaars van Coolgen programma's in ArcelorMittal Gent.

Het doel van dit onderzoek is dan ook om een Coolgen programma, gemaakt met een moderne Git ondersteunende Integrated Development Environment, kortweg IDE, zoals bijvoorbeeld Visual Studio Code. Volledig automatisch te kunnen compileren en binden op het moment dat er wijzigingen opgeslagen worden naar de Azure Repo, dit zal door een Azure Pipeline in gang gezet worden. Hierdoor wordt het bestaande proces dat een aanzienlijk aantal stappen extra heeft vereenvoudigd tot één stap, namelijk de applicatie opslaan en pushen naar een Azure Repo.

Het onderzoek zal als een succes worden beschouwd indien de proof of concept een positief resultaat geeft, dit wil zeggen dat er moet aangetoond worden dat het mogelijk is om een Coolgen programma automatisch te laten compileren en binden door een Azure Pipeline met behulp van IBM DBB. Verder moet ook aangetoond worden dat er versiebeheer mogelijk is met behulp van Azure Repos om zo naar verschillende versies van programma's terug te kunnen keren. Indien beide vereisten worden ingelost kan er gesproken worden van een succesvol onderzoek

en bijgevolg een succesvolle proof of concept.

A.2. State-of-the-art

Vroeger waren de industrieel gestandaardiseerde servers gestapeld tot aan het plafond in elk datacenter. In die tijd was de mainframe computer een dinosaurus die gedoemd was om uit te sterven. IBM zag toch nog een toekomst voor de mainframe, hierdoor bleven ze maar innoveren op het 'Z' platform. Ze werden ondersteund door de opkomst van hybride cloud modellen en vele bedrijfskritische applicaties die op het mainframe moesten blijven. (Moorhead, 2022)

Deze innovaties waren er niet geweest zonder de vastberadenheid van IBM om het platform in leven te houden en sterker nog het nog levendiger te maken dan het voordien was. Door met name de IBM Wazi Developer for Red Hat CodeReady Workspaces uit te rollen hebben ze een cloudnative ontwikkel ervaring voor het z/OS besturingssysteem gemaakt. Met Wazi kunnen ontwikkelaars hun Integrated Development Environment, kortweg IDE, naar keuze gebruiken om mainframe toepassingen te ontwikkelen op het Red Hat OpenShift Container Platform. Een ander gebied van innovatie voor IBM is het bieden van beveiliging voor commerciële cryptocurrency wallets. (Bloomberg, 2021)

Een van deze hedendaagse innovaties is IBM Dependency Based Build, IBM DBB werkt aan de hand van een aantal groovy scripts die de verschillende commando's voor bijvoorbeeld een compilatie van een programma op z/OS uitvoeren. Deze scripts zijn volledig aanpasbaar waardoor ze een heel hoge mate van personalisatie toelaten, dit is een grote troef vooral in de mainframe omgeving van bedrijven die vaak heel hard aangepast zijn aan de werkwijze of policies van dat bedrijf. Volgens Porter (2019) is het leren werken met z/OS applicaties via groovy scripts en een moderne IDE zoals Visual Studio Code of IBM Developer for z/OS een goede manier om te leren omgaan met een mainframe zonder zelf op het mainframe zaken uit te voeren in het 3270 scherm. Dit zorgt ervoor dat vooral nieuwe ontwikkelaars die hun kans wagen in mainframe applicaties niet overweldigd worden door dat 3270 scherm en ze dus op een gekende/moderne manier kunnen kennismaken met een mainframe om dan eventueel later de stap te zetten naar werken met een green screen op hun eigen tempo. (Porter, 2019)

Om het uitrollen van applicaties met behulp van IBM DBB mogelijk te maken kan er gekeken worden naar een DevOps oplossing. Volgens Daniel Sokolowski (2021) verenigt DevOps software development en operations in cross-functional teams om zo de software delivery en operations te verbeteren. Hij zegt dat in een ideaal scenario de cross-functional DevOps teams hun services onafhankelijk uitrollen naar productie. In de praktijk is het vaak zo dat de correcte manier van het uitrollen

van een service andere services nodig heeft en dus zo nood heeft aan coördinatie om zo op een correcte manier hun service te kunnen uitrollen naar productie. Vaak wordt dit probleem opgelost door teams die een communicatie binnenkrijgen en dan manueel de deployment uitvoeren alsook de benodigde coördinatie moeten bepalen. Dit zorgt er dus voor dat de teams niet meer onafhankelijk zijn van elkaar en dus niet het ideale scenario is wat dus niet ten goede komt van de software deployment en operations hun performantie. Een oplossing voor dit probleem is een geautomatiseerd systeem die de coördinatie en uitvoering voor zijn rekening kan nemen, dit is wat men verstaat onder een CI/CD pipeline. Dit is een constructie dat is bedoeld om de software development en operations terug onafhankelijk te maken van elkaar en zo de performantie ervan te verbeteren door een geautomatiseerde oplossing te bieden voor de coördinatie en uitvoering voor het uitrollen van een service. (Daniel Sokolowski, 2021)

Naast de huidige innovaties is het ook wel interessant om te kijken naar hoe het landschap er over pakweg 10 jaar zou kunnen uitzien. Volgens Christopher Tozzi (2022) zal de mainframe nog alom tegenwoordig zijn na die 10 jaar, ook zullen er nog meer innovaties zijn gemaakt om het integreren met andere platformen zoals Windows en Linux of zelfs andere programmeertalen zoals Python, Javascript, C en vele anderen mogelijk te maken. Verder zullen de 'oude' technologieën van de Cobol programmeertaal blijven gemoderniseerd worden zodat het nog performanter wordt en nog makkelijker zal worden om COBOL-code te hergebruiken. De fysische voetafdruk van een mainframe zal ook alsmaar kleiner worden, nu is een moderne mainframe ongeveer even groot als een koelkast maar dat zou in de toekomst nog vele malen kleiner worden volgens Tozzi. (Tozzi, 2022)

A.3. Methodologie

De methodologie bestaat uit een aantal fases. In de eerste fase van dit onderzoek wordt er een uitgebreide literatuurstudie gemaakt door middel van literatuuronderzoek. Deze literatuurstudie zal gebruikt worden als voornaamste gegevensbron voor het verdere verloop van het onderzoek. De literatuurstudie heeft vier grote onderwerpen namelijk de mainframe, IBM Dependency Based Build, Git workflows en Azure DevOps. In het hoofdstuk omtrent de mainframe zal er gekeken worden wat een mainframe is, waarvoor het gebruikt wordt en wat de geschiedenis ervan is.

In het stuk over IBM Dependency Based Build zal er beschreven worden wat IBM DBB is, wat het doet en hoe het werkt. Verder zal er ook te vinden zijn wat een aantal van de valkuilen kunnen zijn waarmee rekening gehouden moet worden.

Het voorlaatste deel van de literatuurstudie zal een aantal Git workflows toelichten

en onderzoeken, hierin zal er gezocht worden naar een workflow die zowel flexibel, veilig en robuust is. Allemaal eigenschappen die synoniem staan aan een mainframe dus dat moet ook getoond worden dat dit nog altijd mogelijk is met een Git workflow.

In het laatste deel van de literatuurstudie zal er beschreven worden wat Azure DevOps is, wat het kan doen en waarvoor het in het onderzoek zal gebruikt worden. Een aantal onderwerpen die aan bod zullen komen in dit hoofdstuk zullen betrekking hebben op onderdelen van de Azure DevOps suite meer bepaald Azure Repos en Azure Pipelines. Waarvoor die zaken in het onderzoek gebruikt kunnen worden en wat voor meerwaarde dat geeft aan het onderzoek.

Deze eerste fase die geschat wordt op anderhalve week heeft als resultaat een diepgaande en volledige literatuurstudie die de rode draad zal zijn wat betreft informatie in het onderzoek. De literatuurstudie moet antwoord kunnen bieden op de volgende vraag. Hoe kan er een DevOps workflow gerealiseerd worden op een mainframe met behulp van IBM Dependency Based Build?

Het doel van de volgende fase is om IBM Dependency Based Build volledig te configureren op de omgeving van ArcelorMittal Gent, zo zal er onder andere een aantal .properties bestanden moeten aangemaakt en aangevuld worden en indien nodig zullen er zelf properties moeten toegevoegd worden aan deze bestanden. Door deze properties aan te passen zal IBM DBB de volledige toegang hebben tot de nodige datasets en bestanden op zowel z/OS als op Unix System Services (USS) om een goede werking van de scripts horende bij de software te kunnen garanderen. Dit zal gebeuren aan de hand van een vooraf bepaald applicatie die gebruikt zal worden als case study om zo de goede properties en configuraties te vinden. Dit vergt wat opzoekwerk en wordt geschat op 1 week om de volledige configuratie van IBM DBB te vervolledigen.

Na de configuratie van IBM DBB kan er overgegaan worden naar de grootste fase van het onderzoek namelijk het uitzetten en realiseren van de proof of concept. Deze fase is een heel belangrijke omwille van het feit dat hier de resultaten worden bekomen om een conclusie te trekken op het einde van dit onderzoek. De opdracht bestaat erin om de volledige ontwikkeling van een vooraf gekozen Coolgen applicatie van het mainframe te halen als case study, dat wil zeggen dat de source aangemaakt zal worden in een Git ondersteunende IDE zoals bijvoorbeeld Visual Studio Code of IBM Developer for Z (IDz) en vanuit diezelfde IDE een pipeline gestart wordt om het programma dan te compileren, binden en te plaatsen in de juiste load libraries op z/OS. Dit zal onder meer gebeuren via een push vanuit de IDE naar de Azure Repo die dan het aangepast programma opslaat en daarna

een pipeline start om het aangepaste programma ook te compileren en te binden en met als doel het plaatsen van de bekomen load module na de bind stap in de juiste load library te plaatsen. Verder moet er ook bij elke succesvolle aanpassing van het programma ook een nieuwe versie ontstaan zo zal een programma CGTST met initiële versie 1.0 na een succesvolle aanpassing bijvoorbeeld de versie 1.1 krijgen toegewezen. Zodat er op een duidelijke en overzichtelijke manier kan bijgehouden worden wat de huidige versie is van een programma, de vorige versies en eventueel de mogelijkheid om naar een vorige versie terug te gaan. Na deze fase die zes en een halve week zal duren zal er genoeg informatie zijn om een goede en volledige conclusie te trekken op de vraag of het zin heeft om volledig over te schakelen naar een DevOps gebaseerde manier van werken binnen de mainframe omgeving van ArcelorMittal Gent.

De volgende fase van dit onderzoek is om de resultaten die doorheen dit onderzoek zijn bekomen te analyseren. Deze analyse zal in samenspraak gebeuren met ArcelorMittal en een aantal ontwikkelaars om ook te polsen naar hun mening over het onderzoek en de resultaten ervan. De fase waarin deze analyse gebeurt zal 1 week in beslag nemen.

Na de resultatenanalyse volgt de fase waarin er conclusies worden getrokken, dit zal afhankelijk zijn van hoeveel vereisten er daadwerkelijk zijn bereikt en aan welke eventueel niet werd voldaan. Vereisten die zeker aan bod zullen komen in deze fase zijn als volgt.

- Kan het programma gecompileerd worden?
- Kan het programma gebind worden?
- Is de compilatie volgens de juiste compilatie parameters gebeurd?
- Is de bind volgens de juiste bind parameters gebeurd?
- Wordt het resultaat van de compilatie/bind op de juiste plaats en manier opgeslagen?
- Is het proces automatisch of moet er nog ergens manueel ingegrepen worden?

Voor deze fase zal er een focusgroep worden gemaakt om de vereisten en het al dan niet behalen ervan te bespreken en meningen uit te wisselen met mensen binnen de mainframe omgeving van ArcelorMittal Gent. Hiervoor wordt opnieuw 1 week gerekend.

De laatste fase is om de scriptie af te werken, zo zal er in die week gekeken worden om alle puntjes op de i te zetten en alles nog eens goed na te kijken alvorens

in te dienen. Deze laatste fase zal niet langer dan 1 week duren en zal ook het einde van dit onderzoek inluiden.

A.4. Verwacht resultaat, conclusie

Aan de hand van de vereisten die gesteld werden in samenspraak met ArcelorMittal Gent zal er gekeken worden of die ook daadwerkelijk worden ingelost. Die vereisten zijn als volgt :

- Het Coolgen programma moet zonder fouten gecompileerd worden.
- Het programma moet succesvol gebind worden en in de juiste load libraries opgeslagen worden.
- Het programma moet uitvoerbaar zijn na compilatie en bind.
- Er moet een automatisch versiebeheer systeem zijn gebaseerd op Git versiebeheer.
- Alle taken die hierboven zijn opgesomd moeten automatisch kunnen gebeuren.

De verwachting is dat het onderzoek zal aantonen dat de bovengenoemde vereisten ingelost kunnen worden, dit zal als resultaat geven dat er een eenvoudige en overzichtelijke pipeline zal werken die een Coolgen programma compileert, bind en aan versiebeheer doet. Hierdoor zal er geconcludeerd kunnen worden dat het wel degelijk mogelijk is om ontwikkelaars te laten werken buiten een mainframe omgeving aan een Coolgen applicatie en zo op die manier een release management systeem te gebruiken dat gebaseerd is op IBM DBB met behulp van Azure DevOps en Git.

B

Properties bestanden (build-conf)

B.1. build.properties

```
################################################################################
# build.properties configuration to specify
#  global build properties for zAppBuild
#
################################################################################


################################################################################
# Global build properties used by zAppBuild
#

#
# Comma separated list of additional build property files to load
# Supports both relative path (to zAppBuild/build-conf/) and absolute path
#
# These properties files expect to contain centrally managed defaults
#  such as system datasets, language script specific settings
#
buildPropFiles=datasets.properties,\
dependencyReport.properties,\
Assembler.properties,\
Cobol.properties,\
PLI.properties,\
LinkEdit.properties,\
Transfer.properties,\
MFS.properties,\
PSBgen.properties,\
DBDgen.properties,\
```

```
ACBgen.properties,\
ZunitConfig.properties

#
# Comma separated list of default application configuration property files to load
# Supports both relative path (to zAppBuild/build-conf/) and absolute path
#
# These properties files expect to contain centrally managed defaults
#  and also may reference properties files containing configuration
#  of the language script configurations such as return codes, deploy types
#
# See also application-conf/application.properties#applicationPropFiles
#
# default:
# applicationDefaultPropFiles=defaultzAppBuildConf.properties
#
# extended sample to set default language script configurations:
#
# applicationDefaultPropFiles=defaultzAppBuildConf.properties,\
# default-application-conf/searchPaths.properties,\
# default-application-conf/scriptMappings.properties,\
# default-application-conf/Cobol.properties,\
# default-application-conf/PLI.properties,\
# default-application-conf/Transfer.properties,\
# default-application-conf/LinkEdit.properties,\
# default-application-conf/ZunitConfig.properties
#
applicationDefaultPropFiles=defaultzAppBuildConf.properties

#
# applicationConfDir
#
# required build property that specifies the directory of the
# application specific build configurations for zAppBuild, also known
# as application-conf folder
#
# a sample of the application-conf can be found in
#   samples/application-conf
#
# zAppBuild expects a file called application.properties in this directory.
#
# The property also allows for the deployment of
# the application-conf directories to an alternate location rather
# in the application repository.
#
# Default:
# Location within the application repository and resolving the configuration
# based on workspace + application and application-conf
#
# applicationConfDir=${workspace}/${application}/application-conf
#
```

```
#
# Example: Static location on USS
# applicationConfDir=/u/build/config/applications/${application}/application-conf
# |- /u/build/config/applications
# |                       |- App1
# |                       |    |- application-conf
# |                       |    |        |- application.properties
# |                       |- App2
# |                       |    |- application-conf
#
# Example: Application config files stored in zAppBuild
# applicationConfDir=${zAppBuildDir}/applications/${application}/application-conf
# |- /u/build/zAppBuild/applications
# |                          |- App1
# |                          |    |- application-conf
# |                          |    |        |- application.properties
# |                          |- App2
# |                          |    |- application-conf
#
applicationConfRootDir=${workspace}/application-conf

#
# file extension that indicates the build file is really a build list or build list filter
buildListFileExt=txt

#
# Determine if a subfolder with a timestamp should be created in the buildOutDir location.
# Applies to all build types except userBuild
# Default: true
createBuildOutputSubfolder=true

#
# Build Timestamp Format
# Applies to all build types except userBuild
# Default: yyyyMMdd.HHmmss.mmm - See Date format method pattern strings
##buildOutputTSformat=yyyyMMdd.HHmmss.mmm

#
# Minimum required DBB ToolkitVersion to run this version of zAppBuild
#  Build initialization process validates the DBB Toolkit Version in use and matches that against th
requiredDBBToolkitVersion=2.0.0

#
# Comma separated list of required build properties for zAppBuild/build.groovy
requiredBuildProperties=buildOrder,buildListFileExt

# dbb.file.tagging controls compile log and build report file tagging. If true, files
# written as UTF-8 or ASCII are tagged.
# If the environment variable _BPXK_AUTOCVT is set ALL, file tagging may have an
# adverse effect if viewing log files and build report via Jenkins.
# In this case, set dbb.file.tagging to false or comment out the line. Default: true
```

```
dbb.file.tagging=true

# MetadataStore configuration properties:

# select MetadataStore configuration (either 'file' or 'db2')
metadataStoreType=file

# location of file metadata store.  Default is $USER
#metadataStoreFileLocation=

# Db2 metadata server url
# build.groovy option -url, --url
#metadataStoreDb2Url=jdbc:db2:<Db2 server location>

# Db2 connection configuration property file
# Sample is povided at $DBB_HOME/conf/db2Connection.conf
#metadataStoreDb2ConnectionConf=


# The dbb.gateway.type property determines which gateway type is used for the entire build process
# Possible values are 'legacy' and 'interactive'.  Default if not indicated is 'legacy'
dbb.gateway.type=legacy

# Procedure Name - specified with the procname parameter
#dbb.gateway.procedureName=

# Account number - specified with the acctnum parameter
#dbb.gateway.accountNumber=

# Group name - specified with a groupid parameter
#dbb.gateway.groupId=

# Region size - specified with the regionsz parameter
#dbb.gateway.regionSize=

# Gateway logging level.  Add values for multiple types:
# 1 - Log error information
# 2 - Log debug information
# 4 - Log communication information
# 8 - Log time information
# 16 - Log information to the system console
dbb.gateway.logLevel=16
```

B.2. datasets.properties

```
##
# Template from https://github.com/IBM/dbb-zappbuild
# Github from IBM with a lot of documentation and
# configuration options for DBB
##
# Dataset references
# Build properties for Partition Data Sets (PDS) used by zAppBuild build scripts.
# Please provide a fully qualified DSN for each build property below.
##

# z/OS macro library. Example: SYS1.MACLIB
MACLIB=SYS1.MACLIB

# Assembler macro library. Example: CEE.SCEEMAC
SCEEMAC=CEE.SCEEMAC

# LE (Language Environment) load library. Example: CEE.SCEELKED
SCEELKED=CEE.SCEELKED

# High Level Assembler (HLASM) load library. Example: ASM.SASMMOD1
SASMMOD1=ASM.SASMMOD1

# Cobol Compiler Data Sets. Example: COBOL.V4R1M0.SIGYCOMP
SIGYCOMP_V6=IGY.V620.SIGYCOMP

# PL/I Compiler Data Sets. Example: PLI.V5R2M0.SIBMZCMP
IBMZPLI_V52=IBMZ.SIBMZCMP
#IBMZPLI_V51=

# MQ COBOL Library. Example: CSQ.V9R1M0.SCSQCOBC
SCSQCOBC=MQM.SCSQCOBC.V930N046

# MQ PLI Library. Example: CSQ.V9R1M0.SCSQPLIC
SCSQPLIC=MQM.SCSQPLIC.V930N046

# MQ Assembler Library. Example: CSQ.V9R1M0.SCSQMACS
SCSQMACS=MQM.SCSQMACS.V930N046

# MQ Load Library. Example: CSQ.V9R1M0.SCSQLOAD
SCSQLOAD=MQM.SCSQLOAD.V920M026

# DB2 Load Library. Example: DB2.V9R1M0.SDSNLOAD
SDSNLOAD=DB2.SDSNLOAD.V1223011

# DB2 Exit Library. Example: DBC0CFG.SDSNEXIT
SDSNEXIT=DB2.SDSNEXIT

# IMS Macro Library. Example: DFS.V11R1M0.SDFSMAC
SDFSMAC=IMS.SDFSMAC.V150M027
```

```
# IMS RESLIB. Example: DFS.V11R1M0.SDFSRESL
SDFSRESL=IMS.RESLIB

# User generated library for DB/DC and DC installations. Example: DFS.V11R1M0.REFERAL
REFERAL=IMS.REFERAL

# IBM Debug Library containing Exits
SEQAMOD=EQAW.SEQAMOD

# Optional IDz Load Library. Example: FEL.V14R0M0.SFELLOAD Data set not found
SFELLOAD=FEL.SFELLOAD

# Optional IDZ zUnit / WAZI VTP library containing necessary copybooks. Example : FEL.V14R2.SBZUSAMP
SBZUSAMP=BZU.SBZUSAMP

# REXX Compiler Data Sets. Example: REXX.V1R4.SFANLMD Data set not found
#SFANLMD=REXX.SFANLMD

# PD Tools Common Component load library. Example : PDTCC.V1R8.SIPVMODA
PDTCCMOD=IPV.SIPVMODA
```

B.3. Cobol.properties

```
# Releng properties used by language/Cobol.groovy

#
# Comma separated list of required build properties for Cobol.groovy
cobol_requiredBuildProperties=cobol_srcPDS,\
cobol_cpyPDS,\
cobol_objPDS,\
cobol_loadDatasets,\
cobol_loadDatasetsFSUB,\
cobol_loadDatasetsSUB,\
cobol_compiler,\
cobol_linkEditor,\
cobol_tempOptions,\
applicationOutputsCollectionName,\
SDSNLOAD,SCEELKED,\
cobol_dependencySearch

#
# COBOL compiler name
cobol_compiler=IGYCRCTL

#
# linker name
cobol_linkEditor=IEWL

#
# COBOL source data sets
cobol_srcPDS=${hlq}.SOURCE.COB
cobol_cpyPDS=${hlq}.SOURCE.COPY
cobol_objPDS=${hlq}.COB.OBJ
cobol_dbrmPDS=${hlq}.COB.DBRM

#
# COBOL load data sets (MAINS)
cobol_loadPDS=DEV1.LOAD.MAIN.PDSE1
cobol_loadPDS2=DEV2.LOAD.MAIN.PDSE1

#
# COBOL load data sets (FSUBS)
cobol_loadPDS_FSUBIMS=DEV1.LOAD.FSUB.IMS.PDSE1
cobol_loadPDS_FSUBIMS2=DEV2.LOAD.FSUB.IMS.PDSE1
cobol_loadPDS_FSUBRRS=DEV1.LOAD.FSUB.RRS.PDSE1
cobol_loadPDS_FSUBRRS2=DEV2.LOAD.FSUB.RRS.PDSE1

#
# COBOL load data sets (SUBS)
cobol_loadPDS_SUB=DEV1.LOAD.SUB.PDSE1

#
```

```
# COBOL bind parameters (linkedit)
# -----PARMS FOR MAIN & FSUB PROGRAMS----
cobol_linkEditParms=LET,STRIPSEC,LIST(NOIMP),RENT,DYNAM(DLL),CASE(MIXED),COMPAT(PM5),XREF,MAP
# -----ONLY FOR SUB PROGRAMS----
cobol_linkEditParmsSUB=NOCALL,LIST,LET,AMODE(31),RMODE(ANY),RENT,CASE(MIXED),COMPAT(PM5),XREF,MAP

#
# COBOL test case source data sets
cobol_testcase_srcPDS=${hlq}.TEST.COBOL

#
# COBOL test case load data sets
cobol_testcase_loadPDS=${hlq}.TEST.LOAD

#
# List the data sets that need to be created and their creation options
cobol_srcDatasets=${cobol_srcPDS},${cobol_cpyPDS},${cobol_dbrmPDS}
cobol_srcOptions=cyl space(1,1) lrecl(80) dsorg(PO) recfm(F,B) dsntype(library)

cobol_loadDatasets=${cobol_loadPDS},${cobol_loadPDS2}
cobol_loadDatasetsFSUB=${cobol_loadPDS_FSUBIMS},${cobol_loadPDS_FSUBIMS2},${cobol_loadPDS_FSUBRRS},$
cobol_loadDatasetsSUB=${cobol_loadPDS_SUB}
cobol_loadOptions=cyl space(1,1) dsorg(PO) recfm(U) blksize(32760) dsntype(library)

cobol_tempOptions=cyl space(5,5) unit(vio) blksize(80) lrecl(80) recfm(f,b) new
cobol_printTempOptions=cyl space(5,5) unit(vio) blksize(133) lrecl(133) recfm(f,b) new


#
# List the data sets for tests that need to be created and their creation options
cobol_test_srcDatasets=${cobol_testcase_srcPDS}
cobol_test_srcOptions=cyl space(1,1) lrecl(80) dsorg(PO) recfm(F,B) dsntype(library)

cobol_test_loadDatasets=${cobol_testcase_loadPDS}
cobol_test_loadOptions=cyl space(1,1) dsorg(PO) recfm(U) blksize(32760) dsntype(library)

# Allocation of SYSMLSD Dataset used for extracting Compile Messages to Remote Error List
cobol_compileErrorFeedbackXmlOptions=tracks space(200,40) dsorg(PS) blksize(27998) lrecl(16383) recfm

#
# List of output datasets to document deletions
cobol_outputDatasets=${cobol_loadPDS},${cobol_loadPDS2},${cobol_dbrmPDS}

#
# Set filter used to exclude certain information from the link edit scanning.
# The value contains a comma separated list of patterns.
# example: A filter of *.SUB1, *.SUB2 will exclude modules SUB1 and SUB2
#          from any dataset. To exclude member HELLO in PDS TEST.COBOL will
#          be matched by the pattern TEST.COBOL.HELLO. The pattern TEST.COBOL.*
#          will match any member in the data set TEST.COBOL.
# The following filter excludes CICS and LE Library references.
```

```
dbb.LinkEditScanner.excludeFilter = ${SCEELKED}.*

#
# additional libraries for compile SYSLIB concatenation, comma-separated, see definitions in applica
cobol_compileSyslibConcatenation= DEV.INCLUDE,\
                                  PROD.INCLUDE,\
                                  DEV.INCLUDE.PARM,\
                                  DEV.SYSTEEM.INCLUDE

#
# additional libraries for linkEdit SYSLIB concatenation, comma-separated, see definitions in applic
cobol_linkEditSyslibConcatenation= DEV.PARMLIB,\
                                   PROD.PARMLIB,\
                                   DEV.LOAD.SUB.PDSE1,\
                                   PROD.LOAD.SUB.PDSE1,\
                                   TBHR.INCLUDE,\
                                   ISP.SISPLOAD,\
                                   SID0S0.QSO.MQM.SCSQCOBC,\
                                   MPR.MP.V2R760.LOADLIB,\
                                   CEE.SCEELKED,\
                                   CEE.SCEELKEX,\
                                   CEE.SCEEOBJ,\
                                   CEE.SCEECPP

#   * copy files the to mapped dataset definition (PLEASE NOTE! This setting takes precendence over
#   * defining additional allocations in the compile step
#
#  note that the SYSLIB is defaulted to the dataset definition 'cobol_cpyPDS' and is not required to
#  sample: cobol_dependenciesAlternativeLibraryNameMapping = {"MYFILE": "cobol_myfilePDS", "DCLGEN"
######
#cobol_dependenciesAlternativeLibraryNameMapping=

# cobol_dependenciesDatasetMapping - an optional dbb property mapping to map dependencies to differe
#  this property is used when dependencies are copied to the different build libraries, e.q dclgens
#  note, that a dependency file needs to match a single rule
#
#  sample:
#   cobol_dependenciesDatasetMapping = cobol_cpyPDS :: **/copybook/*.cpy
#   cobol_dependenciesDatasetMapping = cobol_dclgenPDS :: **/dclgens/*.cpy
#
#  default copies all dependencies into the dependency dataset definition which was previously passe
#   cobol_dependenciesDatasetMapping = cobol_cpyPDS :: **/*
######
#cobol_dependenciesDatasetMapping = cobol_cpyPDS :: **/*
```

C

Language groovy scripts

C.1. Cobol.groovy

```
/*
 * createVersion - Creates a version for DB2 BIND PACKAGE
 * looks like $$$DBRMVR-TSTCOB2-2023112858001
 * $$$DBRMV(R) -> either only DBRMVR or also DBRMVS => only in double compile ex. new PL/I compiler
 * TSTCOB2 -> program name ex. program TESTHFD will have TESTHFD in it's version string
 * 2023112858001 -> first 8 digits (20231128) are the date in format yyyyMMdd
 *               -> last 5 digits (58001) is the time in seconds since last midnight: hours * 3600 +
 */
def createVersion(LogicalFile logicalFile) {
    def dateTime = new Date()
    def tz = TimeZone.getTimeZone('GMT+1')
    def (hours, minutes, seconds) = [dateTime.format('HH', tz) as Integer, dateTime.format('mm', tz)
    def packtime = "${dateTime.format(' 'yyyyMMdd',  tz )}${hours * 3600 + minutes * 60 + seconds}"
    def version = "\$\$\$DBRMVR-${logicalFile.getLname()}-$packtime"

    return version

}

/*
 * createBindPackageCommand - creates a JCLExec command to package bind the program (buildFile)
 */
def createBindPackageCommand(String buildFile, String member, String owner) {
    String jclDataset = "${System.getProperty('user.name')}.SOURCE.CNTL"
    String jclPackageMember = 'PACKBIND'

    clistDSN = bindUtils.makeBindPackage(buildFile,
                                props.cobol_dbrmPDS,
                                props.buildOutDir,
                                props.getFileProperty('db2Subsys', buildFile),
                                owner,
```

```
                                props.getFileProperty('db2Qual', buildFile))

    jclLocation = new File("${props.buildOutDir}/${member}_packagedummy.jcl")

    // Getting the template file
    def dummyBind = new CopyToHFS()
    dummyBind.setDataset(dummyBindDataset)
    dummyBind.setMember(dummyBindMember)
    dummyBind.setFile(jclLocation)
    dummyBind.execute()

    //Altering the template file
    jclLocation.append("//SYSTSIN␣␣DD␣DSN=$clistDSN,\n//␣␣␣␣␣␣␣␣␣␣DISP=SHR")

    // Copying the altered file to PDS
    new CopyToPDS().file(jclLocation).dataset(jclDataset).member(jclPackageMember).execute()

    // Assigning the copied JCL
    JCLExec bindPackage = new JCLExec()
    bindPackage.setDataset(jclDataset)
    bindPackage.setMember(jclPackageMember)

    return bindPackage
}

/*
 * createBindPlanCommand - creates a JCLExec command to plan bind the program (buildFile)
 */
def createBindPlanCommand(String buildFile, String member) {
    String jclDataset = "${System.getProperty('user.name')}.SOURCE.CNTL"
    String jclPlanMember = 'PLANBIND'

    clistDSN = bindUtils.makeBindPlan(buildFile,
                                props.cobol_dbrmPDS,
                                props.buildOutDir,
                                props.getFileProperty('db2Subsys', buildFile))

    jclLocation = new File("${props.buildOutDir}/${member}_plandummy.jcl")

    // Getting the template file
    def dummyBind = new CopyToHFS()
    dummyBind.setDataset(dummyBindDataset)
    dummyBind.setMember(dummyBindMember)
    dummyBind.setFile(jclLocation)
    dummyBind.execute()

    //Altering the template file
    jclLocation.append("//SYSTSIN␣␣DD␣DSN=$clistDSN,\n//␣␣␣␣␣␣␣␣␣␣DISP=SHR")

    // Copying the altered file to PDS
    new CopyToPDS().file(jclLocation).dataset(jclDataset).member(jclPlanMember).execute()
```

```
    // Assigning the copied JCL
    JCLExec bindPlan = new JCLExec()
    bindPlan.setDataset(jclDataset)
    bindPlan.setMember(jclPlanMember)

    return bindPlan
}

/*
 * createCompileCommand - creates a MVSExec command for compiling the COBOL program (buildFile)
 */
def createCompileCommand(String buildFile, LogicalFile logicalFile, String member, File logFile) {
    String compiler = props.getFileProperty('cobol_compiler', buildFile)

    (parmFile, parms) = createCobolParms(buildFile, logicalFile)

    // get the right load PDSEs in place
    if (logicalFile.isFSUB())
        loadDatasets = props.cobol_loadDatasetsFSUB.split(',')
    else if (logicalFile.isSUB())
        loadDatasets = props.cobol_loadDatasetsSUB.split(',')
    else
        loadDatasets = props.cobol_loadDatasets.split(',')

    println("Show␣the␣load␣library␣datasets:␣$loadDatasets")

    // define the MVSExec command to compile the program
    MVSExec compile = new MVSExec().file(buildFile).pgm(compiler).parm(parms)

    // add DD statements to the compile command
    if (isZUnitTestCase){
        compile.dd(new DDStatement().name("SYSIN").dsn("${props.cobol_testcase_srcPDS}($member)").op
    }
    else
    {
        compile.dd(new DDStatement().name("SYSIN").dsn("${props.cobol_srcPDS}($member)").options('sh
    }

    compile.dd(new DDStatement().name("SYSPRINT").options(props.cobol_printTempOptions))
    compile.dd(new DDStatement().name("SYSOPTF").dsn(parmFile).options('shr'))
    compile.dd(new DDStatement().name("SYSDEFSD").options(props.cobol_tempOptions))
    compile.dd(new DDStatement().name("SYSMDECK").options(props.cobol_tempOptions))

    (1..17).toList().each { num ->
        compile.dd(new DDStatement().name("SYSUT$num").options(props.cobol_tempOptions))
    }

    // define object dataset allocation
    compile.dd(new DDStatement().name("SYSLIN").dsn("${props.cobol_objPDS}($member)").options('shr'
```

```
// add a syslib to the compile command with optional bms output copybook and concatenation
compile.dd(new DDStatement().name("SYSLIB").dsn(props.cobol_cpyPDS).options("shr"))

// add additional datasets with dependencies based on the dependenciesDatasetMapping
PropertyMappings dsMapping = new PropertyMappings('cobol_dependenciesDatasetMapping')
dsMapping.getValues().each { targetDataset ->
    // exclude the defaults cobol_cpyPDS and any overwrite in the alternativeLibraryNameMap
    if (targetDataset != 'cobol_cpyPDS')
        compile.dd(new DDStatement().dsn(props.getProperty(targetDataset)).options("shr"))
}

// add custom concatenation
def compileSyslibConcatenation = props.getFileProperty('cobol_compileSyslibConcatenation', buil
def compileSyslibAddConcatenation = props.cobol_compileSyslibAddConcat
if (compileSyslibConcatenation && compileSyslibAddConcatenation)
    compileSyslibConcatenation = compileSyslibConcatenation + ',' + compileSyslibAddConcatenatio

if (logicalFile.isDB2())
    compileSyslibConcatenation = 'DEV.DCLGEN,PROD.DCLGEN,GEBR.DCLGEN,' + compileSyslibConcatenat

println("compileSyslibConcat:␣${compileSyslibConcatenation}")
if (compileSyslibConcatenation) {
    def String[] syslibDatasets = compileSyslibConcatenation.split(',');
    for (String syslibDataset : syslibDatasets )
        compile.dd(new DDStatement().dsn(syslibDataset).options("shr"))
}

// add subsystem libraries
if (buildUtils.isMQ(logicalFile))
    compile.dd(new DDStatement().dsn(props.SCSQCOBC).options("shr"))

// add additional zunit libraries
if (isZUnitTestCase)
compile.dd(new DDStatement().dsn(props.SBZUSAMP).options("shr"))

// add a tasklib to the compile command with optional IMS, DB2, and IDz concatenations
String compilerVer = props.getFileProperty('cobol_compilerVersion', buildFile)
compile.dd(new DDStatement().name("TASKLIB").dsn(props."SIGYCOMP_$compilerVer").options("shr"))
if (buildUtils.isDB2(logicalFile)) {
    if (props.SDSNEXIT) compile.dd(new DDStatement().dsn(props.SDSNEXIT).options("shr"))
    compile.dd(new DDStatement().dsn(props.SDSNLOAD).options("shr"))
}
if (props.SFELLOAD)
    compile.dd(new DDStatement().dsn(props.SFELLOAD).options("shr"))

// add optional DBRMLIB if build file contains DB2 code
if (buildUtils.isDB2(logicalFile))
    compile.dd(new DDStatement().name("DBRMLIB").dsn("$props.cobol_dbrmPDS($member)").options('s


// adding alternate library definitions
```

```
    if (props.cobol_dependenciesAlternativeLibraryNameMapping) {
        alternateLibraryNameAllocations = buildUtils.parseJSONStringToMap(props.cobol_dependenciesA
        alternateLibraryNameAllocations.each { libraryName, datasetDefinition ->
            datasetName = props.getProperty(datasetDefinition)
            if (datasetName) {
                compile.dd(new DDStatement().name(libraryName).dsn(datasetName).options("shr"))
            }
            else {
                String errorMsg = "*! Cobol.groovy. The dataset definition $datasetDefinition could
                println(errorMsg)
                props.error = "true"
                buildUtils.updateBuildResult(errorMsg:errorMsg)
            }
        }
    }

    // add IDz User Build Error Feedback DDs
    if (props.errPrefix) {
        compile.dd(new DDStatement().name("SYSADATA").options("DUMMY"))
        // SYSXMLSD.XML suffix is mandatory for IDZ/ZOD to populate remote error list
        compile.dd(new DDStatement().name("SYSXMLSD").dsn("${props.hlq}.${props.errPrefix}.SYSXMLSD.
    }

    // add a copy command to the compile command to copy the SYSPRINT from the temporary dataset to
    compile.copy(new CopyToHFS().ddName("SYSPRINT").file(logFile).hfsEncoding(props.logEncoding))
    compile.copy(new CopyToHFS().ddName("*").file(new File("${props.buildOutDir}/${member}_compall.l

    return compile
}

/*
 * createLinkEditCommand - creates a MVSExec xommand for link editing the COBOL object module produc
 */
def createLinkEditCommand(String buildFile, LogicalFile logicalFile, String member, File logFile) {
    String linker = props.getFileProperty('cobol_linkEditor', buildFile)
    String linkEditStream = props.getFileProperty('cobol_linkEditStream', buildFile)
    String linkDebugExit = props.getFileProperty('cobol_linkDebugExit', buildFile)
    String parms = ""

    // assign the first loadPDS in the list of PDSEs as the primary
    String cobol_loadPDS = loadDatasets[0]

    // assign parameters for the bind
    if (logicalFile.isSUB())
        parms = props.getFileProperty('cobol_linkEditParmsSUB', buildFile)
    else
        parms = props.getFileProperty('cobol_linkEditParms', buildFile)

    println("Link-Edit parms for $buildFile = $parms")

    if (props.verbose) println "*** Link-Edit parms for $buildFile = $parms"
```

```
// define the MVSExec command to link edit the program
MVSExec linkedit = new MVSExec().file(buildFile).pgm(linker).parm(parms)

// add DD statements to the linkedit command
String deployType = buildUtils.getDeployType("cobol", buildFile, logicalFile)
if(isZUnitTestCase){
    linkedit.dd(new DDStatement().name("SYSLMOD").dsn("${props.cobol_testcase_loadPDS}($member)"
}
else {
    linkedit.dd(new DDStatement().name("SYSLMOD").dsn("${cobol_loadPDS}($member)").options('shr'
}

// add SYSDEFSD to avoid IEW2689W 4C40 DEFINITION SIDE FILE IS NOT DEFINED message from program
linkedit.dd(new DDStatement().name("SYSDEFSD").options(props.cobol_tempOptions))
linkedit.dd(new DDStatement().name("SYSPRINT").options(props.cobol_printTempOptions))
linkedit.dd(new DDStatement().name("SYSUT1").options(props.cobol_tempOptions))

// Assemble linkEditInstream to define SYSIN as instreamData
String sysin_linkEditInstream = "INCLUDE␣SYSLIB(${member})\n"

// appending configured linkEdit stream if specified
if (linkEditStream) {
    sysin_linkEditInstream += "␣␣" + linkEditStream.replace("\\n","\n").replace('@{member}',mem
}

if (logicalFile.isFSUB()) {
    sysin_linkEditInstream += "␣␣INCLUDE␣SYSLIB(DSNULI)"
}

// appending mq stub according to file flags
if(buildUtils.isMQ(logicalFile)) {
    sysin_linkEditInstream += buildUtils.getMqStubInstruction(logicalFile)
}

// appending debug exit to link instructions
if (props.debug && linkDebugExit!= null) {
    sysin_linkEditInstream += "␣␣␣" + linkDebugExit.replace("\\n","\n").replace('@{member}',mem
}

// Define SYSIN dd
if (sysin_linkEditInstream)
    linkedit.dd(new DDStatement().name("SYSIN").instreamData(sysin_linkEditInstream))

// add SYSLIN along the reference to SYSIN if configured through sysin_linkEditInstream
def omgeving = (buildUtils.isDev(logicalFile) && !(buildUtils.isProd(logicalFile))) || (!(build

linkedit.dd(new DDStatement().name("SYSLIN").dsn("${props.cobol_objPDS}($member)").options('shr'
linkedit.dd(new DDStatement().dsn("CEE.SCEELIB(C128N)").options("shr"))
linkedit.dd(new DDStatement().dsn("CBC.SCLBSID(IOSTREAM)").options("shr"))
linkedit.dd(new DDStatement().dsn("CBC.SCLBSID(COMPLEX)").options("shr"))
```

```
linkedit.dd(new DDStatement().dsn("${omgeving}.PARMLIB(IMPORTS)").options("shr"))
linkedit.dd(new DDStatement().dsn("DEV.PARMLIB(IMPORTS)").options("shr"))

// add RESLIB if needed
if ( props.RESLIB )
    linkedit.dd(new DDStatement().name("RESLIB").dsn(props.RESLIB).options("shr"))

// add a syslib to the compile command with optional concatenation
linkedit.dd(new DDStatement().name("SYSLIB").dsn(cobol_loadPDS).options("shr"))

// add custom concatenation
def linkEditSyslibConcatenation = props.getFileProperty('cobol_linkEditSyslibConcatenation', bu
def linkEditSyslibAddConcatenation = props.cobol_linkEditSyslibAddConcat

// if IMS add syslib concat
if (logicalFile.isIMS()) {
    imsReslib = (logicalFile.isDev()) ? 'DEV.IMS.RESLIB':'PROD.IMS.RESLIB'
    linkEditSyslibConcatenation = "$imsReslib,$linkEditSyslibConcatenation"
}

// add SDSNLOAD to the syslib concat
if (logicalFile.isDB2() || logicalFile.isFSUB()) {
    db2Load = (logicalFile.isDev()) ? ['DB2.DBO1.SDSNLOAD', 'DB2.DBO2.SDSNLOAD']:['DB2.DBP1.SDSN
    (linkEditSyslibConcatenation?.trim()) ? (linkEditSyslibConcatenation = linkEditSyslibConcate
}

// if extra syslib concat in Cobol.properties then add
if (linkEditSyslibConcatenation && linkEditSyslibAddConcatenation)
    linkEditSyslibConcatenation = linkEditSyslibConcatenation + ',' + linkEditSyslibAddConcatena

if (linkEditSyslibConcatenation) {
    println("Linkedit␣syslib␣concat:␣${linkEditSyslibConcatenation}")
    def String[] syslibDatasets = linkEditSyslibConcatenation.split(',');
    for (String syslibDataset : syslibDatasets )
        linkedit.dd(new DDStatement().dsn(syslibDataset).options("shr"))
}

// if you want extra lib concat but not in syslib
def linkEditAddlibConcat = props.cobol_linkEditAddlibConcat
if (linkEditAddlibConcat) {
    println("Linkedit␣altlib␣concat:␣$linkEditAddlibConcat")
    def String[] addlibDatasets = linkEditAddlibConcat.split(',');
    def int teller = 0;
    for (String addlibDataset : addlibDatasets) {
        if (teller == 0)
            linkedit.dd(new DDStatement().name("ALTLIB").dsn(addlibDataset).options("shr"))
        else
            linkedit.dd(new DDStatement().dsn(addlibDataset).options("shr"))
        teller ++
    }
}
```

```
    linkedit.dd(new DDStatement().dsn(props.SCEELKED).options("shr"))

    // Add Debug Dataset to find the debug exit to SYSLIB
    if (props.debug && props.SEQAMOD)
        linkedit.dd(new DDStatement().dsn(props.SEQAMOD).options("shr"))

    if (buildUtils.isMQ(logicalFile))
        linkedit.dd(new DDStatement().dsn(props.SCSQLOAD).options("shr"))

    // add a copy command to the linkedit command to append the SYSPRINT from the temporary dataset
    linkedit.copy(new CopyToHFS().ddName("SYSPRINT").file(logFile).hfsEncoding(props.logEncoding).ap

    if (logicalFile.isFSUB())
        linkedit.copy(new CopyToHFS().ddName("SYSDEFSD").file(new File("${props.buildOutDir}/${memb

    return linkedit
}

/*
 * addImportStatements - searches through the MDECK file generated at compile/bind time and adds new
 * imports to the existing IMPORTS file on z/OS. In case of duplicate entry it won't perform the ad
 * of the import statement.
 */
def addImportStatements(String member) {
    // copy original IMPORTS file to HFS
    def imports = new CopyToHFS()
    imports.setDataset("DEV.PARMLIB")
    imports.setMember("IMPORTS")
    imports.setFile(new File("${props.buildOutDir}/imports.imp"))
    imports.execute()

    // search IMPORTS file if there is already a import statement for the application
    String SEARCH_STR = "'$member','"
    def importsFile = new File("${props.buildOutDir}/imports.imp")
    def result = ""

    // assign deck file
    deckFile = new File("${props.buildOutDir}/${member}_sysdefsd.imp")

    // search deck file for the search string
    if (deckFile.text.contains(SEARCH_STR)) {
        deckFile.withReader { reader ->
            while ((line = reader.readLine()) != null) {
                if (line.contains(SEARCH_STR) && !(line.contains('_')) && !(importsFile.text.contain
                    result += "␣${line.trim()}\n"
            }
        }
    }

    // append the result to the IMPORTS file
```

```
    importsFile.append(result)

    // copy the changed IMPORTS file to PDS
    if (result) {
        def newImports = new CopyToPDS()
        newImports.setFile(importsFile)
        newImports.setDataset("DEV.PARMLIB")
        newImports.setMember("IMPORTS")
        newImports.execute()
    }

    // clean up directory
    if (importsFile.exists())
        importsFile.delete()
    if (deckFile.exists())
        deckFile.delete()
}
```

D

Utility groovy scripts

D.1. BuildUtilities.groovy

```
/*
 * sortBuildList - sorts a build list by rank property values
 */
def sortBuildList(List<String> buildList) {
    List<String> sortedList = []
    TreeMap<Integer, List<String>> rankings = new TreeMap<Integer, List<String>>()
    List<String> fileBuildOrder = props.cobol_fileBuildOrder.split(',')
    List<String> unranked = new ArrayList<String>()

    // sort buildFiles by rank
    buildList.each { buildFile ->
        // create LogicalFile to determine the properties isFSUB(), isDB2(), ...
        String dependencySearch = props.getFileProperty('cobol_dependencySearch', buildFile)
        SearchPathDependencyResolver dependencyResolver = new SearchPathDependencyResolver(dependen
        LogicalFile logicalFile = createLogicalFile(dependencyResolver, buildFile)
        def flag = false

        for (rank in fileBuildOrder) {
            def index = fileBuildOrder.indexOf(rank)

            switch (rank) {
                case "FSUB":
                    List<String> ranking = rankings.get(index)
                    if (!ranking) {
                        ranking = new ArrayList<String>()
                        rankings.put(index,ranking)
                    }
                    if (isFSUB(logicalFile)) {
                        ranking << buildFile
                        flag = true
                    }
```

```
                    break;
                case "SUB":
                    List<String> ranking = rankings.get(index)
                    if (!ranking) {
                        ranking = new ArrayList<String>()
                        rankings.put(index,ranking)
                    }
                    if (isSUB(logicalFile)) {
                        ranking << buildFile
                        flag = true
                    }
                    break;
                case "MAIN":
                    List<String> ranking = rankings.get(index)
                    if (!ranking) {
                        ranking = new ArrayList<String>()
                        rankings.put(index,ranking)
                    }
                    if (isMain(logicalFile)) {
                        ranking << buildFile
                        flag = true
                    }
                    break;
                default:
                    List<String> ranking = rankings.get(fileBuildOrder.size())
                    if (!ranking) {
                        ranking = new ArrayList<String>()
                        rankings.put(index,ranking)
                    }
                    ranking << buildFile
                    flag = true
                    break;
            }

            println("Show␣the␣ranking␣after␣$buildFile:␣$rankings")
            if (flag)
                break;
        }

        if (!flag)
            unranked << buildFile
    }

    // loop through rank keys adding sub lists (TreeMap automatically sorts keySet)
    rankings.keySet().each { key ->
        List<String> ranking = rankings.get(key)
        if (ranking)
            sortedList.addAll(ranking)
    }

    sortedList.addAll(unranked)
```

```
    return sortedList
}

/**
 * Method to create the logical file using SearchPathDependencyResolver
 *  evaluates if it should resolve file flags for resolved dependencies
 */

def createLogicalFile(SearchPathDependencyResolver spDependencyResolver, String buildFile) {

    LogicalFile logicalFile

    if (props.resolveSubsystems && props.resolveSubsystems.toBoolean()) {
        // include resolved dependencies to define file flags of logicalFile
        logicalFile = spDependencyResolver.resolveSubsystems(buildFile,props.workspace)
    }
    else {
        logicalFile = SearchPathDependencyResolver.getLogicalFile(buildFile,props.workspace)
    }

    return logicalFile

}

/*
 * isDev - tests to see if the program is a main program. If the logical file is false, then
 * check to see if there is a file property.
 */
def isDev(LogicalFile logicalFile) {
    boolean isDev = logicalFile.isDev()
    if (!isDev) {
        String devFlag = props.getFileProperty('isDev', logicalFile.getFile())
        String prodFlag = props.getFileProperty('isProd', logicalFile.getFile())
        if (prodFlag && (prodFlag == devFlag || !prodFlag.toBoolean())) {
            logicalFile.setDev(true)
            logicalFile.setProd(false)
        }
        else {
            logicalFile.setDev(true)
            logicalFile.setProd(false)
        }
    }

    return logicalFile.isDev()
}

/*
 * isProd - tests to see if the program is a main program. If the logical file is false, then
 * check to see if there is a file property.
 */
```

```
def isProd(LogicalFile logicalFile) {
    boolean isProd = logicalFile.isProd()
    if (!isProd && !isDev(logicalFile)) {
        logicalFile.setProd(true)
        logicalFile.setDev(false)
    }

    return logicalFile.isProd()
}

/*
 * isMain - tests to see if the program is a main program. If the logical file is false, then
 * check to see if there is a file property.
 */
def isMain(LogicalFile logicalFile) {
    boolean isMain = logicalFile.isMain()
    if (!isMain && logicalFile.isFSUB() == logicalFile.isSUB()) {
        String mainFlag = props.getFileProperty('isMain', logicalFile.getFile())
        if (mainFlag && mainFlag.toBoolean()) {
            logicalFile.setMain(mainFlag.toBoolean())
            logicalFile.setFSUB(false)
            logicalFile.setSUB(false)
        }
        else if (props.getFileProperty('isFSUB', logicalFile.getFile()) == props.getFileProperty('isS
                logicalFile.setMain(true)
                logicalFile.setFSUB(false)
                logicalFile.setSUB(false)
        }
    }

    return logicalFile.isMain()
}

/*
 * isFSUB - tests to see if the program is a fsub program. If the logical file is false, then
 * check to see if there is a file property.
 */
def isFSUB(LogicalFile logicalFile) {
    boolean isFSUB = logicalFile.isFSUB()
    if (!isFSUB && !isMain(logicalFile)) {
        String fsubFlag = props.getFileProperty('isFSUB', logicalFile.getFile())
        if (fsubFlag) {
            logicalFile.setFSUB(fsubFlag.toBoolean())

            if (fsubFlag.toBoolean()) {
                logicalFile.setSUB(false)
                logicalFile.setMain(false)
            }
        }
    }
```

```
    return logicalFile.isFSUB()
}

/*
 * isSUB - tests to see if the program is a sub program. If the logical file is false, then
 * check to see if there is a file property.
 */
def isSUB(LogicalFile logicalFile) {
    boolean isSUB = logicalFile.isSUB()
    if (!isSUB && !isMain(logicalFile)) {
        String subFlag = props.getFileProperty('isSUB', logicalFile.getFile())
        if (subFlag) {
            logicalFile.setSUB(subFlag.toBoolean())

            if (subFlag.toBoolean()) {
                logicalFile.setFSUB(false)
                logicalFile.setMain(false)
            }
        }
    }

    return logicalFile.isSUB()
}

/*
 * isDB2 - tests to see if the program is a DB2 program. If the logical file is false, then
 * check to see if there is a file property.
 */
def isDB2(LogicalFile logicalFile) {
    boolean isDB2 = logicalFile.isDB2()
    if (!isDB2) {
        String db2Flag = props.getFileProperty('isDB2', logicalFile.getFile())
        if (db2Flag)
            logicalFile.setDB2(db2Flag.toBoolean())
    }

    return logicalFile.isDB2()
}

/*
 * isIMS - tests to see if the program is a DL/I program. If the logical file is false, then
 * check to see if there is a file property.
 */
def isIMS(LogicalFile logicalFile) {
    boolean isIMS = logicalFile.isIMS()
    if (!isIMS) {
        String imsFlag = props.getFileProperty('isIMS', logicalFile.getFile())
        if (imsFlag)
            logicalFile.setIMS(imsFlag.toBoolean())
    }
```

```
    return logicalFile.isIMS()
}
```

D.2. BindUtilities.groovy

```
// Make the DB2 BIND PACKAGE CLIST and have it copied to your DBRMhlq
def makeBindPackage(String file, String dbrmHLQ, String workDir, String SUBSYS, String OWNER, String
    // define local properties
    def MEMBER = CopyToPDS.createMemberName(file)
    def COLLID = "${SUBSYS}.DEFAULT"
    def LIB = dbrmHLQ.split('\\.', 2)[1]

    def clistPDS = "${dbrmHLQ}.CLIST.$MEMBER"
    def srcOptions = "cyl␣space(1,1)␣lrecl(80)␣dsorg(PO)␣recfm(F,B)␣dsntype(library)␣msg(1)"

    // create PACKAGE CLIST if necessary
    def clistBindPackage = new File("${workDir}/${MEMBER}_package.clist")
    if (clistBindPackage.exists())
        clistBindPackage.delete()

    clistBindPackage << """
␣␣␣␣␣␣␣␣DSN␣SYSTEM($SUBSYS)␣RETRY(10)
␣␣␣␣␣␣␣␣BIND␣PACKAGE($COLLID)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣LIBRARY($LIB)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣OWNER($OWNER)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣MEMBER($MEMBER)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣DEFER(PREPARE)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣KEEPDYNAMIC(NO)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣NOREOPT(VARS)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣RELEASE(DEALLOCATE)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣CURRENTDATA(NO)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣QUALIFIER($QUAL)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣DEGREE(1)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣VALIDATE(BIND)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣ISOLATION(CS)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣EXPLAIN(YES)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣PATH(SIDUDT,SIDFUN,SIDPROC,SYSIBM,SYSFUN,SYSPROC,USER)
␣␣␣␣␣␣␣␣␣␣END
␣␣␣␣""".toString()

    // Create CLIST PDS if necessary
    new CreatePDS().dataset(clistPDS).options(srcOptions).create()

    // Save CLIST to PDS
    new CopyToPDS().file(clistBindPackage).dataset(clistPDS).member("PACKAGE").execute()

    return "$clistPDS(PACKAGE)"
}

// Make the DB2 BIND PLAN CLIST and have it copied to your DBRMhlq
def makeBindPlan(String file, String dbrmHLQ, String workDir, String SUBSYS) {
    // define local properties
    def MEMBER = CopyToPDS.createMemberName(file)
```

```
    def clistPDS = "${dbrmHLQ}.CLIST.$MEMBER"
    def srcOptions = "cyl␣space(1,1)␣lrecl(80)␣dsorg(PO)␣recfm(F,B)␣dsntype(library)␣msg(1)"

    // create PLAN CLIST if necessary
    def clistBindPlan = new File("${workDir}/${MEMBER}_plan.clist")
    if (clistBindPlan.exists())
        clistBindPlan.delete()

    clistBindPlan << """
␣␣␣␣␣␣␣␣DSN␣SYSTEM($SUBSYS)␣RETRY(10)
␣␣␣␣␣␣␣␣BIND␣PLAN($MEMBER)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣OWNER(****)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣ACTION(REPLACE)␣RETAIN␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣DISCONNECT(EXPLICIT)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣PATH(SIDUDT,SIDFUN,SIDPROC,SYSIBM,SYSFUN,SYSPROC,USER)␣␣␣␣␣␣+
␣␣␣␣␣␣␣␣␣␣␣␣PKLIST(*.DEFAULT.*)
␣␣␣␣␣␣␣␣END
␣␣␣␣""".toString()

    // Create CLIST PDS if necessary
    new CreatePDS().dataset(clistPDS).options(srcOptions).create()

    // Save CLIST to PDS
    new CopyToPDS().file(clistBindPlan).dataset(clistPDS).member("PLAN").execute()

    return "$clistPDS(PLAN)"
}
```

E

Properties bestanden (application-conf)

E.1. application.properties

```
# Build properties used by zAppBuild/build.groovy
# Even testen of de encoding werkt

#
# Run zUnit Tests
# Defaults to "false", to enable, set to "true"
#runzTests=true

#
# Comma separated list of additional application property files to load
# Supports both relative path (to ${application}/application-conf/) and absolute path
# Add the different property files such as file.properties and PLI.properties
# ex. applicationPropFiles=file.properties,PLI.properties
applicationPropFiles=file.properties,Assembler.properties,PLI.properties,Transfer.properties,Cobol.p

#
# Comma separated list all source directories included in application build. Supports both absolute
# and relative paths.  Relative assumed to be relative to ${workspace}.
# ex: applicationSrcDirs=${application},/u/build/common/copybooks
applicationSrcDirs=${application}

#
# Comma separated list of the build script processing order
# Should not be changed!
buildOrder=MFS.groovy,Cobol.groovy,Assembler.groovy,PLI.groovy,LinkEdit.groovy,DBDgen.groovy,PSBgen.

#
# Comma seperated list of the test script processing order
```

```
# Should not be changed!
testOrder=ZunitConfig.groovy

#
# Flag to log output in table views instead of printing raw JSON data
# See also build-conf/build.properties
# default = false
# formatConsoleOutput=false

#
# The main build branch.  Used for cloning collections for topic branch builds instead
# of rescanning the entire application.
# Should not be changed!
mainBuildBranch=main

#
# The git repository URL of the application repository to establish links to the changed files
# in the build result properties
gitRepositoryURL=https://***@dev.azure.com/****/****/_git/COBOL_DEMO

#
# exclude list used when scanning or running full build
excludeFileList=**/*.properties,**/*.xml,**/*.groovy,**/*.json,**/*.md,**/application-conf/*

#
# comma-separated list of file patterns for which impact calculation should be skipped. Uses glob fi
# sample: skipImpactCalculationList=**/epsmtout.cpy,**/centralCopybooks/*.cpy
#####
#skipImpactCalculationList=

#
# Job card, please use \n to indicate a line break and use \ to break the line in this property file
# Example: jobCard=//RUNZUNIT JOB  ,MSGCLASS=H,CLASS=A,NOTIFY=&SYSUID,REGION=0M
#####
jobCard=

################################################################################
# Build Property management
################################################################################
# zAppBuild allows you to manage default properties and file properties:
# - Documentation on how to override corresponding default build properties can be found at:
#    https://github.com/IBM/dbb-zappbuild/docs/FilePropertyManagement.md

# ### Properties to enable and configure build property overrides using individual artifact properti

# flag to enable the zAppBuild capability to load individual artifact properties files for all indiv
# Note: To only activate loadFileLevelProperties for a group of files, it is recommended to use DBB'
# syntax in application-conf/file.properties instead.
# default: false
loadFileLevelProperties=false
```

```
# Property to enable/disable and configure build property overrides using language configuration ma
# file - languageConfigurationMapping.properties
# If loadFileLevelProperties is set as true above, the properties from the individual artifact prop
# properties from language configuration properties file.
# Note: To only activate loadLanguageConfigurationProperties for a group of files, it is recommended
# syntax in application-conf/file.properties instead.
loadLanguageConfigurationProperties=false

# relative path to folder containing individual artifact properties files
# assumed to be relative to ${workspace}/${application}
propertyFilePath=properties

# file extension for individual artifact properties files
# default: properties
propertyFileExtension=properties

################################################################################
# Dependency Analysis and Impact Analysis configuration
################################################################################

#
# boolean flag to configure the SearchPathDependencyResolver to evaluate if resolved dependencies im
#  the file flags isIMS, isDB2, isMQ when creating the LogicalFile
#
#  default:false
resolveSubsystems=false

#
# SearchPathImpactFinder resolution searchPath configuration
#  list of multiple search path configurations which are defined below
#
# this configuration is used when running zAppBuild with the --impactBuild option
#  to calculate impacted files based on the identified changed files
impactSearch=${linkSearch}

#
# copybookSearch
# searchPath to locate Cobol copybooks
# used in dependency resolution and impact analysis
#
# Please be as specific as possible when configuring the searchPath.
# Alternate configurations:
#
# dependency resolution from multiple repositories / multiple root folders:
# copybookSearch = search:${workspace}/?path=**/copybook/*.cpy
#
# dependency resolution across all directories in build workspace, but filtering on the file extensi
 copybookSearch = search:${workspace}/?path=**/*.cpy

# dependency resolution across all directories in build workspace, but filtering on the file extensi
# copybookSearch = search:${workspace}/?path=**/*.cpy;**/*.cobcpy
```

```
#
# dependency resolution in the application directory and a shared common copybook location:
# copybookSearch = search:${workspace}/?path=${application}/copybook/*.cpy;/u/build/common/copybooks
#
# More samples can be found along with the syntax for the search path configurations at:
# https://www.ibm.com/docs/en/dbb/2.0.0?topic=apis-dependency-impact-resolution#6-resolving-logical-
#
#####
#copybookSearch = search:${workspace}/?path=${application}/copybook/*.cpy

#
# pliincludeSearch
# searchPath to locate PLI include files
# used in dependency resolution and impact analysis
#####
pliincludeSearch = search:${workspace}/?path=${application}/plinc/*.cpy

# asmMacroSearch
# searchPath to locate Assembler macro files
# use category filters on what you want to include during the scan (i.e. excludes macro-def keyword)
# used in dependency resolution and impact analysis
#####
asmMacroSearch = search:[SYSLIB:MACRO]${workspace}/?path=${application}/maclib/*.mac

# asmCopySearch
# searchPath to locate Assembler copy files
# used in dependency resolution and impact analysis
#####
asmCopySearch = search:[SYSLIB:COPY]${workspace}/?path=${application}/maclib/*.mac

#
# rexxCopySearch
# searchPath to locate rexx copy - defaults to the local rexx folder in the main application folder
# used in dependency resolution and impact analysis
#####
#rexxCopySearch = search:[SYSLIB:COPY]${workspace}/?path=${application}/rexx/*.rexx

#
# linkSearch
#
# searchPath to locate impacted linkcards or main programs after an included submodule is changed
# leverages the output collection, which has the dependency info from the executable
# category LINK only; used only in impact analysis
#
# Additional samples:
#
# impact resolution across all directories in build workspace, but filtering on the file extension
# staticLinkSearch = search:[:LINK]${workspace}/?path=**/*.cbl
#
# impact resolution across all directories in build workspace, but filtering on the file extension
# staticLinkSearch = search:[:LINK]${workspace}/?path=**/*.cbl,**/*.pli
```

```
#
# More samples can be found along with the syntax for the search path configurations at:
# https://www.ibm.com/docs/en/dbb/2.0.0?topic=apis-dependency-impact-resolution#6-resolving-logical-
#
# Special case with Dependency Scanner Transfer Control Statement capturing turned on (default is o
# the scanners detect a static call to the literal, which would need to turn into a new rule for CAL
# staticCallSearch = search:[:CALL]${workspace}/?path=${application}/cobol/*.cbl
#
linkSearch = search:[:LINK]${workspace}/?path=${application}/assembler/*.asm

# zunitTestConfigSearch
# searchPath to locate zunit config files
# used in impact analysis
#####
#zunitTestConfigSearch = search:[:ZUNITINC]${workspace}/?path=${application}/cobol/*.cbl;${applicati

#
# zunitPlayfileSearch
# searchPath to locate zunit playback files
# used in dependency resolution
#####
#zunitPlayfileSearch = search:[SYSPLAY:]${workspace}/?path=${application}/testplayfiles/*.bzuplay

#
# zunitTestcasePgmSearch
# searchPath to locate impacted test case programs
# see also build-conf/build.properties -> createTestcaseDependency
# used in impact analysis
#####
#zunitTestcasePgmSearch = search:[SYSPROG:PROGRAMDEPENDENCY]${workspace}/?path=${application}/cobol/
```

E.2. file.properties

```
# Application script mappings and file property overrides

#
# Script mappings for all application programs
dbb.scriptMapping = Assembler.groovy :: **/*.asm
dbb.scriptMapping = MFS.groovy :: **/*.mfs
dbb.scriptMapping = PSBgen.groovy :: **/psb/*.asm
dbb.scriptMapping = DBDgen.groovy :: **/dbd/*.asm
dbb.scriptMapping = Cobol.groovy :: **/*.cbl
dbb.scriptMapping = LinkEdit.groovy :: **/*.lnk
dbb.scriptMapping = PLI.groovy :: **/*.pli
dbb.scriptMapping = ZunitConfig.groovy :: **/*.bzucfg
dbb.scriptMapping = Transfer.groovy :: **/*.jcl, **/*.xml

#
# General file level overwrites through DBB Build Properties, you can add more files
# Usage of these properties:
#   -If you want all .pli files in your application directory to be mains
#       - isMain = true :: **/*.pli
#   -If you also have assembler files in your application directory that you want to be mains
#       - isMain = true :: **/*.pli, **/*.asm
#   -If you only want specific .pli files that have "str" in their name to be mains
#       - isMain = true :: **/*str*.pli
#
#   -These usage steps are applicable to all properties below...
#
# Environment, Development or Production: (Default is Development)
isDev = true :: **/*.pli, **/*.asm, **/*.cbl
#isProd = true :: **/*.cbl
#
# What module?
isMain = true :: **/*.pli, **/*.asm, **/tstcbl1.cbl
isFSUB = true :: **/tstdb2.cbl, **/tstfsub1.cbl
isSUB = true :: **/tstsub1.cbl
#
# Has IMS?
isIMS = true :: **/tstpli2.pli, **/tstpli3.pli, **/tstpli4.pli
#isIMS = false :: **/app.asm
#
# Has DB2?
isDB2 = true :: **/tstpli2.pli, **/tstpli3.pli, **/tstpli4.pli, **/tstdb2.cbl, **/tstsub1.cbl, **/ts
#isDB2 = false :: **/app.pli

#
# DB2 Bind package and plan variables this includes qualifier and subsystem
# Defaults look at the isDev and isProd properties, this means no DB2A if not
# specified below, it is recommended to NOT have a general qualifier like **/*.pli
# ex. DB2qual = ##T :: **/tstpli1.pli
#     DB2subsys = DB2O :: **/mainpli.pli
```

```
#
# Execute DB2 bind
generateDb2BindInfoRecord = true :: **/tstpli2.pli, **/tstpli3.pli, **/tstpli4.pli, **/tstdb2.cbl, *
#
# DB2 Bind max RC
db2Bind_maxRC = 4
#
# DB2 qualifier
db2Qual = ##T :: **/tstpli2.pli, **/tstpli3.pli, **/tstpli4.pli, **/tstdb2.cbl, **/tstsub1.cbl
#
# DB2 subsystem
db2Subsys = DB2O :: **/tstpli2.pli, **/tstpli3.pli, **/tstpli4.pli, **/tstdb2.cbl, **/tstsub1.cbl
#
# optional DB2 bind parameters
# Set the owner instead of having your own ID as owner
#db2Owner = ***

#
# file mapping for generated zUnit Test case programs (Cobol) to use a seperate set of libraries
######
# cobol_testcase = true :: **/testcase/*.cbl

# file mapping for generated zUnit Test case programs (PL/I) to use a seperate set of libraries
######
# pli_testcase = true :: **/testcase/*.pli

# mapping for overwriting the impactResolution rules in application.properties
######
# impactResolutionRules=[${copybookRule},${linkRule}] :: **/copy/*.cpy,**/cobol/*.cbl

#
# PropertyMapping to map files using the Transfer.groovy language script to different target dataset
#####
transfer_datasetMapping = transfer_jclPDS :: **/*.jcl
transfer_datasetMapping = transfer_xmlPDS :: **/*.xml
#
# file mapping for overwriting the default deployType of the Transfer.groovy language script
#####
transfer_deployType = JCL :: **/*.jcl
transfer_deployType = XML :: **/*.xml
```

E.3. Cobol.properties

```
# Application properties used by zAppBuild/language/Cobol.groovy

#
# default COBOL program build rank - used to sort language build file list
# leave empty - overridden by file properties if sorting needed
cobol_fileBuildRank=
cobol_fileBuildOrder=SUB,FSUB,MAIN

#
# COBOL dependencySearch configuration
# searchPath defined in application.properties
cobol_dependencySearch=${copybookSearch}

#
# default COBOL compiler version
# can be overridden by file properties
cobol_compilerVersion=V6

#
# default COBOL maximum RCs allowed
# can be overridden by file properties
cobol_compileMaxRC=4
cobol_linkEditMaxRC=4

#
# lists of properties which should cause a rebuild after being changed
#cobol_impactPropertyList=cobol_compilerVersion,cobol_compileParms
#cobol_impactPropertyListSQL=cobol_compileSQLParms

#
# default COBOL compiler parameters
# can be overridden by file properties
cobol_extraCompileParms=
cobol_compileErrorPrefixParms=ADATA,EX(ADX(ELAXMGUX))
cobol_compileDebugParms=TEST

#
# default LinkEdit parameters can be found in Cobol.properties (zAppBuild/build-conf)
# can be overridden by file properties
# override parameters or add additional parameters
# -----PARMS FOR MAIN & FSUB PROGRAMS-----
# cobol_linkEditParms=LET,,,STRIPSEC,LIST(NOIMP),,RENT,DYNAM(DLL),CASE(MIXED),COMPAT(PM5)
# -----ONLY FOR SUB PROGRAMS-----
# cobol_linkEditParmsSUB=NOCALL,LIST,LET,,,AMODE(31),RMODE(ANY),RENT,CASE(MIXED),COMPAT(PM5)

# Optional linkEditStream defining additional link instructions via SYSIN dd
cobol_linkEditStream=
#INCLUDE SYSLIB(IMPORTS)
```

```
# If using a debug exit for IBM Debug tool, provide the SYSIN instream DD which is appended to SYSIN
# Samp: cobol_linkDebugExit=     INCLUDE SYSLIB(EQAD3CXT) \n
cobol_linkDebugExit=

#
# execute link edit step
# can be overridden by file properties
cobol_linkEdit=true

#
# store abbrev git hash in ssi field
# available for buildTypes impactBuild, mergeBuild and fullBuild
# can be overridden by file properties
cobol_storeSSI=false

#
# flag to generate IDENTIFY statement during link edit phase
# to create an user data record (IDRU) to "sign" the load module with
# an identify String: <application>/<abbreviatedGitHash>
# to increase traceability
#
# can be overridden by file properties
# default: false
cobol_identifyLoad=false

#
# default deployType
cobol_deployType=LOAD

#
# deployType for build files with isIMS=true
cobol_deployTypeDLI=IMSLOAD

#
# scan link edit load module for link dependencies
# can be overridden by file properties
cobol_scanLoadModule=true

#
# additional libraries for compile SYSLIB concatenation, comma-separated
cobol_compileSyslibAddConcat=AMGHENT.LOC.LIB.PARMLIB

#
# additional libraries for linkEdit SYSLIB concatenation, comma-separated
cobol_linkEditSyslibAddConcat=AMGHENT.LOC.LIB.PARMLIB

# additional libraries for the linkEdit ALTLIB concat (comma-separated), for when you don't want
# those libraries to be in your SYSLIB
cobol_linkEditAddlibConcat=
```

E.4. .gitattributes

```
# Bestand voor de encoding van verschillende bestanden te configureren


# line endings
* text eol=lf

# git's files
.gitattributes zos-working-tree-encoding=iso8859-1
.gitignore zos-working-tree-encoding=iso8859-1

# file encodings
*.txt zos-working-tree-encoding=ibm-1148
*.groovy zos-working-tree-encoding=ibm-1148
*.properties zos-working-tree-encoding=ibm-1148
*.asm zos-working-tree-encoding=ibm-1148
*.md zos-working-tree-encoding=ibm-1148
*.log zos-working-tree-encoding=ibm-1148
*.html zos-working-tree-encoding=ibm-1148
*.json zos-working-tree-encoding=ibm-1148
*.jcl zos-working-tree-encoding=ibm-1148
*.pli zos-working-tree-encoding=ibm-1148
*.cbl zos-working-tree-encoding=ibm-1148
```

F

Bash scripts Azure Pipelines

F.1. AzRocketGit-init.sh (origineel)

```
#! /bin/sh
## Az/DBB Demo- Git clone v1.2 (njl)
. $HOME/.profile
MyRepo=$1
MyWorkDir=$2 ; mkdir -p $MyWorkDir ; cd $MyWorkDir
branch=$3
# Strip the prefix of the branch name for cloning branch=${branch##*"refs/heads/"}

workSpace=$(basename $MyRepo) ;workSpace=${workSpace%.*}
echo "************************************************************"
echo "**          Started: ------- Rocket-Git Clone on HOST/USER: $(uname -la)/$USER"
echo "**                                                  MyRepo: " $MyRepo
echo "**                                                  MyWorkDir: " $PWD
echo "**                                                  workSpace: " $workSpace
echo "**                                                  branch: " $3 "->" $branch
git clone -b $branch $MyRepo 2>&1
cd $workSpace

echo "Show status"
git status

echo "Show all Refs"
git show-ref
```

(IBM, 2021b)

F.2. AzDBB-build.sh (origineel)

```
#! /bin/sh
## AzDBB Demo- DBB Build v1.2 (njl)
. $HOME/.profile
MyWorkDirr=$1 ; cd $MyWorkDir
MyWorkSpace=$2
MyApp=$3
BuildMode"=$4 "$5 #DBB Build modes: --impactBuild, --reset, --fullBuild, '—fullBuild –
'scanOnly
zAppBuild=$HOME/dbb-zappbuild/build.groovy
echo "***************************************************************"
echo "**␣␣␣␣␣␣␣␣Started:␣␣␣␣␣␣␣␣DBB␣Build␣on␣HOST/USER:␣$(uname␣-Ia)/$USER"
echo "**␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣MyWorkDir:" $PWD
echo "**␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣MyWorkspace:" $MyWorkSpace
echo "**␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣MyApp:" $MyApp
echo "**␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣DBB␣Build␣Mode:" $BuildMode
echo "**␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zAppBuild␣Path:" $zAppBuild
echo "**␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣DBB_HOME:" $DBB_HOME
echo "**␣"
echo "␣**␣␣␣␣␣␣␣␣Git␣Status␣for␣MyWorkSpace:"
git -C $MyWorkSpace status
groovyz $zAppBuild --workspace $MyWorkSpace --application $MyApp -outDir . --hlq $USER --logEncoding
if [ "$?" -ne "0" ]; then
echo "DBB␣Build␣Error.␣Check␣the␣build␣log␣for␣details"
exit 12
fi

## Except for the reset mode, check for "nothing to build" condition and throw an error to stop pipe
if [ "$BuildMode" != "--reset" ]; then
buildlistsize=$(wc -c < $(find . -name buildList.txt))
if [ $buildlistsize = 0 ]; then
echo "***␣␣␣␣␣␣␣␣Build␣Error:␣No␣source␣changes␣detected.␣RC=12"
exit 12
fi
else
echo "***␣DBB␣Reset␣completed"
fi
exit 0
```

(IBM, 2021b)

F.3. AzRocketGit-init.sh (aangepast)

```
#! /bin/sh
## Az/DBB Demo- Git clone v1.2 (njl)
. $HOME/.profile
MyRepo=$1
MyWorkDir=$2 ; mkdir -p $MyWorkDir ; cd $MyWorkDir
branch=$3
# Strip the prefix of the branch name for cloning
branch=${branch##*"refs/heads/"}
workSpace=$(basename $MyRepo) ;workSpace=${workSpace%.*}
echo "************************************************************"
echo "** Started: Rocket-Git Clone on HOST/USER: $(uname -la)/$USER"
echo "** MyRepo:" $MyRepo
echo "** MyWorkDir:" $PWD
echo "** workSpace:" $workSpace
echo "** branch:" $3 "->" $branch

git clone -b $branch $MyRepo 2>&1
cd $workSpace

echo "Show status"
git status
echo "Show all Refs"
git show-ref
```

F.4. AzDBB-build.sh (aangepast)

```
#! /bin/sh
## Az/DBB Demo- DBB Build v1.2 (njl)
. $HOME/.profile
MyWorkDir=$1 ; cd $MyWorkDir
MyRepo=$2
MyApp=$3
BuildMode="$4 $5" #DBB Build modes: --impactBuild, --reset, --fullBuild, '--fullbuild --scanOnly'
zAppBuild=$DBB_HOME/dbb-zappbuild/build.groovy
MyWorkSpace=$(basename $MyRepo) ;MyWorkSpace=${MyWorkSpace%.*}
echo "************************************************************"
echo "** Started: DBB Build on HOST/USER: $(uname -la)/$USER"
echo "** MyWorkDir:" $PWD
echo "** MyWorkspace:" $MyWorkSpace
echo "** MyApp:" $MyApp
echo "** DBB Build Mode:" $BuildMode
echo "** zAppBuild Path:" $zAppBuild
echo "** DBB_HOME:" $DBB_HOME
echo "**"
echo " ** Git Status for MyWorkSpace:"
git -C $MyWorkSpace status


groovyz $zAppBuild --workspace $MyWorkSpace --application $MyApp -outDir . --hlq $USER --logEncoding
if [ "$?" -ne "0" ]; then
 echo "DBB Build Error. Check the build log for details"
 if [ -d "$MyWorkSpace" ]; then
    rm -r "$MyWorkSpace"
 fi
 if [ -d "/etc/dbb/load" ]; then
    rm -r "/etc/dbb/load"
 fi
 exit 12
fi
## Except for the reset mode, check for "nothing to build" condition and throw an error to stop pipe
if [ "$BuildMode" != "--reset" ]; then
buildlistsize=$(wc -c < $(find . -name buildList.txt))
if [ $buildlistsize = 0 ]; then
 echo "*** Build Error: No source changes detected. RC=12"
 if [ -d "$MyWorkSpace" ]; then
    rm -r "$MyWorkSpace"
 fi
 if [ -d "/etc/dbb/load" ]; then
    rm -r "/etc/dbb/load"
 fi
 exit 12
fi
else
echo "*** DBB Reset completed"
fi
```

```
echo "***␣Cleaning␣workspace␣***"
if [ -d "$MyWorkSpace" ]; then
    rm -r "$MyWorkSpace"
fi
if [ -d "/etc/dbb/load" ]; then
    rm -r "/etc/dbb/load"
fi
exit 0
```

Bibliografie

AWS. (g.d.). *Start Building on AWS Today* (AWS, Red.). Verkregen maart 6, 2024, van https://aws.amazon.com/

Bloomberg, J. (2021, april 9). *How the mainframe became a surprising platform for innovation*. Verkregen december 10, 2023, van https://siliconangle.com/2021/04/09/mainframe-became-surprising-platform-innovation/

Broadcom. (2024, februari 12). *Modernize Mainframe Development Processes and Tooling* (Broadcom, Red.). https://mainframe.broadcom.com/devops

Ceruzzi, P. E. (2003, april 8). *A History of Modern Computing second edition*.

CH, M. (2024, maart 6). *CopyCats Competitors* (M. CH, Red.). Verkregen maart 6, 2024, van https://www.computerhistory.org/revolution/mainframe-computers/7/169

Daniel Sokolowski, G. S., Pascal Weisenburger. (2021). Automating Serverless Deployments for DevOps Organizations. *Automating Serverless Deployments for DevOps Organizations*, 57–69. https://doi.org/10.1145/3468264.3468575

Elliot, J. (2015, augustus 12). *History and Evolution of IBM Mainframes over SHARE's 60 Years* (J. Elliot, Red.). Verkregen maart 7, 2024, van https://docplayer.net/46901393-History-and-evolution-of-ibm-mainframes-over-share-s-60-years.html

Fujitsu. (g.d.). *BS2000 Mainframes* (Fujitsu, Red.). Verkregen maart 6, 2024, van https://www.fujitsu.com/emeia/products/computing/servers/mainframe/bs2000/

Google. (g.d.). *The new way to cloud starts here* (Google, Red.). Verkregen maart 6, 2024, van https://cloud.google.com/

IBM. (g.d.-a). *Drive your business with hybrid cloud* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/cloud/why-ibm

IBM. (g.d.-b). *Find a product* (IBM, Red.). Verkregen maart 7, 2024, van https://www.ibm.com/products?ibmTopics[0][0]=cat.topic:ITInfrastructure

IBM. (g.d.-c). *IBM Z* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/z

IBM. (2021a). *IBM Dependency Based Build Course; DBB Architecture Overview* (IBM, Red.). Verkregen maart 7, 2024, van https://mediacenter.ibm.com/media/IBM+Dependency+Based+Build+CourseB+DBB+Architecture+Overview/1_q8mwed69

IBM. (2021b, maart). *Azure DevOps and IBM Dependency Based Build Integration* (T. D. Nelson Lopez Shabbir Moledina, Red.). Verkregen april 29, 2024, van

https://www.ibm.com/support/pages/system/files/inline-files/Azure-DBB%20Integration%20v2g.pdf

IBM. (2021c, juni 1). *Program Diretory for IBM Dependency Based Build* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/docs/en/dbb/1.0?topic=zos-program-directory-dependency-based-build#HDRZJUMP1

IBM. (2022, maart 15). *What is new and noteworthy* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/docs/en/dbb/1.0?topic=what-is-new-noteworthy

IBM. (2023a, mei 9). *IBM Dependency Based Build overview* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/docs/en/dbb/2.0?topic=dependency-based-build-overview

IBM. (2023b, mei 19). *What is new and noteworthy* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/docs/en/dbb/1.0?topic=what-is-new-noteworthy

IBM. (2023c, december 14). *What is new and noteworthy* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/docs/en/dbb/1.1?topic=what-is-new-noteworthy

IBM. (2024a, maart 6). *The IBM 700 Series* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/history/700

IBM. (2024b, maart 6). *The IBM System/360* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/history/system-360

IBM. (2024c, maart 6). *The IBM System/370* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/history/system-370

IBM. (2024d, maart 6). *IBM zSystem* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/history/eserver-zseries

IBM. (2024e, maart 6). *What is a mainframe? It's a style of computing* (IBM, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/docs/en/zos-basic-skills?topic=today-what-is-mainframe-its-style-computing

LaMonica, M. (2004, april 7). *Mainframe competitors aim at Big Blue target* (M. LaMonica, Red.). CNET. Verkregen maart 6, 2024, van https://www.cnet.com/tech/tech-industry/mainframe-competitors-aim-at-big-blue-target/

Lopez, N. (2023, juni). *Git for Mainframe Developers* (N. Lopez, Red.). Versie 1.4. Verkregen maart 7, 2024, van https://www.ibm.com/support/pages/system/files/inline-files/Git%20training%20for%20Mainframers_2.pdf

Microsoft. (2022, april 10). *What is source control?* (Microsoft, Red.). Verkregen maart 7, 2024, van https://learn.microsoft.com/en-us/azure/devops/user-guide/source-control?view=azure-devops

Microsoft. (2023, december 4). *What is Azure Pipelines?* (Microsoft, Red.). Verkregen maart 7, 2024, van https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops

Microsoft. (2024a, januari 4). *What is Azure DevOps?* (Microsoft, Red.). Verkregen maart 7, 2024, van https://learn.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops?view=azure-devops

Microsoft. (2024b, april 3). *Self-hosted Windows agents* (Microsoft, Red.). Verkregen april 29, 2024, van https://learn.microsoft.com/en-us/azure/devops/pipelines/agents/windows-agent?view=azure-devops

Microsoft. (2024c, april 5). *Create and manage agent pools* (Microsoft, Red.). Verkregen april 29, 2024, van https://learn.microsoft.com/en-us/azure/devops/pipelines/agents/pools-queues?view=azure-devops&tabs=yaml,browser

Microsoft. (2024d, april 22). *Create an organization* (Microsoft, Red.). Verkregen april 29, 2024, van https://learn.microsoft.com/en-us/azure/devops/organizations/accounts/create-organization?view=azure-devops

Moorhead, P. (2022, april 5). *IBM Z16 – The Mainframe Is Dead, Long Live The Mainframe*. Verkregen december 10, 2023, van https://www.forbes.com/sites/patrickmoorhead/2022/04/05/ibm-z16--the-mainframe-is-dead-long-live-the-mainframe/?sh=60b91a16ee9d

Oracle. (g.d.). *Engineered systems* (Oracle, Red.). Verkregen maart 6, 2024, van https://www.oracle.com/engineered-systems/

Porter, K. (2019, juni 7). *z/OS development with IBM Dependency Based Build and IDz*. Verkregen december 10, 2023, van https://kalebporter.medium.com/modern-z-os-development-with-ibm-dependency-based-build-9cf945d72bfb

Society, A. C. (2017, juni 13). *ACS Heritage Project: Chapter 26* (G. Philipson, Red.). Verkregen maart 6, 2024, van https://ia.acs.org.au/article/2017/ACS-Heritage-Project--Chapter-26.html

Tozzi, C. (2022, november 17). *Where Will Mainframes be in Ten Years?* Verkregen december 10, 2023, van https://www.precisely.com/blog/mainframe/mainframe-industry-ten-years

Unisys. (g.d.). *Enable mission-critical, high-volume transaction processing* (Unisys, Red.). Verkregen maart 6, 2024, van https://www.unisys.com/solutions/enterprise-computing/clearpath-forward/

VMWare. (g.d.). *Solutions for Your Hybrid Cloud Business* (VMWare, Red.). Verkregen maart 6, 2024, van https://www.ibm.com/cloud/why-ibm