## Hoofdstuk 1: Wat is DevOps?

1. Leg Continuous delivery/deployment uit m.b.v. een figuur.
2. Wat zijn de verschillende tools om DevSecOps te implementeren?

## Hoofdstuk 2: Container technology

1. Wat is het verschil tussen Hardware- en OS-level virtualisatie, a.d.h.v. een figuur.
2. Leg uit omgevingsonafhankelijkheid en hoe laat containertechnologie toe om omgevingsonafhankelijkheid aan u te geven als programmeur?
3. Wat is het rol van verschillende besturingssysteemlagen in containers?
4. Bespreek container versus image.
5. Hoe zit dat met containers waarvan lagen worden gedeeld?
6. Op basis van welk principe gaat uw container?
7. Wat weet je over OCI?
8. Wat weet je over CNI?
9. Leg het huidige containerlandschap uit.

## Hoofdstuk 3: Kubernetes

1. Beschrijf Kubernetes.
2. Leg Kubernetes key concept: PODS uit a.d.h.v. een figuur.
3. Leg de Kubernetes Architectuur uit.
4. Wat is Canary Release an A/B testing?

## Hoofdstuk 4: CNCF

1. Wat weet je over HELM?
2. Wat zijn Kubernetes Operators?
3. Wat zijn Vitess Features en wat kan dit doen?
4. Leg uit wat TiKV en ETCD is en wanneer je welke gebruikt.
5. Wat weet je over ENVOY?

## Hoofdstuk 1: Wat is DevOps?

|  |
| --- |
| **Continuous delivery/deployment :**  Continue integratie: builden en testen.  Continue delivery: implementeren en integratietesten.  Continue deployment: builden, testen, implementeren en alles brengen naar productie.  Best practices:   * Builds moeten koffietesten doorstaan (minder dan 5 minuten) * Kleine stukjes code commiten * De build mag niet broken achtergelaten worden * Eerst implementeren naar een kopie van productie, dan pas naar productie. * Niet verder implementeren als vorige stap mislukt. |

|  |
| --- |
| **DevSecOps:** DevSecOps zorgt ervoor dat veiligheid een gedeelde verantwoordelijkheid is van het hele team, en automatisch wordt meegenomen in elke stap van de softwarelevenscyclus.    **SAST – Static Application Security Testing**   * Scan eigen/aangepaste code op programmeerfouten of intwerpflaws die kunnen leiden tot kwetsbaareden die misbruikt kunnen worden. * Voornamelijk gebruikt tijdens de code-, build- en ontwikkelingsfasen can de levenscyclus van de software.   **SCA – Software Component Analysis**   * Scan broncode op binaire bestanden op bekende kwetsbaarheden in open source en externe componenten. * Biedt inzicht op beveiligings- en licentierisico’s om prioritization en remediation efforts te versnellen. * Kan naadloos worden geïntegreerd in de CI/CD-proces om continu nieuwe kwetsbaarheden in open source te detecteren, van integratie bij de bouw tot de preproductie-release.   **IAST – Interactive Application Security Testing**   * Werkt op de achtergrond tijdens handmatige of geautomatiseerde functionele tests, analyseert het runtime-gedrag van webapplicaties. * Gebruikt instrumentation om de request/response interactie, gedrag en dataflow te observeren. * Detecteer runtime-vulnirabilities en herhaalt en test deze bevindingen automatisch, waardoor ontwikkelaard inzicht krijgen tot aan de regel code waar deze bevinden.   **DAST – Dynamic Application Security Testing**   * Geautomatiseerde blackbox-testtechnologie die nabootsts hoe een hanker zou interageren met uw webapplicatie of API. Testen gebeurt via een netwerkverbinding. * Vereist geen toegang tot de broncode of aanpassingen om de applicatiestapel te scannen. |

## Hoofdstuk 2: Container Technology

|  |
| --- |
| **Hardware virtualisatie:**   * Hypervisor beheert toegang tot gedeeld hardware. * VM’s zijn volledig geïsoleerd. * Elke VM vereist eigen besturingssysteem.   **OS-level virtualisatie:**   * De hostkernel staat toe dat er meerder procesruimtes zijn. * Lichtgewicht: containers dele de host-os-kernel. * Elke container heeft zijn eigen root file system. |

|  |
| --- |
| **Docker namespaces/Environment Independance**  Elk draaiend programma of proces op een Linux-machine heeft een uniek nummer, een procesidentifier.  Een PID-namespace is een verzameling unieke nummers die processen identificeert.   * Een host kan meerdere PID-namespaces hebben. * Elk PID namespace kan zijn eigen PID 1, 2, 3, … bevatten. * Docker creëert standaard een nieuwe PID namespace voor elke container, waarbij processen in die container worden geïsoleerd van processen in andere containers.   Bijvoorbeeld:   * Voer NGINX rechtstreeks op je computer uit. * Stel je voor dat je NGINX al hebt gestart en start een andere instantie (in dezelfde container). * Het tweede proces kan niet bij de benodigde bronnen komen omdat het eerst proces deze al gebruikt (port conflict). * Een veel voorkomend probleem bij implementatie op echte systemen.   Omgevingsafhankelijkheid stelt software in staat om afhankelijkheden te hebben van schaarse systeembronnen, ongeacht de behoeften van andere software die op dezelfde locatie draait.   * Binnen dezelfde netwerkpoort, gebruik van dezelfde tijdelijk bestandsnaam, verschillende versies van een globale bibliotheek, gebruikt van dezelfde omgevingsvariabelen. * Docker gebruikt namespaces, resource limits, filesystem roots en geritualiseerde netwerkcomponenten om dit achter de schermen te regelen. |

|  |
| --- |
| **Rol van OS layers in Containers:**  Container Host of Host OS:   * Het besturingssysteem waarop de Docker-client en Docker-daemon draaien. * In het geval van Linux en niet-Hyper-V containers deelt het Host OS zijn kernel met de draaiende Docker-containers.   Container OS:   * Verwijst naar een image dat een besturingssysteem bevat, zoals Ubuntu, CentOS of windowsservercore. * Over het algemeen bouw je je eigen image boven een BaseOS image, zodat je delen van dat besturingssysteem kan gebruiken. * Windows containers vereisen een BaseOS, terwijl Linux containers dat niet vereisen.   Kernel:   * Beheert functies op lager niveau zoals geheugenbeheer, bestandssysteem, netwerk en procesplanning. |

|  |
| --- |
| Bij het starten van een container vanuit een image, zal de Docker-engine een nieuwe, dunne beschrijfbare containerlaag boven op de laagstapel van het image creëren.   * Alle wijzigingen die worden aangebracht in de draaiende container: het schrijven van bestanden, het wijzigen van bestaande bestanden en het verwijderen van bestanden worden naar deze laag geschreven. * Wanneer de container wordt verwijderd, wordt de beschrijfbare laag ook verwijdert, het onderliggende image blijft ongewijzigd.   Vermijd intensief schrijven naar de containerlaag en gebruik Docker ‘volumes’ om persistente gegevens op te slaan.   * Er is overhead betrokken bij de special opslagdriver die door Docker wordt gebruikt. * Het schrijven naar containerlaag wordt gebruikt tijdens het bouwproces van het image. |

|  |
| --- |
| Omdat elke container zijn eigen beschrijfbare containerlaag heeft en alle wijzigingen in deze laag worden opgeslagen:   * Kunnen meerder containers toegang delen tot dezelfde onderliggende image en toch hun eigen gegevenstoestand hebben. |

|  |
| --- |
| **Linux Control Groups (CGroups)**  Probleem:   * Hoe kan ik een groep taken (processen) beperken, prioriteren, controleren en metingen verkrijgen? * Control Groups * Device access whitelist (apparaten toestaan) * Beperking van resources: geheugen, CPU, apparaten, blok IO, … * Prioriteren: zie krijgt er meer van de CPU, geheugen, … * Accounting: resourcegebruik per groep * Controle: bevriezen en checkpointen * Injectie: packettagging   **Linux Namespaces**  Probleem:   * Hoe kan ik een geïsoleerd beeld van globale bronnen bieden aan een groep taken? * **Namespaces**     **Changing Root**  Probleem:   * Geef elk proces de illusie dat het is gemount vanaf de root-directory. * Het “pivot-root”-commando dupliceert de volledige root-directory in de MNT namespace.   **Secure Containers**  Probleem:   * Vijandig proces kan ontsnappen uit de container omdat die het hele systeem namespaces of containerized heeft. Als de kernel kwetsbaarheden heeft, kan de container hier ook gebruik van maken. * Linux Security Models (LSM) * Mandatory Access Controls: Verplicht toegangscontrole * Definieer mogelijkheden per proces (toegang tot systeemaanroepen) |

|  |
| --- |
| **Open Container Initiative – OCI**  Een project van de Linux Foundation dat open industrie standaarden creëert rond containerformaten en runtimes.  Ontstaan in 2015 onder andere door Docker en CoreOS.  Drie specificaties/normen:   * **Runtime-specificatie** (runtime-spec): * Definieert de configuratie, uitvoeringsomgeving, levenscyclus van een container (de API) * Geimplementeert door runC, gedoneerd door Docker als een laag-niveau container-runtime. * I**mage-specificatie** (image-spec): * Specificatie voor het formaat van containerimages voor het verzenden van software. * **Distributiespecificatie** (distributed-spec): * Specifieert de API over hoe containerimages/inhoud kunnen worden gedistribueerd via onder andere repositories. |

|  |
| --- |
| **Container Networking Interface – CNI**  Een interface tussen container-runtime en de implementatie van het netwerk.   * Ontstaan bij CoreOS als onderdeel van RLT, nu ondersteund door Cloud Native Computing Foundation (CNCF)   Een op plug-in gerichte netwerkoplossing voor containers en container orchestrators   * Wordt gebruikt door Kubernetes, Mesos, Cloud Foundry, enz.     CNI zorgt er dus voor dat uw netwerkinterface gestandaardiseerd wordt, dat de API’s vastliggen, … |

|  |
| --- |
| **Low-level runtime**  Een low-level runtime is in staat om een container op te starten en deze te verbinden met een bestaand netwerk.   * Doet niet: netwerk creëren van een container, container images beheren, omgeving van de container voorbereiden en lokale aanhoudende opslag beheren.   Drie belangrijke benaderingen (allemaal OCI-runtime-specificatie compatibel):   * Runc: command-line tools voor het starten en uitvoeren van op het OS-niveau gevirtualiseerde containers volgends de OCI-specificatie * Kata-runtime: command-line tools voor het starten van hardwaregevirtualiseerde Linux containers volgende de OCI-specificatie. * gVisor: Een gebruikersruimte kernel voor containers ontwikkeld door Google (beperkte kernel boven op de reguliere kernel), met extra beveiliging in vergelijking met containers die rechtstreeks op de Linux-kernel draaien. Focus: beveiliging.   **High-level container runtime**  High-level runtime houdt zich bezig met het creëren van het netwerk van de container, het beheren van container images, het voorbereiden van de omgeving van de container en het beheren van lokale/aanhoudende opslag.  Drie belangrijke benaderingen (CRI-compatibel):   * Containerd: een container-runtime gedoneerd aan de CNCF door Docker. Standaard in veel Kubernetes-distributies. Ondersteunt alle OCI-compatibel runtimes. * CRI-O: een brug tussen Kubernetes en OCI-compatibel runtimes, gemaakt door Red Hat en gelijktijdig uitgebracht met elke Kubernetes-versie. Standaard in OpenShift. * Docker: Docker zelf kan worden gebruikt als een CRI-compatibel container-runtime, maar veel Kubernetes Distributeurs stappen hiervan af vanwege de complexiteit van Docker.   **High-level container Management**  High-level Management houdt zich bezig met het orchestreren van containers op de infrastructuur.  Drie belangrijkste benaderingen:   * Kubernetes en critcl: Maakt het mogelijk om container-runtimes en applicaties op een Kubernetes-node te inspecteren en debuggen. Gebruikt de CRI-standaard van Kubernetes voor het benaderen van container-runtimes. * Docker zelf: Docker zelf kan worden gebruikt voor high-level management. * Podman: Een CLI-tool van Red Hat voor het beheren van pods en containers. Maakt gebruik van libpod, dat op zijn beurt runc gebruikt en compatibel is met Docker-images. |

## Hoofdstuk 3: Kubernetes

|  |
| --- |
| **Kubernetes (K8s)** is een open-source platform voor het automatiseren van de uitrol, het beheer en de schaalbaarheid van containerapplicaties. Het is oorspronkelijk ontwikkeld door Google en nu beheerd door de Cloud Native Computing Foundation (CNCF).  Kubernetes helpt bij:   * **Uitrollen** van containers op een geautomatiseerde manier. * **Schaalbaarheid**: automatisch meer of minder instanties van een applicatie starten afhankelijk van de belasting. * **Zelfherstel**: als een container crasht, wordt die automatisch opnieuw gestart. * **Load balancing**: verdeelt netwerkverkeer over meerdere instanties van een applicatie. * **Configuratiebeheer en secrets**: je kan configuratiebestanden en wachtwoorden veilig beheren. * **Rolling updates:** applicaties updaten zonder downtime.   Belangrijke concepten in Kubernetes  **Pod**   * De kleinste eenheid in Kubernetes. Een pod bevat meestal 1 container, soms meerder die samen moeten draaien.   **Node**   * Een fysieke of virtuele machine waarop pods draaien   **Cluster**   * Een verzameling nodes die samen een omgeving vormen waarin Kubernetes workloads beheert.   **Deployment**   * Zorgt voor declaratieve updates van pods.   **Service**   * Zorgt voor netwerktoegang tot pods. Handig voor load balancing of externe toegang.   **Ingress**   * Beheert externe toegang tot services, vaak via HTTP/HTTPS.   **Namespace**   * Logische scheiding van resources binnen een cluster. |

|  |
| --- |
| Pods zijn een atoomeenheid of kleinste eenheid van werk in Kubernetes.  Pods zijn 1 of meerdere containers die volumes en een netwerknamespace delen en deel uitmaken van een enkele context.   * Kubernetes voert containers niet direct uit, maar alleen pods.   Pods fungeren als een eenheid van implementatie, horizontale schaling en replicatie.  Pods zijn:   * REST-objecten * Ephemeral (ze kunnen op elk moment worden vernietigd en hebben geen vaste netwerkadressen). * Kunnen labels hebben die attributen specifiëren die zinvol zijn voor gebruikers.   Een Kubernetes-implementatie biedt hogere declaratieve updates voor pods.   * Beschrijf de gewenste staat in termen van onder andere het aantal kopieën en vereiste fouttolerantie.   **Multi-Container pods**  Het een-container-per-pod model is het meest voorkomende Kubernetes-gebruikersscenario.   * Denk aan een pod als een omhulsel rond een enkele container en Kubernetes beheert pods in plaant van containers rechtstreeks.   In sommige situaties bevat een pod 1 of meerdere sidecar-containers.   * Deze implementeren ondersteunende functies. |

|  |
| --- |
| Control Plane componenten:  **Kube-API-server**  Biedt een REST interface aan het Kubernetes control plane en de datastore (etcd).   * Alle clients en andere applicaties communiceren met Kubernetes strikt via de API-server.   Fungeert als een gatekeeper voor het cluster door authenticatie, autorisatie, verificatie van verzoeken en door toelatingscontrole af te handelen, naast het fungeren als de frontend naar de onderliggend datastore.  **ETCD**  Fungeert als de datastore van het cluster.   * Biedt een sterke, consistente en highly available key-value store voor het persisten van de clusterstatus. Slaat configuratie-informatie op.   ETCD is zelf een snel en veilig gedistribueerd systeem, waardoor een fout-tolerante, consistente weergave van het cluster wordt gecreëerd.  **Cloud-Controller-Manager**  Daemon die cloud-provider specifieke integratie biedt in de code control loop van Kubernetes.  De controllers binnen de cloud controller manager omvatten:   * **Node controllers**: verantwoordelijk voor het maken van node objecten wanneer nieuwe servers worden gemaakt inde cloud infrastructuur. * **Routw controller**: verantwoordelijk voor het configureren van routes in de cloud. * **Service controller**: integratie met load balancers, IP-adres toewijzing, health checks.   Google, Amazon, … hebben allemaal hun eigen standaarden voor het leveren van volumes en services aan een cluster.   * Cloud controller manager fungeert als een middle man.   **Kube-Controller-Manager**  Dient als primaire service/daemon da talle core component control loops beheert.   * Controlelussen regelen de status van het systeem.   Controleert de clusterstatus via de API-server en stuurt het cluster naar de gewenste status.  Enkele voorbeelden van controllers:   * **Node controllers**: verantwoordelijk voor het opmerken en reageren wanneer nodes uitvallen. * **Replication controller**: beheert het correct aantal replicas. * **Service account en token controller**: maken van standaardaccounts en API-toegangstokens.   **Kube-Scheduler**  Verbose policy-rich engine die de vereisten van workloads evalueert en probeert de workload op een overeenkomstige resource te plaatsen.   * Bepaalt welke nodes welke pods moeten uitvoeren. * De standaardscheduler gebruikt bin packing.   Houdt nieuw gecreëerde pods in de gaten en selecteert een node waarop ze moeten worden uitgevoerd.  Werklastvereisten kunnen omvatten:   * Algemene hardware vereisten * Affiniteit/anti-affiniteit * Labels * Andere diverse aangepaste bronvereisten   Node Componenten:  **Kube-Proxy**  Beheert de netwerkrules op elke node   * Staat netwerkcommunicatie toe naar pods van netwerksessies binnen of buiten je cluster.   Voert connectieforwarding of load balancing uit voor Kubernetes Services.  **Kubelet**  Node agent veranwoordelijk voor het beheren van de lifecycle van elke pod op zijn host.  Registreert de node bij de API-server.  Kubelet begrijpt YAML pod manifesten die het kan lezen vanuit verschillend bronnen, waaronder:   * API server * Bestandspad/File path * HTTP endpoint * HTTP servermodus die containermanifesten accepteert via een eenvoudige API. |

|  |
| --- |
| **Canary Release**  Een Canary Release is een implementatiestrategie waarbij een nieuwe versie van de software geleidelijk wordt uitgerold naar een klein subset van gebruikers voordat deze wordt vrijgegeven aan de volledige gebruikersbasis.   * De ultieme technische test in de “echte” productieomgeving. * Een zeer kleine groep gebruikers wordt mogelijk beïnvloed door eventuele bugs.   **A/B Testing**  A/B Testing (ook wel split testing genoemd) houdt in dat twee versies van een webpagina, applicatie of functie worden vergeleken door willekeurige verschillende versies aan gebruikers te presenteren en te beoordelen welke beter presteert.   * Een van de twee versies worden willekeurig aan gebruikers getoond. * Evaluatie van bedrijfsmetrics, in plaats van technische testen. |

## Hoofdstuk 4: CNCF

|  |
| --- |
| **HELM**  Het opzetten van een applicatie in Kubernetes = het schrijven van een gedetailleerd YAML-manifestbestand voor pods, services, workloads, enz.  HELM is een command line package manager voor Kubernetes, waarmee ontwikkelaars applicaties gemakkelijker kunnen verpakken, configureren en implementeren op Kubernetes.   * Vergelijkbaar met Debian’s APT of Python pip, het verpakkingsformaat wordt “charts” genoemd.   HELM is gericht op ‘dag 1’ operaties:   * Het installeert en werkt software en softwareafhankelijkheden bij. * Het configureert software implementaties. * Het haalt softwarepakketten op uit repositories. |

|  |
| --- |
| **Kubernetes Operators**  Kubernetes Operators zijn uitbreidingen op Kubernetes waarmee je complexe, stateful applicaties kunt beheren zoals een expert dat zou doen, maar dan geautomatiseerd.  Een **Operator** is een softwarecomponent die:   * Gebruik maakt van de Kubernetes API * De toestand van een applicatie observeert * Automatisch actie onderneemt indien nodig.   Je kunt het zien als een soort “robot-systeembeheerder” voor je applicatie binnen Kubernetes.  Bij de keuze tussen HELM en Kubernetes operatoren:   * HELM: als je alleen een applicatie installeert. * K8s operator: afhankelijk van de mate van aanpassingen en volwassenheid van het cluster. * Standaardconfiguratie: HELM * Speciale Configuratie: K8s Operators * Eerste installatie: HELM * Meer geavanceerd: K8s Operators |

|  |
| --- |
| **Vitess**  Vitess is een database clustering systeem voor horizontale schaalbaarheid van MySQL, ontworpen voor cloud-native omgevingen zoals Kubernetes. Het wordt veel gebruikt om grote hoeveelheden verkeer, data en gelijktijdige gebruikers aan te kunnen, met behoud van MySQL-compatibiliteit.  Vitess werd oorsponkelijk ontwikkeld door YouTube en is nu een CNCF-project, net als Kubernetes.  Vitess maakt het mogelijk om MySQL-databases te sharden, schalen en beheren, zonder dat je je applicatie hoeft te herschrijven. Het gedraagt zich als een proxylaag tussen je applicatie en MySQL-instances, en voegt daar intelligente logica aan toe.  Belangrijkste **Features van Vitess**  Performantie:   * **Sharding:** Verdeelt je database over meerder MySQL-instances (shards), zodat je horizontaal kunt schalen. * **Connection pooling**: Beheert efficient databaseverbindingen, belangrijk voor grote aantallen gelijktijdige gebruikers. * **Query de-duping**:  Hergebruik de resultaten van een lopende query voor identieke verzoeken terwijl de lopende query nog steeds uitgevoerd wordt. * **Transaction manager**: Beperkt het aantal gelijktijdige transacties en beheer deadlines om de doorgang te optimaliseren.   Beveiliging:   * **Query rewriting & routing**: Stuurt SQL-queries naar de juiste shard(s) en past ze aan indien nodig. * **Failover en reparenting**: Ondersteunt automatische en handmatige failovers bij crashes van primaries. * **Backup & restore**: geïntegreerde tools voor back-ups van gesharde databases. |

|  |
| --- |
| **ETCD** ETCD is een lichtgewicht, gedistribueerde key-value store, gebouwd door CoreOS.  Het is geoptimaliseerd voor lage latency, hoge consistentie, en wordt gebruikt voor:   * Configuratiebeheer * Leader election * Service Discovery * Kubernetes intern (voor de control plane)   **Gebruik** (door Kubernetes voor onder andere):   * Te onthouden welke pods er zijn * Welke nodes er zijn * Welke configuraties zijn toegepast   **TiKV**  TiKV is een gedistribueerde transactionele key-value database, ontwikkeld door PingCAP.  Het is bedoeld als Back0end opslagengine voor relationele databases.  Biedt ACID-transacties over meerder keys en tabellen.  Het is gebouwd op RocksDB (per node) + een gedistribueerd Raft protocol.  **Gebruik**:   * TiKV wordt gebruikt waar je een schaalbare, transactionele opslag nodig hebt. * Het vormt het fundament voor TiDB, een MySQL-compatibele gedistribueerde SQL-database. * Geschikt voor Online Transaction processing (OLTP) workloads die MySQL overstijgen in schaal. |

|  |
| --- |
|  |