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

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

Hoofdstuk 3: Kubernetes

12. Beschrijf Kubernetes.
13. Leg Kubernetes key concept: PODS uit a.d.h.v. een figuur.
14. Leg de Kubernetes Architectuur uit.
15. Wat is Canary Release an A/B testing?

Hoofdstuk 4: CNCF

16. Wat weet je over HELM?
17. Wat zijn Kubernetes Operators?
18. Wat zijn Vitess Features en wat kan dit doen?
19. Leg uit wat TiKV en ETCD is en wanneer je welke gebruikt.
20. Wat weet je over ENVOY? (staat niet meer in de slides, niet meer kennen?)
21. Wat weet je over HARBOR?
22. Wat weet je over TUF?

Hoofdstuk 5: Software Development Models

23. Wat zijn Plan-Driven en Agile Processes?
24. Wat is het Waterfall Model?
25. Wat is Incremental Development?
26. Wat versta je onder Reuse-Oriented Software Engineering?
27. Leg uit: Verandering en hoe kan je de kosten verminderen?
28. Wat is prototype development?
29. Wat is incremental delivery?

Hoofdstuk 6: Agile Software Development

30. Wat is het verschil tussen Plan-Driven en Agile Development?
31. Leg uit: Practices van extreme programming
32. Wat is SCRUM?

Hoofdstuk 7: Requirements engineering

33. Wat zijn functionele en niet-functionele requirements?
34. Wat zijn requirements change management?

Hoofdstuk 8 : Design and implementation

35. Wat is configuration management tool interaction?
36. Wat weet je over open-source development?

Hoofdstuk 9: Software Testing

37. Bespreek inspections and testing.
38. Bespreek white box en black box testing.
39. Bespreek code coverage.
40. Bespreek development testing.
41. Bespreek automated testing.
42. Bespreek test-driven development.

Hoofdstuk 10: Configuration management

43. Hoe ziet continuous integration uit in een agile building omgeving?

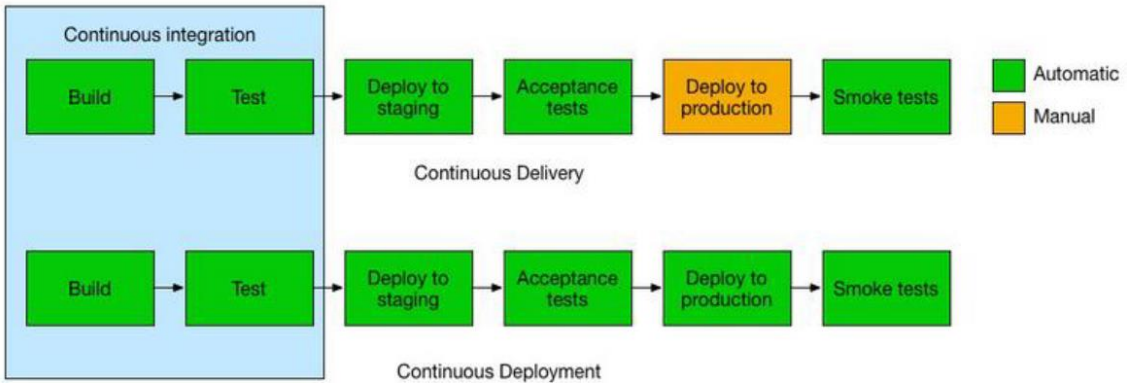
Hoofdstuk 1: Wat is DevOps?

1.

Continuous delivery/deployment :
Continue integratie: builden en testen.
Continue delivery: implementeren en integratietesten.
Continue deployment: builden, testen, implementeren en alles brengen naar productie.

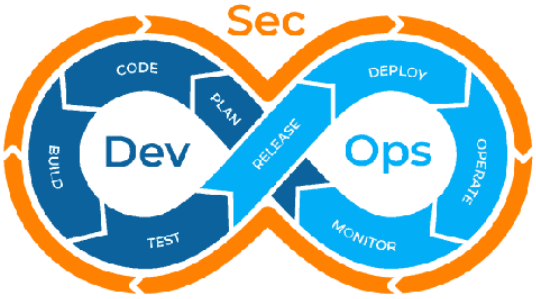
Best practices:

- 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.



2.

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

3.

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.

4.

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.

5.

Rol van OS layers in Containers:

Container Host of Host OS:

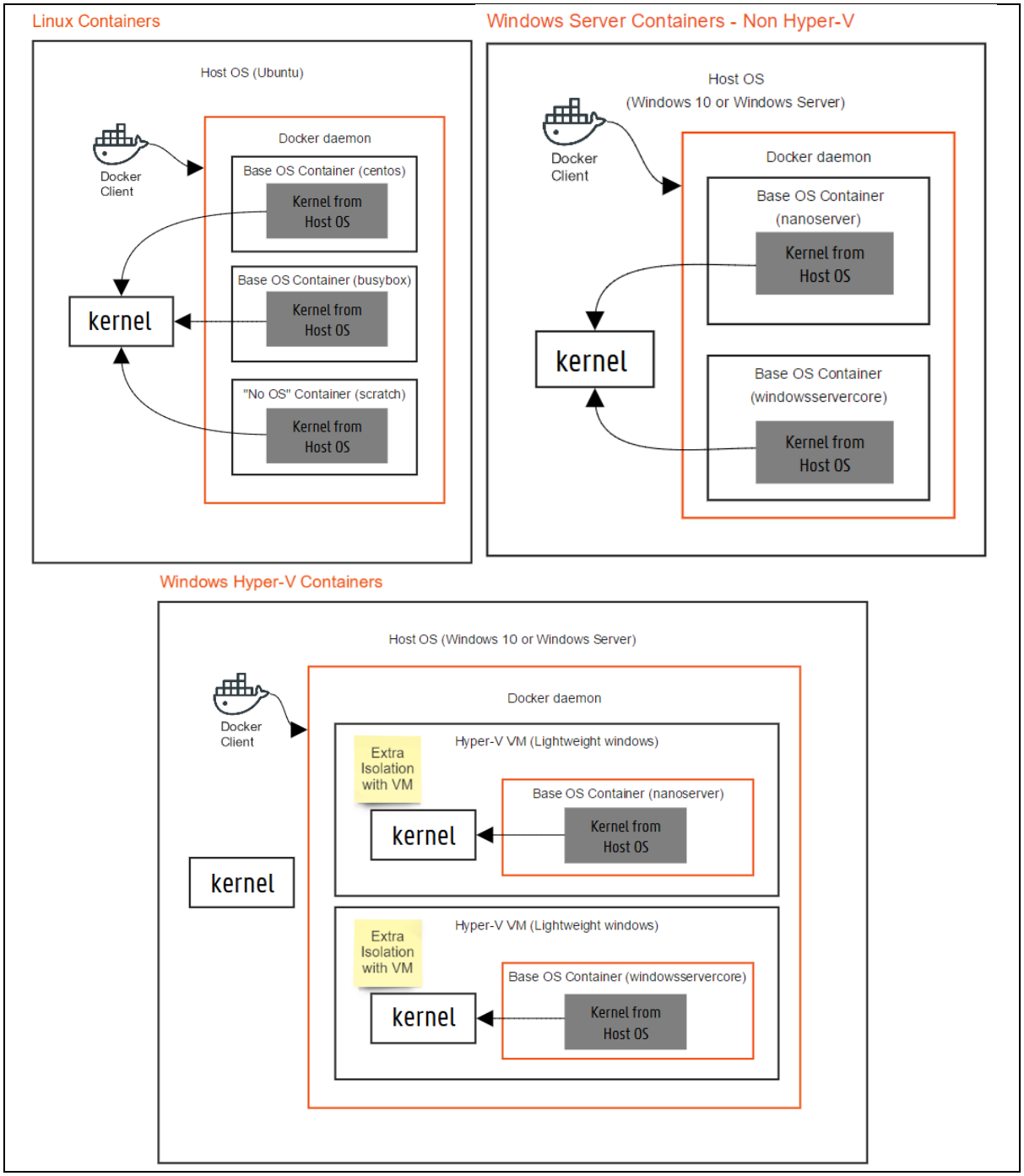
- Het besturingssysteem waarop de Docker-client en Docker-daemon draaien.
- In het geval van Linux en niet-Hyper-V containers deelt het Host OS zijn kernel met de draaiende Docker-containers.

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.


Linux Containers
Host OS (Ubuntu)
Docker daemon
Docker Client
Base OS Container (centos)
Kernel from Host OS
Base OS Container (busybox)
Kernel from Host OS
"No OS" Container (scratch)
Kernel from Host OS
kernel
Windows Server Containers - Non Hyper-V
Host OS
(Windows 10 or Windows Server)
Docker daemon
Docker Client
Base OS Container (nanoserver)
Kernel from Host OS
Base OS Container (windowsservercore)
Kernel from Host OS
kernel
Windows Hyper-V Containers
Host OS (Windows 10 or Windows Server)
Docker daemon
Docker Client
Hyper-V VM (Lightweight windows)
Extra Isolation with VM
Base OS Container (nanoserver)
Kernel from Host OS
kernel
Hyper-V VM (Lightweight windows)
Extra Isolation with VM
Base OS Container (windowsservercore)
Kernel from Host OS
kernel
kernel

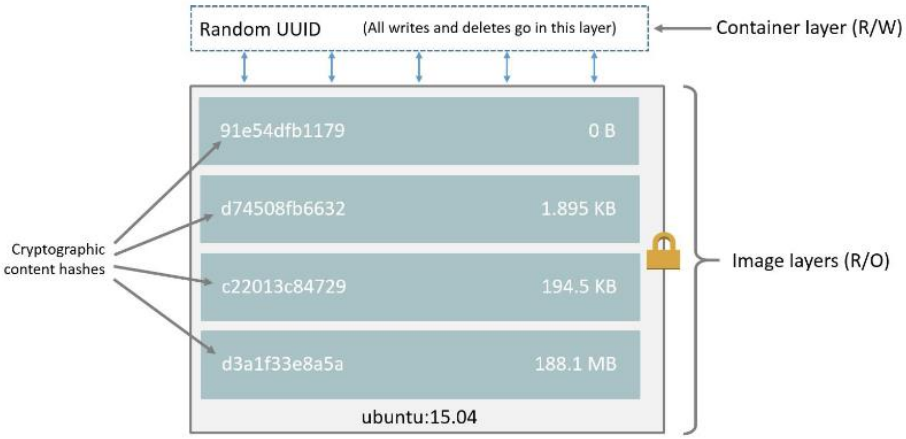
6.

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.

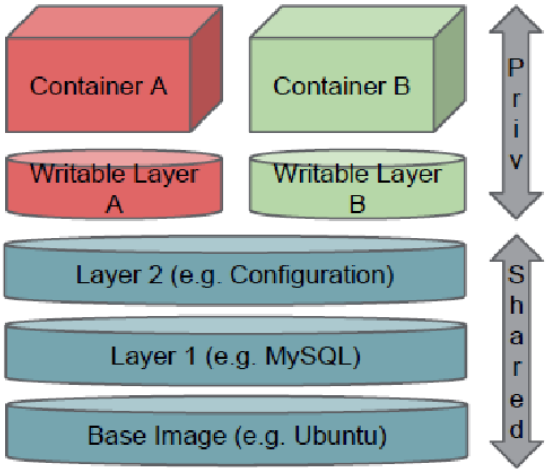
- Er is overhead betrokken bij de special opslagdriver die door Docker wordt gebruikt.
- Het schrijven naar containerlaag wordt gebruikt tijdens het bouwproces van het image.



7.

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.



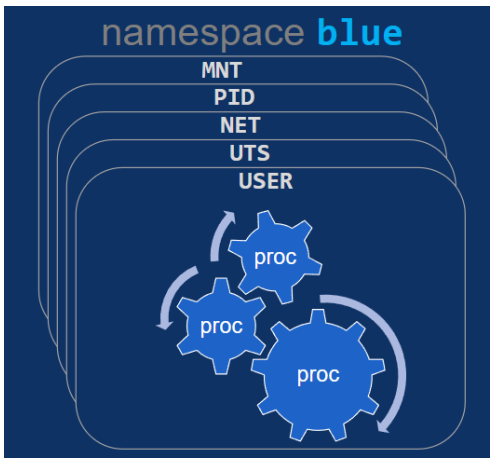
8.

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)

9.

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.

10.

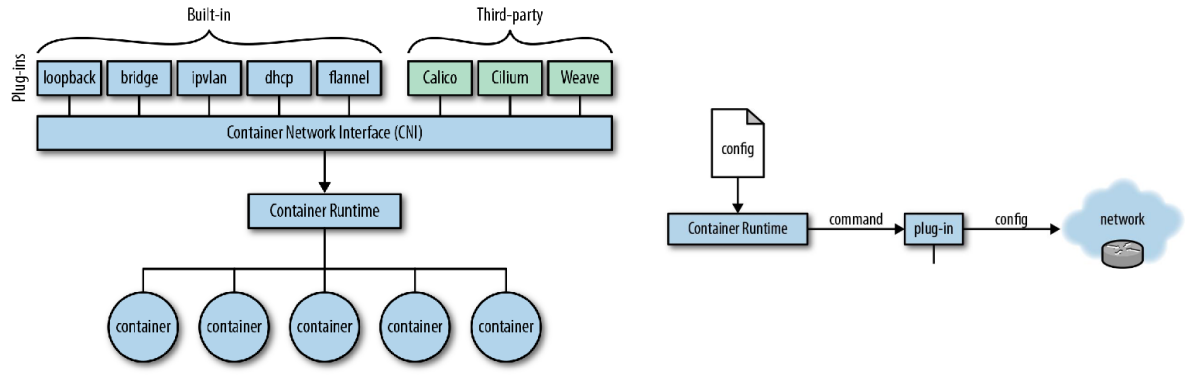
Container Networking Interface – CNI

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)

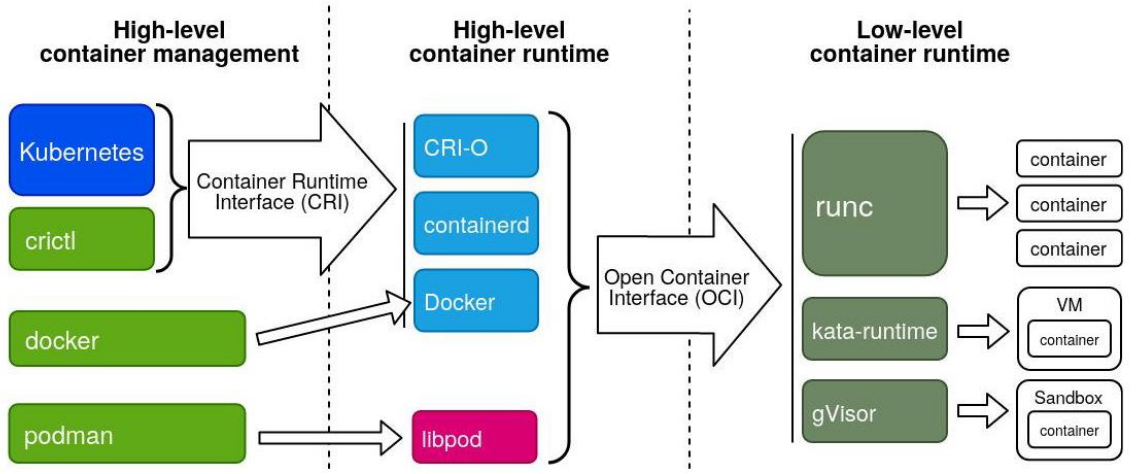
Een op plug-in gerichte netwerkoplossing voor containers en container orchestrators

- Wordt gebruikt door Kubernetes, Mesos, Cloud Foundry, enz.



CNI zorgt er dus voor dat uw netwerkinterface gestandaardiseerd wordt, dat de API's vastliggen, ...

11.



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

12.

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.

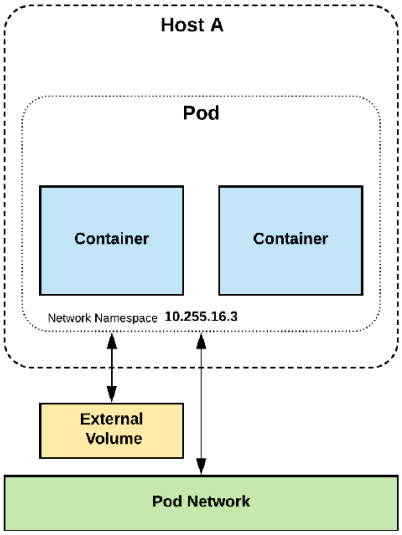
13.

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.

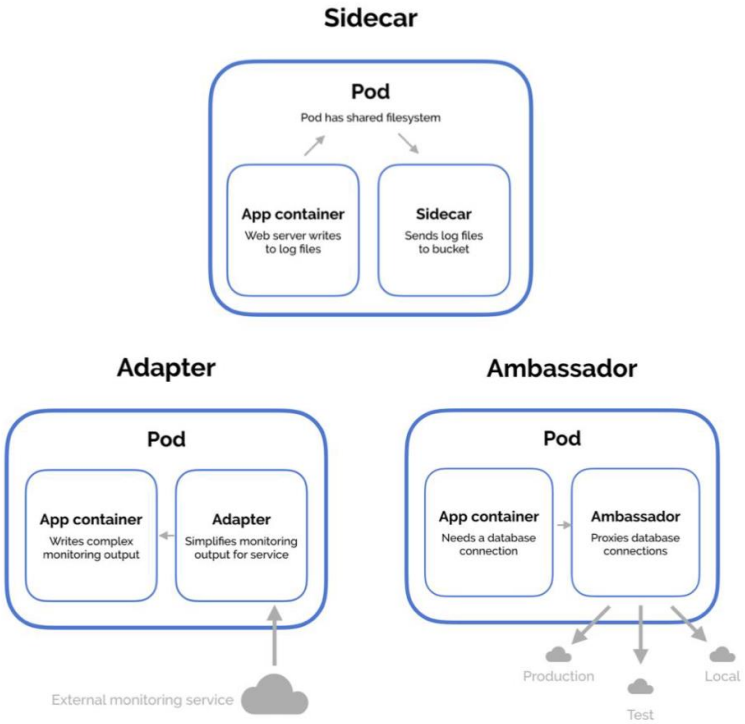
- Beschrijf de gewenste staat in termen van onder andere het aantal kopieën en vereiste fouttolerantie.

Multi-Container pods
Het een-container-per-pod model is het meest voorkomende Kubernetes-gebruikersscenario.

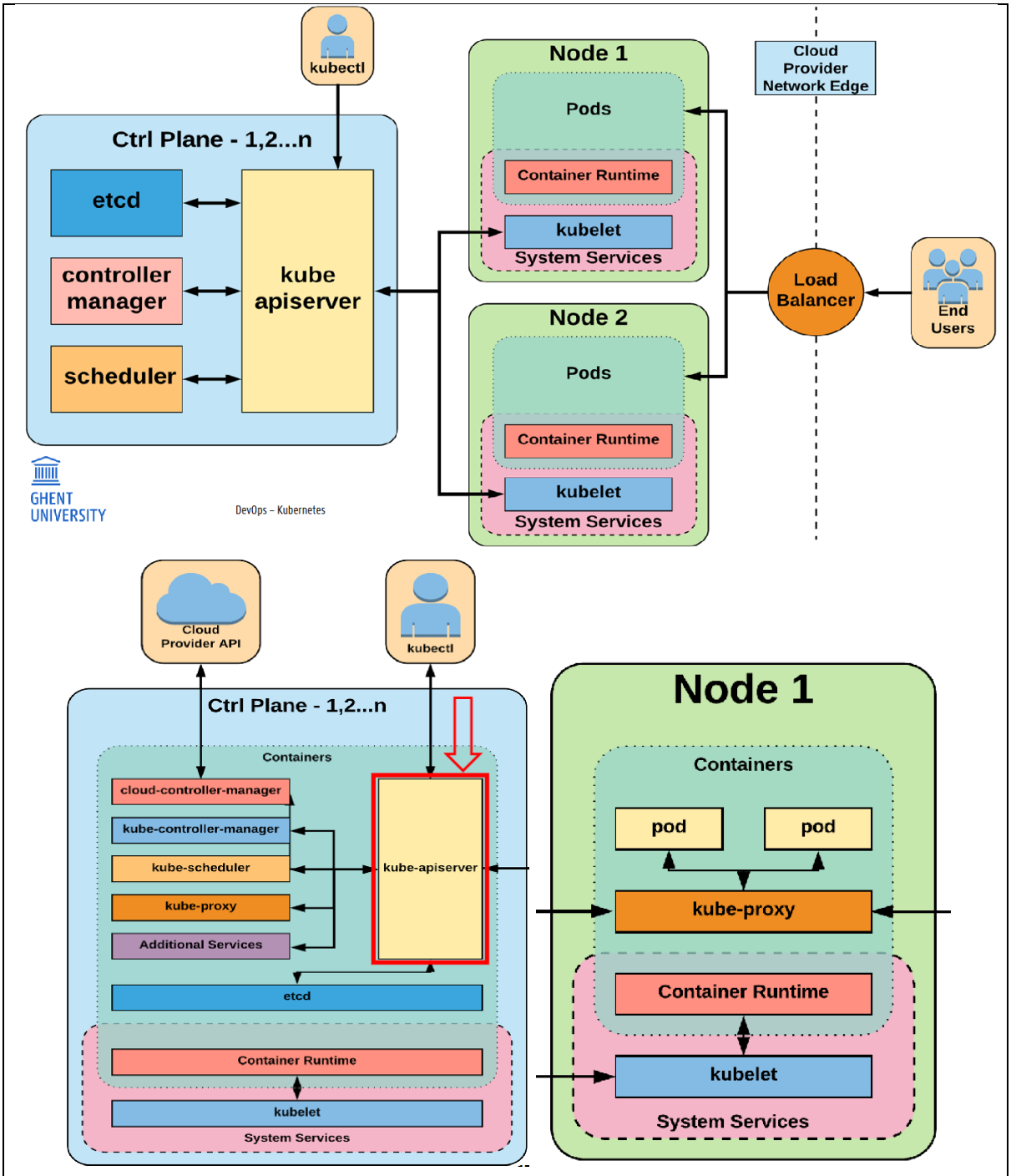
- Denk aan een pod als een omhulsel rond een enkele container en Kubernetes beheert pods in plaant van containers rechtstreeks.

In sommige situaties bevat een pod 1 of meerdere sidecar-containers.

- Deze implementeren ondersteunende functies.



14.



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

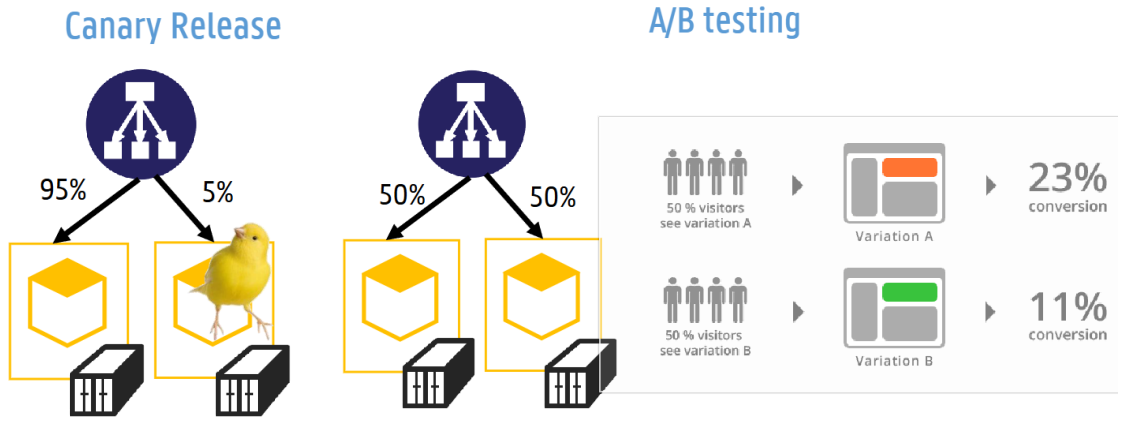
Node agent veranwoordelijk voor het beheren van de lifecycle van elke pod op zijn host.

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.

15.



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

16.

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.

17.

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

18.

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.

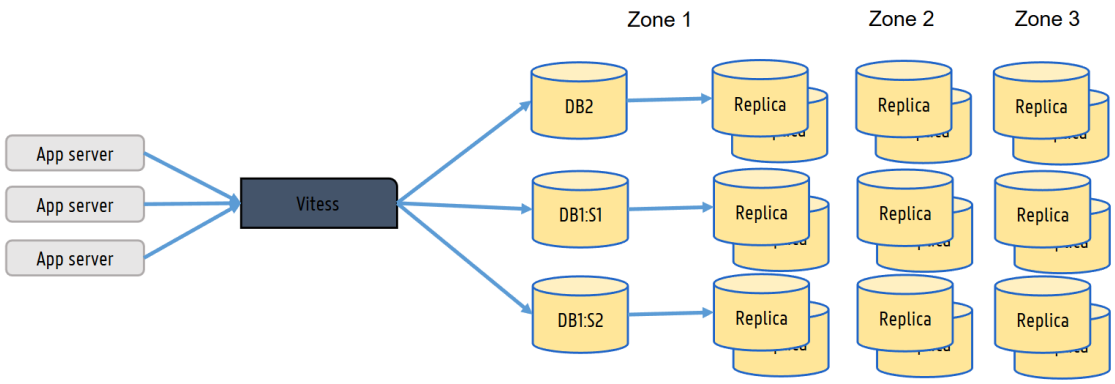
Vitess maakt het mogelijk om MySQL-databases te sharden, schalen en beheren, zonder dat je je applicatie hoeft te herschrijven. Het gedraagt zich als een proxylaag tussen je applicatie en MySQL-instances, en voegt daar intelligente logica aan toe.

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.



19.

ETCD
ETCD is een lichtgewicht, gedistribueerde key-value store, gebouwd door CoreOS.
Het is geoptimaliseerd voor lage latency, hoge consistentie, en wordt gebruikt voor:

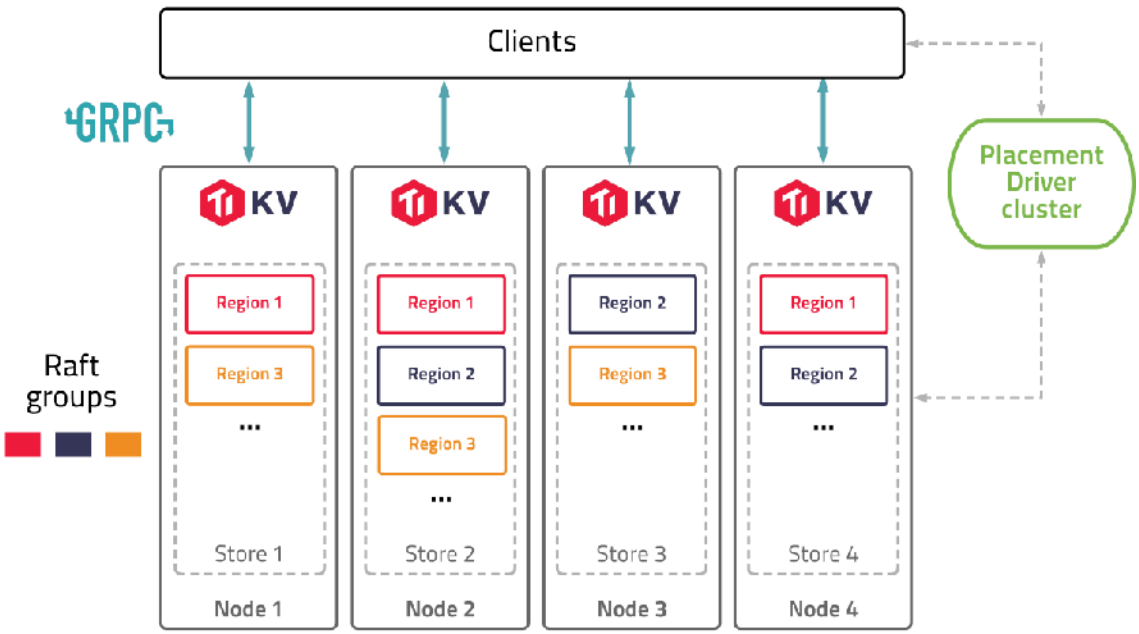
- Configuratiebeheer
- Leader election
- Service Discovery
- Kubernetes intern (voor de control plane)

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.

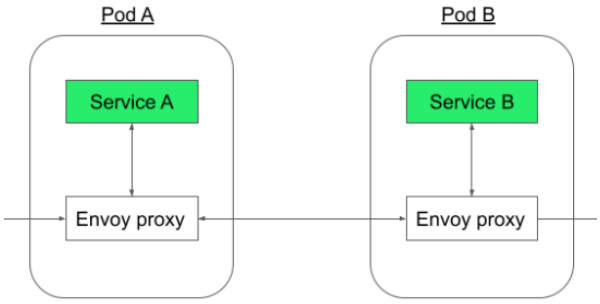


20.

ENVOY is een high-performance, open-source proxy ontworpen voor gebruik in moderne gedistribueerde microservices-architecture. Het fungeert al een service mesh dataplane, API- gateway of edge proxy, en is vooral popular in combinatie met Kubernetes.

ENVOY zorgt voor:

- **Load balancing**
  Verdeelt verkeer slim over meerder services of instances.
- **Observability**
  Wanneer al het serviceverkeer in een infrastructuur via een ENVOY-mesh stroomt, is het eenvoudig om probleemgebieden visualiseren
- **Traffic routing**
  Slimme request-routes op basis van headers, versies, etc.
- **Rate limiting**
  Beschermt diensten tegen overbelasting
- **Sidecar Proxy**
  Naast elke service draait een sidecar proxy voor het doorsturen van netwerkverkeer



21.

Harbor is een open-source container image registry die bovenop Docker Registry is gebouwd. Het wordt gebruikt om Docker-images (of andere OCI-compliant artifacts) op te slaan en te beheren, met functies die gericht zijn op veiligheid, compliance en efficiënt beheer.

Features:

- Zowel zelf gehost als beheer in de cloud.
- Ondersteuning voor non-container artifacts: OCI artifacten zoals Helm-charts, OCI-container images, enz.
- Geo-replication: synchroniseren van container images (of andere artifacts) tussen meerdere Harbor-instanties op verschillende geografische locaties, voor een hoge beschikbaarheid in verschillende data centers.
- Garbage collection: verwijderen van onnodige gegevens met garbage collection tasks.

- Security:
    - Vulnerability scanning: Clair, Trivy
    - Ondertekenen/verifieren van images
    - Authenticatie methoden: LDAP, Active Directory, OpenID Connection.

Sterktes:

- Enige afgestudeerde CNCF-containerregistryproject
- Hoge activiteit op open-source repository en veel bijdragers
- Integratie met kwetsbaarheidscanning en ondertekende/vertrouwde inhoud
- Ondersteunt meerdere authenticatiemethoden.
- Integratie met elke Kubernetes-disctibuties.

Nadeel:

- Nu ook actief als een repository voor machine learning modellen: risk of bloating.

22.

The Update Framework is ene beveiligingssysteem dat ervoor zorgt dat software-updates (zoals packages of container images) veilig zijn.

In DevOps wil je zeker weten dat de software of images die je gebruikt:

- Niet aangepast worden zijn door iemand anders.
- De juiste versie zijn.
- Van een vertrouwde bron komen.

TUF helpt daarbij door alles te ondertekenen met digitale handtekeningen.

Hoe werkt TUF?
TUF maakt gebruik van **metadata** (informatie over de bestande) die wordt ondertekend door verschillende rollen:

- Root: Beheert de belangrijkste sleutels (de base)
- Timestamp: Controleert of je de nieuwste update hebt.
- Snapshot: Lijst van alle versies van bestanden.
- Targets: Verwijst naar de echte bestanden (zoals een Docker image of softwarepakket).

Elke rol heeft zijn eigen **sleutels** om dingen te ondertekenen. Als er iets niet klopt, dan zal TUF het updateprocedure stopzetten.

Hoofdstuk 5

23.

Plan-Drive development
Dit is een manier van software maken waarbij **alles op voorhand wordt gepland.**

Kenmerken:
- Je maakt eerst een volledig plan: wat bouwen we, hoe, wanneer, wie?
- Daarna volgen vaste fasen: analyse -> ontwerp -> implementatie -> testen -> opleveren.
- Je wijkt zo weinig mogelijk af van het plan.

Goed voor:
- Grote projecten met weinig veranderingen.
- Overheidsprojecten of contractwerken.
- Als je duidelijk weet wat je nodig hebt op voorhand.

Agile development
Dat is een flexibelere manier van werken waarbij je **stap voor stap** software maakt en telkens bijstuurt op basis van feedback.

Kenmerken:
- Je werkt in korte iteraties (meestal 1-3 weken), ook wel sprints genoemd.
- Na elke sprint heb je een werkend stukje software.
- Je kan makkelijk inspelen op veranderingen of nieuwe wensen.
- Veel communicatie met het team en de klant.

Goed voor:
- Projecten waar veel verandert.
- Start-ups, SaaS-projecten.
- Teams die snel willen leveren en verbeteren.

24.

Het waterfall model is een manier van werken waarbij je het project in vaste stappen doet en je eerst een stap moet afwerken voor je naar de volgende mag.

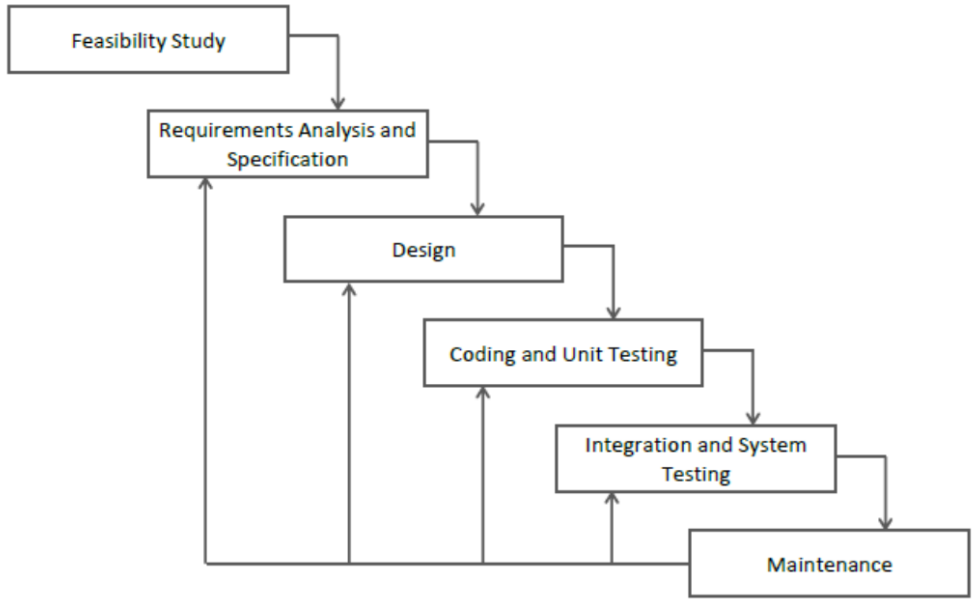
Kenmerken:
- Elke stap gebeurt 1 keer.
- Je keert niet terug naar een vorige stap.
- Je hebt pas op het einde een werkend product.

Goed voor:
- Projecten met duidelijk en stabiele eisen.
- Als alles vooraf goed gepland kan worden.

Minder goed voor:

- Projecten waar dingen vaak veranderen.
- Je ziet pas laat of het goed werkt.

De zes typische stappen:



25.

Incremental development betekent dat je een softwareproject in **kleine stukjes bouwt** (incrementeren), en elke keer iets werkends toevoegt.

Hoe werkt het?

- Je maakt eerst een **basisversie** van de software.
- Daarna voeg je telkens **nieuwe functies** toe.
- Elke stap of versie werkt op zichzelf en kan getest worden.

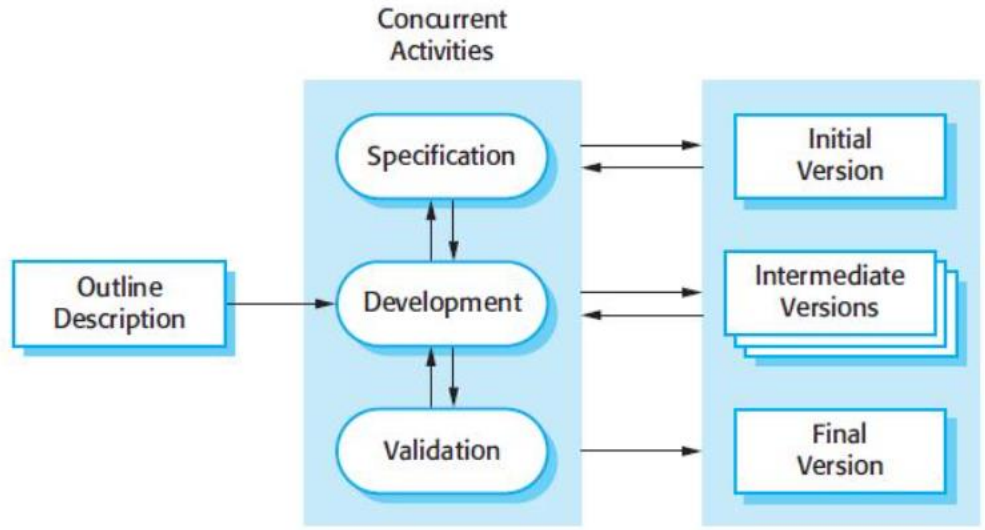
Voordelen:

- De hoeveelheid analyse en documentatie die opnieuw moet worden gedaan, is veel minder dan bij het waterfall model.
- Het is gemakkelijker om feedback van klanten te krijgen over het ontwikkelingswerk dat is verricht.
- Klanten kunnen opmerkingen geven over de demonstraties van de software en zien hoeveel er is geïmplementeerd.
- Snellere levering en implementatie van bruikbare software voor de klanten zijn mogelijk.
- Klanten kunnen eerder gebruik maken van de software en er waarde uit halen dan bij een waterfall model.

Nadelen:

- De systeemstructuur neig te verslechteren naarmate nieuwe increments worden toegevoegd.

- Tenzij tijd en geld worden besteed aan refactoring om de software te verbeteren, heeft regelmatige verandering de neiging om de structuur ervan te beschadigen.



26.

Reuse-Oriented Software Engineering is een manier van software ontwikkelen waarbij je **bestaand componenten** hergbruikt in plaats van alles van nul te schrijven.

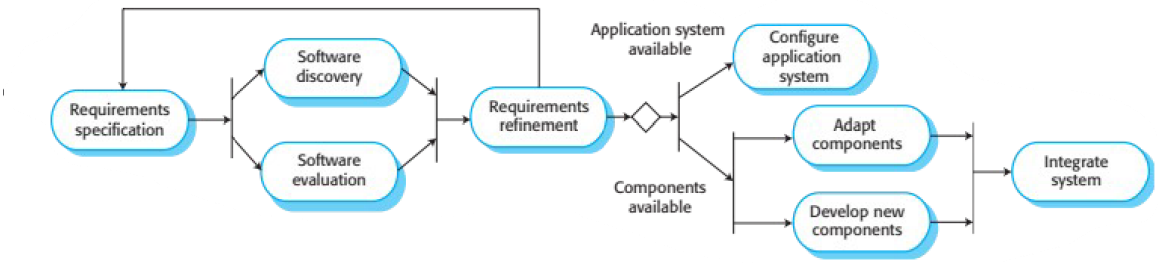
Je stelt dus software samen door blokken (componenten) te **hergebruiken** die al bestaan.

Voordelen:

- Doordat minder software vanaf nul wordt ontwikkeld, worden kosten en risico's verminderd.
- Snellere levering en implementatie van systemen.

Nadelen:

- Minder controle, je bent afhankelijk van anderen.
- Alles moet goed samenwerken.
- Beveiliging van externe onderdelen moet je goed nakijken.



27.

Reducing the costs of rework
Vermijden van wijzigingen:
- Het softwareproces omvat activiteiten die mogelijke veranderingen kunnen anticiperen

Tolerantie voor verandering:
- Het project is ontworpen zodat veranderingen met relatief lage kosten kunnen worden opgevangen.
- Dit omvat meestal een vorm van incrementele ontwikkeling. Voorgestelde wijzigingen kunnen worden geïmplementeerd in increments die nog niet zijn ontwikkeld. Als dit onmogelijk is, dan moet slechte 1 increment (een klein deel van het systeem) worden gewijzigd om de verandering op te nemen.

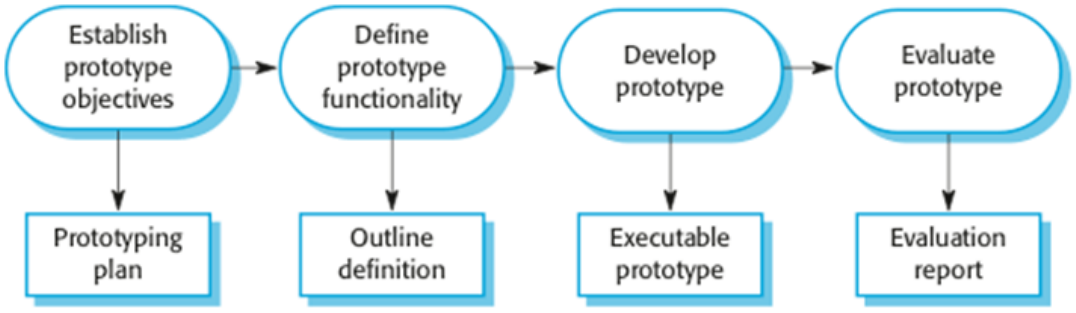
28.

Prototype Development betekent dat je eerst een simpele, werkende versie van je software maakt, om te laten zien hoe het er ongeveer gaat uitzien en werken.

Het is nog niet af en vaak **niet volledig functioneel**, meer geeft wel al een goed idee van:
- De **interface**
- De **belangrijkste functies**
- De **structuur**

Voordelen:
- Snel resultaat
- Goedkoper dan alles in 1 keer te bouwen
- Minder risico op misverstanden
- Klanten voelen zich meer betrokken

Nadelen:
- Je prototype wordt soms verward met het eindproduct
- Het is tijdrovend als je er te veel details in verwerkt.



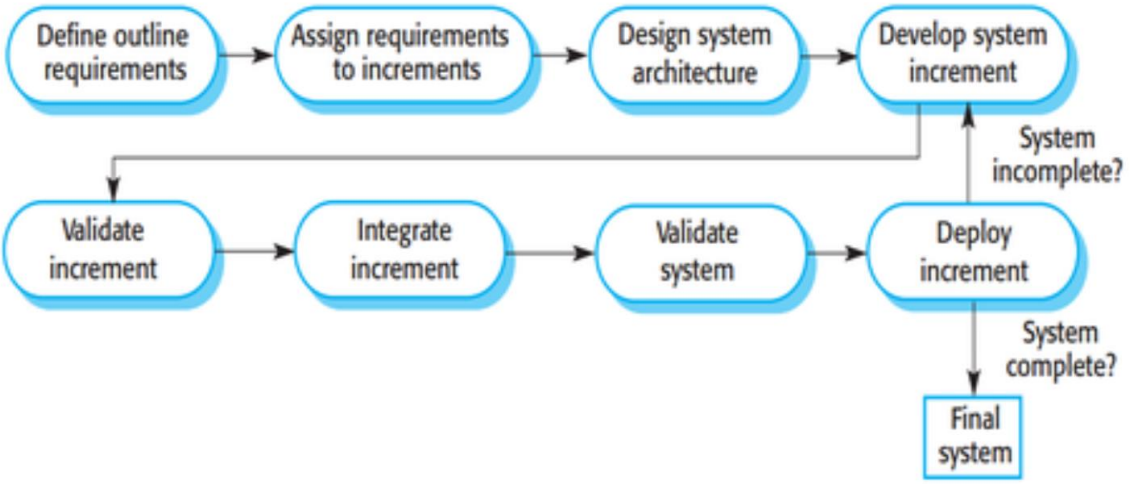
29.

Incremental Delivery betekent dat je software in kleine stukjes oplevert, telkens met nieuwe functies die echt werken.

In plaats van 1 grote oplevering op het einde, lever je stap voor stap werkende delen van het systeem aan de klant of gebruiker.

Voordleen:
- Snelle waarde voor de klant
- Feedback na elke versie
- Minder risico: je weet sneller of je op de juiste weg zit
- Je kan vroeg beginnen met testen en verbeteren

Nadelen:
- Je moet goed plannen welke functies eerst komen
- Latere aanpassingen kunnen invloed hebben op vorige versies



Hoofdstuk 6

30.

Plan Driven Development wil zeggen dat je een **volledig plan** zal maken en dat stap voor stap deze plan zal uitvoeren.

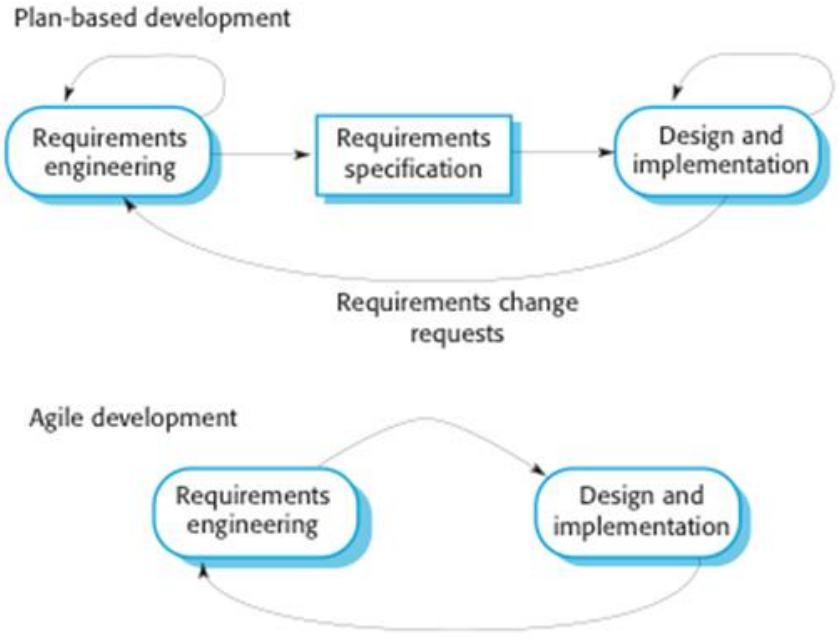
Kenmerken:

- Alles wordt vooraf uitgedacht (requirements, ontwerp, planning)
- Niet noodzakelijk waterfall model, incremental development is mogelijk
- Veranderingen zijn moeilijk en duur.
- Je krijgt het eindproduct meestal pas op het einde.

Agile development wilt zeggen dat je software zal maken in kleine stukjes (iteraties), met veel feedback en ruimte voor veranderingen.

Kenmerken:

- Je werkt in korte sprints
- Na elke sprint is er een werkend stukje software
- Veranderingen zijn normaal en makkelijk aan te pakken
- Je werkt samen met de klant tijdens het hele proces



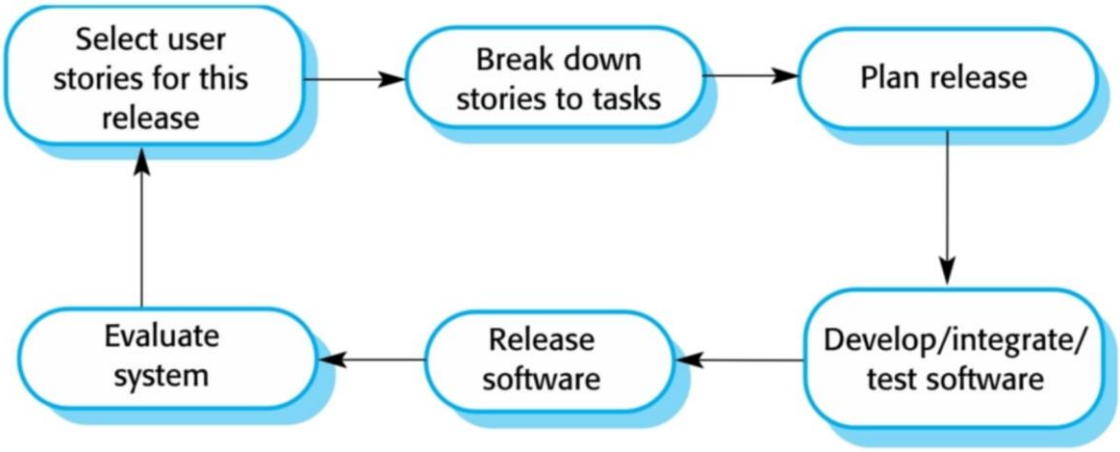
31.

Extreme Programming is een manier van software bouwen die focust op:

- Snelle feedback
- Veel samenwerking
- Technische excellentie
- Simpele oplossingen

XP bevat een reeks praktische technieken (practices) die teams helpen om sneller software te leveren van hoge kwaliteit.

1. Pair programming
   Twee programmeurs werken samen aan dezelfde computer: 1 typt en de andere denk mee.
2. Test-Drive Development
   Je schrijft eerst de test en dan pas de code.
3. Continuous integration
   Je integreert je code vaak (meerdere keren per dag).
   Zo voorkom je conflicten en bugs, en je weet snel of alles goed werkt.
4. Simple Design
   Bouw enkel wat je nu nodig hebt en niks extra.
   Zo blijft het systeem makkelijk te begrijpen en aan te passen.
5. Refactoring
   Verbeter regelmatig je code (zonder gedrag te veranderen)
   Zo blijft je code netjes en makkelijk te onderhouden.
6. Collective Code Ownership
   Iedereen mag aan elk deel van de code werken
   Geen “mijn” code of “jouw” code – het is van het hele team.
7. On-site customer
   De klant of gebruiker zit vlakbij het team
   Je krijgt direct antwoord als er iets onduidelijk is



32.

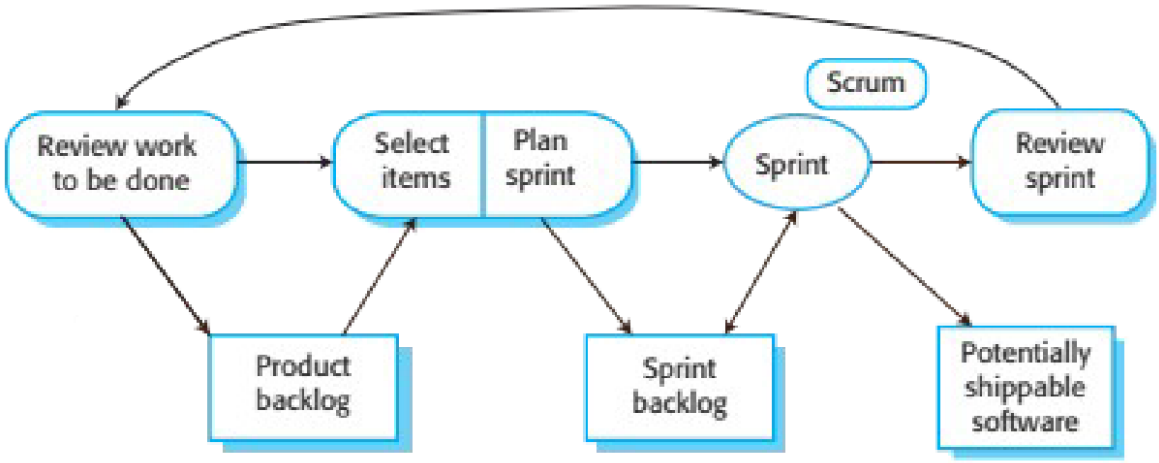
Scrum is een agile manier van werken om software te maken. Je werkt in een klein team, in korte periodes (sprints), en je levert telkens een werkend stukje software op. Het doel: sneller resultaat + regelmatige feedback + makkelijk kunnen aanpassen.

Het omvat drie fasen:

1. **Initial Phase**
   In deze fase wordt een ruw planning opgesteld waarbij de algemene doelstellingen van het project worden vastegelegd en de softwarearchitectuur wordt ontworpen.
2. **Sprint-Cycles**
   Vervolgens worden er opeenvolgende sprintcycli uitgevoerd, waarbij elke cyclus een increment van het systeem ontwikkeld.
3. **Project closure phase**
   Deze fase sluit het project af, voltoord de vereiste documentatie zoals systeemhulpframes en gebruikershandleidingen en evalueert de lessen die zijn geleerd tijdens het project.

Voordelen:

- Snelle feedback
- Flexibel met veranderingen
- Team werkt beter samen
- De klant ziet snel resultaten.



Hoofdstuk 7

33.

Functionele requirements beschrijven wat het systeem moet doen, hoe het systeem zou moeten reageren op specifieke invoer en hoe het systeem zich zou moeten gedragen in bepaalde situaties.
Ze gaan over de functies, taken en gedrag van de software.
Dit zijn dingen die je kunt zien of gebruiken.

Niet functionele requirements beschrijven hoe goed het systeem moet werken. Ze Zijn beperkingen op de diensten of functies die door het systeem worden aangeboden, zoals timingbeperkingen, beperkingen op het ontwikkelingsproces, standaarden enz.
Dit gaat over de snelheid, veiligheid, betrouwbaarheid enz.

Domeineisen zijn regels, kennis of beperkingen die komen uit het specifieke vakgebied (het domein) waarvoor je software maakt.

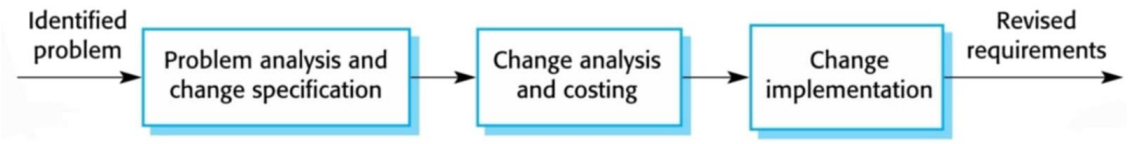
34.

Requirements change management beschrijft hoe je wijzigingen in de eisen of wensen van een softwareproject op een goede manier beheert.

Problem analysis and change specification:
Tijdens deze fase wordt het probleem of het wijzigingsvoorstel geanalyseerd om te controleren of het geldig is. Deze analyse wordt teruggekoppeld naar degene die het wijzigingsverzoek heeft ingediend, die mogelijk reageert met een meer specifiek wijzigingsvoorstel of besluit het verzoek in te trekken.

Change analysis and costing:
De invloed van de voorgestelde wijziging wordt beoordeeld met behulp van traceability info en algemene kennis van de systeemeisen. Zodra deze analyse is voltooid, wordt besloten of al dan niet door te gaan met de wijzigingen door de eisen.

Change implementation:
Het requirements document, en indien nodig het systeemontwerp en de implementatie worden aangepast. Idealiter moet het document zo zijn georganiseerd dat wijzigingen eenvoudig kunnen worden geïmplementeerd.



Hoofdstuk 8

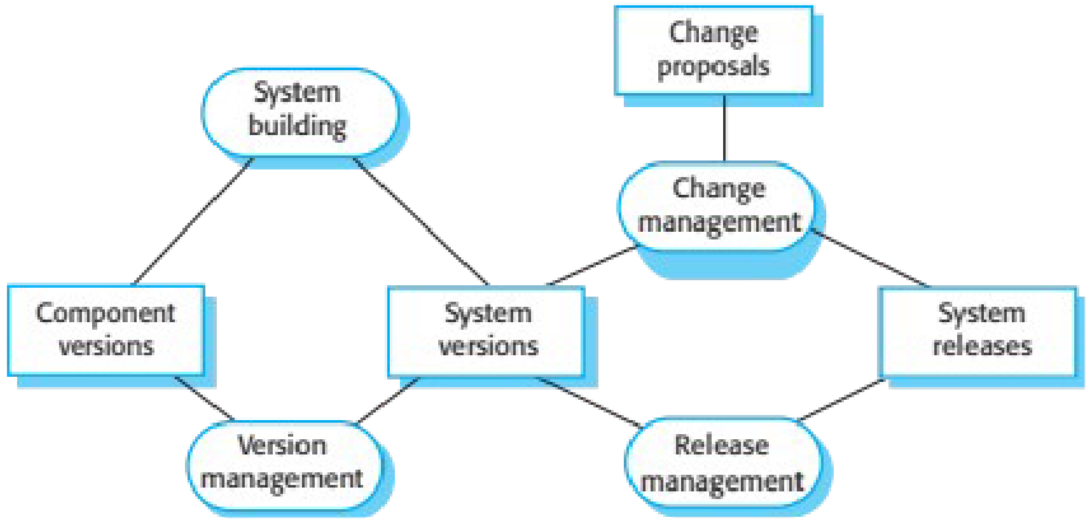
35.

Configuration Management
Het bijhouden, controleren en beheren van alle onderdelen van een softwaresysteem.
Dit zorgt ervoor dat je altijd weet:

- Welke versie van de code draait waar?
- Welke instellingen hoort bij welke omgeving?
- Wat is er veranderd, door wie en wanneer?

Configuration Management tools
Tools die dit automatisch beheren zoals: Git, Ansible, Puppet, Terraform, enz.

Configuration Management tool interaction
Hoe tools zoals Git, Ansible of Terraform samenwerken om software en infrastructuur te goed te beheren en te automatiseren.



36.

Open-source development is een benadering van softwareontwikkeling waarbij de broncode van een softwaresysteem wordt gepubliceerd en vrijwilligers worden uitgenodigd om deel te nemen aan het ontwikkelingsproces.

- De broncode is beschikbaar voor iedereen en gratis.
- Iedereen maf de code bekijken, gebruiken.

Voordelen:

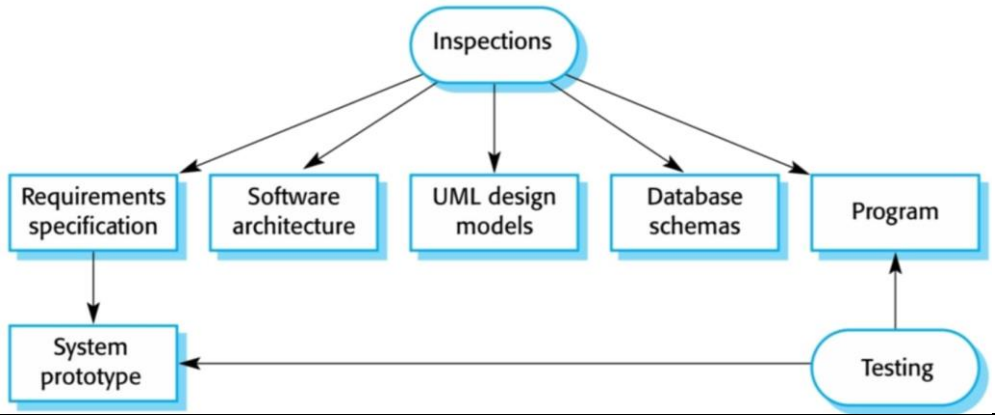
- Gratis te gebruiken
- Snelle innovatie: veel mensen denken mee
- Flexibel: je kunt het aanpassen aan je noden
- Veilig: fouten worden vaak sneller ontdekt.

Nadelen:
- Minder officiële support
- Niet alle projecten zijn evengoed onderhouden
- Je moet soms zelf documentatie of functies uitzoeken.

37.

Software Inspections zijn een manier om de code of documenten handmatig te controleren op fouten, fouten in logica of slechte stijl.
Hier is geen nood aan executies van het systeem en wordt voor elke implementatie gebruikt.
Voordelen:
- Tijdens het testen kunnen fouten andere fouten verbergen.
- Incomplete versies van het systeem kunnen worden geïnspecteerd zonder extra kosten.
- Naast het zoeken van programmeerdefecten kan een inspectie ook bredere kwaliteitskenmerken van een programma overwegen, zoals naleving van normen, draagbaarheid en onderhoudbaarheid.

Software Testing is het automatisch of handmatig uitvoeren van software om te kijken of alles werkt zoals het hoort.



38.

White box testing
je test de software terwijl je de interne code en logica kent.
Je kijkt dus binnenin het systeem.

Black box testing
Je test de software zonder te weten hoe de code eruitziet.
Je kijkt alleen naar wat er in gaat en wat er uit komt.

39.

Code coverage is een term voor verschillende metingen die temaken hebben met de percentage code dat is getest.

- Statements coverage: zijn alle regels code uitgevoerd?
- Function coverage: zijn alle functies/methoden minstens 1 keer uitgevoerd?

100% code coverage betekent niet dat alles getest is en er geen bugs meer zijn.

40.

Development Testing is het testen van software tijdens het ontwikkelen, nog voordat het naar de testers of de klant gaat.
Het doel is om vroeg fouten te vinden, zodat ze goedkoop en snel kinnen worden opgelost.

- Unit testen: functies/methodes worden getest
- Component testen: testen op interface
- System testen: volledig systeem (moet niet volledig zijn -> focus op component interactie.

41.

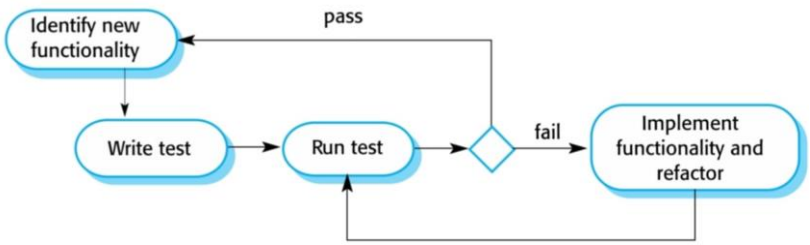
Automated testing
Waar mogelijk moet unit testen geautomatiseerd worden, zodat testen automatisch worden uitgevoerd en gecontroleerd zonder handmatige tussenkomst.
Testen bestaan uit;

- Setup part: input aanleg + verwachte output
- Oproepen testen van methoden/objecten
- Assertion part waart resultaat wordt vergeleken met verwachte resultaten.

42.

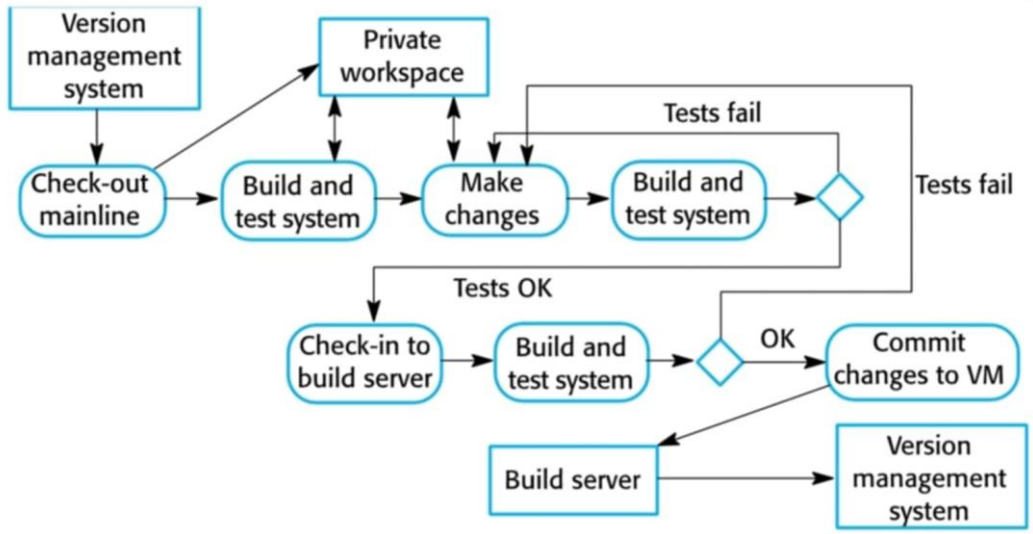
Test Drive Development is een benadering van programma-ontwikkeling waarbij je testen en code-ontwikkeling afwisselt.
Testen worden geschreven voor code en het 'slagen' van testen is cruciaal voor de ontwikkeling.
TDD was oorspronkelijk als deel van agile methodes.
Voordelen:

- Code coverage
- Regressie testen
- Simplified debuggen



Hoofdstuk 10

43.



Agile building

1. Haal het hoofdsysteem op uit het version management system naar de private workspace van de developer.
2. Bouw het systeem en voer geautomatiseerde testen uit om ervoor te zoergen dat het gebouwde systeem alle testen doorstaat.
   2.1 Als dit niet het geval is, is de build broken en moet je degene informareren die het laatste basissysteem heeft ingecheckt. Zij zijn verantwoordelijk voor het oplossen van het probleem.
3. Voer de wijzigingen door aan de systeemcomponenten.
4. Bouw het systeem in de private workspace en voer systeemtesten opnieuw uit.
   4.1 Als de testen falen, maak dan verdere aanpassingen.
5. Zodra het systeem de testen heeft volstaan, check je het in bij het buildsysteem, maar commit het niet als een nieuwe system baseline.
6. Bouw het systeem op de buildserver en voer de testen uit.
   6.1 je moet dit doen voor het geval dat anderen componenten hebben gewijzigd sinds jij het systeem hebt uigecheckt. Als dat het geval is, check dan de componenten uit die zijn mislukt en bewerk deze zodat de testen slagen in je private workspace.
7. Als het systeem de testen doorstaat op het build systeem, commit dan de wijzigingen die je hebt aangebracht als een nieuwe baseline in de system mainline.