

# WP3\_Del Deskstudie Cloud infrastructuur



# Samenvatting

In het kader van het Werkpakket 3 zijn we gestart met de ontwikkeling van een multi-cloud platform. Dit platform moet het mogelijk maken dat verschillende partners data en applicaties kunnen delen over organisatiegrenzen heen, zonder vendor lock-in en met behoud van data-soevereiniteit. De noodzaak voor dit platform vloeit voort uit de concrete behoeften van Werkpakket 3 (data en modellen) en Werkpakket 4 (reconstructie applicaties), die beide afhankelijk zijn van toegang tot gedistribueerde datasets en compute resources.

Het traditionele model, waarbij elke organisatie zijn eigen silo heeft van data en systemen, is ontoereikend voor de uitdagingen die grote volumes aan simulatiedata, real-time sensordata en historische reconstructies vereisen. Tegelijkertijd is een volledig gecentraliseerd model organisatorisch onhaalbaar: partners willen autonomie behouden over hun data en infrastructuur.

## Onderzoeksaanpak

Dit document beschrijft een systematische aanpak waarin we eerst hebben geleerd van bestaande multi-cloud initiatieven, vervolgens empirisch verschillende technische oplossingen hebben getest, en tot slot een concrete architectuur hebben ontworpen die praktisch implementeerbaar is.

**Fase 1:** Leren van anderen. We hebben drie belangrijke initiatieven grondig bestudeerd:

- **Haven** (Nederlandse gemeenten) leerde ons over praktische compliance, incrementele groei en community-gedreven ontwikkeling
- **GAIA-X** (Europese federatie) bood inzichten in trust frameworks, self-descriptions en het belang van code-first boven governance-first
- **OpenStack** (enterprise private cloud) gaf ons battle-tested architectuurpatronen voor multi-tenancy, resource abstractie en API design

Elk initiatief heeft sterke punten en valkuilen. Onze strategie is niet om één model te kopiëren, maar een hybride aanpak te ontwikkelen die het beste van alle drie combineert, toegespitst op onze behoeften.

**Fase 2:** Empirisch testen. We hebben vier cloud storage providers systematisch getest (Azure, Google Cloud, LeafCloud, on-premise) over verschillende dimensies:

- Performance. Upload/download snelheid, latency, concurrent access
- Kosten. Storage, transfer en operations kosten over 3-jaars horizon

De bevindingen waren genuanceerd: geen enkele provider is "beste" voor alle use cases. In plaats daarvan blijkt een tiered multi-cloud strategie optimaal - verschillende providers voor verschillende data temperaturen (hot/warm/cold) en gebruik patronen.

Cruciaal was het ontdekken van het "S3 API probleem": hoewel alle providers beweren S3-compatible te zijn, zijn er subtiele incompatibiliteiten in authenticatie, features en gedrag. De oplossing, MinIO Gateway als abstractielaag, maakt het mogelijk om één uniforme API te bieden naar applicaties terwijl achter de schermen meerdere providers worden ondersteund.

**Fase 3:** Architectuur ontwerp. Op basis van geleerde lessen en testresultaten hebben we een zes-lagen architectuur ontworpen. Dit kan gezien worden als een referentie architectuur voor verdere implementatie.

### **Belangrijkste bevindingen**

- Multi-cloud is noodzakelijk, niet alleen wenselijk.

Onze analyse toont aan dat een single-provider strategie duurder zou zijn dan onze multi-cloud aanpak. Belangrijker nog: het zou ons kwetsbaar maken voor vendor lock-in. Multi-cloud is een risicomitigatie strategie die ook cost benefits oplevert.

- Abstractielagen zijn de sleutel tot succes.

De MinIO Gateway, Kubernetes, en API Gateway vormen abstractielagen die de complexiteit van multi-cloud verbergen voor applicaties. Zonder deze lagen zou elke applicatie provider-specific code moeten bevatten, wat onhoudbaar is. De 2-5ms latency overhead die abstractie introduceert is ruimschoots acceptabel gegeven de flexibiliteit die het oplevert.

- Security moet geïntegreerd zijn, niet toegevoegd.

Ons zero-trust model - waarbij elk request wordt geauthentiseerd en geautoriseerd, en alles wordt gelogd - is niet een feature maar een architectureel fundament. Multi-party samenwerking zonder fundamentele security is een groot risico.

- Community governance is even belangrijk als techniek.

Haven's succes komt niet alleen door goede techniek maar ook door community-gedreven ontwikkeling. GAIA-X's tragere adoptie komt deels door te veel governance vooraf. We adopteren Haven's model: start met code, formaliseer governance geleidelijk.

# Inhoud

	<b>Samenvatting</b>	<b>3</b>
	<b>Inhoud</b>	<b>5</b>
<b>1</b>	<b>Hybride multicloud oplossingen</b>	<b>7</b>
1.1	Haven	7
1.1.1	Haven als standaard	7
1.1.2	Technisch	8
1.1.3	Common Ground	8
1.1.4	Haven+	9
1.1.5	Leringen van Haven	9
1.2	GAIA-X	10
1.2.1	Kernarchitectuur van GAIA-X	10
1.2.2	GAIA-X + International Data Spaces (IDS)	11
1.2.3	Leringen van GAIA-X	12
1.3	OpenStack	13
1.3.1	OpenStack architectuur overzicht	13
1.3.2	OpenStack Cascading	14
1.3.3	Leringen van OpenStack	14
1.4	Vergelijking	16
1.4.1	Het fundament: Haven's pragmatisme	16
1.4.2	De federatie-laag: GAIA-X's visie	17
1.4.3	De technische diepgang: OpenStack's architectuur	17
<b>2</b>	<b>Vergelijkend onderzoek cloud storage oplossingen</b>	<b>19</b>
2.1	Context	19
2.2	Selectie van storage providers	19
2.3	Test methodologie	20
2.3.1	Test datasets	20
2.3.2	Test scenarios	21
2.3.3	Cost analysis methodologie	21
2.4	Performance test resultaten	23
2.4.1	Upload performance	23
2.4.2	Download performance	24
2.4.3	Concurrent access performance	25
2.5	Cost analysis resultaten	25
2.5.1	Storage costs per provider	25
2.5.2	Transfer costs	26
2.5.3	Operation kosten	26
2.5.4	Total Cost of Ownership (3 jaar)	26
2.5.5	Break-even Analysis	27
2.6	Het S3 API probleem	28
2.6.1	De S3 API en zijn implementaties	28
2.6.2	Authenticatie en signing	28

2.6.3	Missende features	29
2.6.4	Subtiele verschillen	29
2.6.5	SDK fragmentatie	29
2.6.6	Impact op applicatieontwikkeling	30
2.7	2.7 MinIO als oplossing	30
2.7.1	Wat is MinIO?	30
2.7.2	Architectuur van de Gateway	31
2.7.3	Feature normalisatie	31
2.7.4	Beperkingen en trade-offs	31
<b>3</b>	<b>Technische architectuur</b>	<b>33</b>
3.1	Inleiding	33
3.2	3.2 Architectuur ontwerpprincipes	33
3.2.1	Principe 1: Federatie boven Centralisatie	34
3.2.2	Principe 2: Open Source First	35
3.2.3	Principe 3: Gelaagde Architectuur	36
3.2.4	Principe 4: Security by Design	37
3.2.5	Principe 5: Developer Friendly	38
3.2.6	Principe 6: Cost Conscious	39
3.3	De Zes-Lagen Architectuur	41
3.4	Layer 1: Multi-Cloud Infrastructure	42
3.4.1	3.4.1 Verantwoordelijkheid en Scope	42
3.5	Layer 2: Data Abstraction Layer	42
3.5.1	Verantwoordelijkheid en Scope	42
3.6	Layer 3: Security & Identity Layer	42
3.6.1	Verantwoordelijkheid en Scope	42
3.7	Layer 4: Orchestration Layer	43
3.7.1	Verantwoordelijkheid en Scope	43
3.8	Layer 5: Federation & API Layer	43
3.8.1	Verantwoordelijkheid en Scope	43
3.9	3.9 Layer 6: Application Layer	44
3.9.1	3.9.1 Verantwoordelijkheid en Scope	44

# 1 Hybride multicloud oplossingen

## 1.1 Haven

De transitie van ICT-infrastructuur naar “cloud” is volop gaande. Ook bij gemeenten is dit een steeds belangrijker wordend thema. Soms vanuit een eigen visie maar vaker door de marktontwikkelingen gedreven. Daarmee is het gebruik van cloud onmisbaar en onontkoombaar.

Er zijn diverse redenen om gebruik te maken van een clouddienst, zoals: snelheid, schaalbaarheid, tijdelijkheid, beveiliging, kosten en flexibiliteit. Het uitgangspunt is daarbij dat je de leverancier zoveel mogelijk het werk laat doen en bijvoorbeeld eist dat de leverancier bewijst dat hij aan de gestelde eisen en voorwaarden voldoet, bij zowel de inrichting als in de operationele fase. Daarbij dienen de continuïteit van dienstverlening, privacybescherming en informatiebeveiliging te zijn geborgd, conform wet- en regelgeving; ‘public’ waar kan, ‘private’ waar moet.

Gemeenten hebben en houden verantwoordelijkheid, beschikking en eigenaarschap over de eigen data. Gemeenten zijn als opdrachtgever volledig in staat om bij veranderende behoeften en wensen van richting of leverancier te veranderen.

Dit doen gemeenten samen, via “Haven”. Dit initiatief begon in 2019 als een technische standaard voor enkele gemeenten die cloudonafhankelijk wilden werken en staat inmiddels in de Nederlandse Digitaliseringsstrategie als voorbeeld van een ‘versneller’ die voor de hele overheid moet gaan gelden.

Gemeenten gebruiken applicaties voor het uitvoeren van hun taken. Die applicaties draaien op een IT infrastructuur (zoals rekenkracht, opslag en netwerkverbindingen). Iedere gemeente heeft zijn IT infrastructuur anders georganiseerd. Bij de ene gemeente draait bijvoorbeeld veel lokaal en bij de ander juist meer in de cloud. Applicaties moeten worden aangepast aan de infrastructuur waarop ze draaien. Dat maakt het voor gemeenten lastig om samen applicaties te ontwikkelen en deze snel in te zetten bij alle gemeenten.

### 1.1.1 Haven als standaard

Haven biedt daarvoor een oplossing. Concreet: Haven legt verbinding tussen ontwikkelde applicaties en iedere gemeentelijke IT infrastructuur. Met Haven kunnen gemeenten applicaties overal hosten zonder dat zij daarvoor hun IT infrastructuur hoeven aan te passen. Daardoor kunnen gemeenten applicaties samen ontwikkelen en snel inzetten bij alle gemeenten. Dit zorgt onder meer voor uniformiteit, lagere kosten en minder afhankelijkheid van leveranciers.

### 1.1.2 Technisch

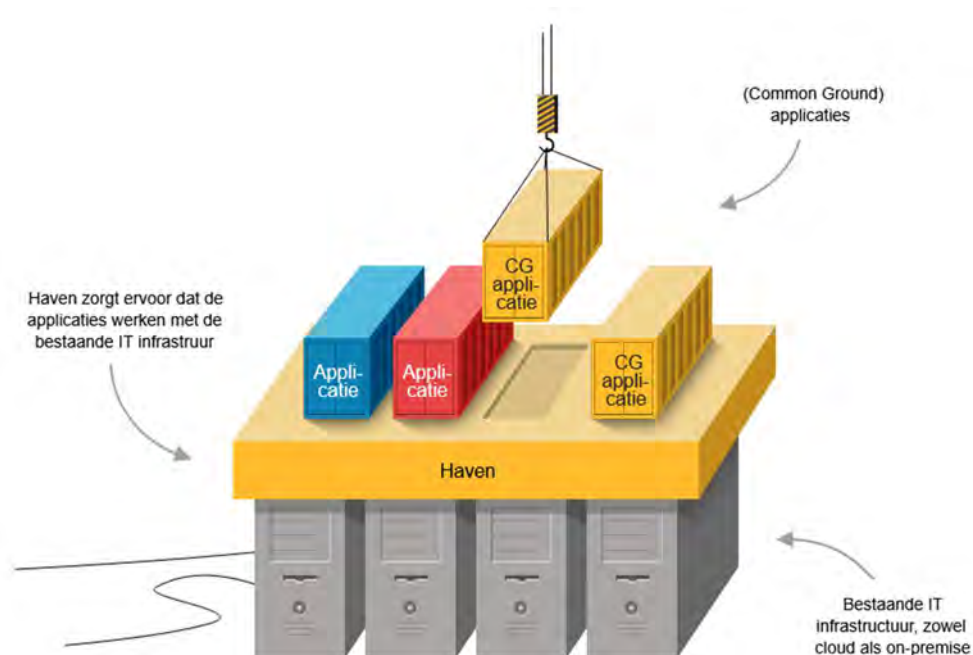
Haven schrijft een specifieke configuratie van Kubernetes voor die dient te worden geïmplementeerd op bestaande technische infrastructuur, bijvoorbeeld een cloud of on-premise platform.

De voorgeschreven configuratie zorgt ervoor dat iedere Haven omgeving functioneel gelijk is ongeacht de onderliggende technische infrastructuur. Zie het als een abstractielaag die resulteert in een gezamenlijk vertrekpunt.

Dit brengt diverse voordelen met zich mee: uniformiteit in technische infrastructuur, uitwisselbaarheid van toepassingen, leveranciersonafhankelijkheid, platformafhankelijkheid en kostenreductie.

Implementatie van een Haven omgeving kan door gemeenten zelf worden gedaan of door een leverancier naar keuze.

Haven is dus geen 'all inclusive' cloud oplossing en ook geen security baseline.



### 1.1.3 Common Ground

De architectuurprincipes van Common Ground liggen aan de basis van de ontwikkeling van een landelijk federatief stelsel. Dit stelsel is onderdeel van interbestuurlijke afspraken over het gebruik van data en moet verantwoord en meervoudig datagebruik mogelijk maken. Common Ground is de overkoepelende visie en architectuur, terwijl Haven een technische implementatie is van één specifiek principe uit Common Ground.

Haven is een project dat is onderdeel van de Common Ground architectuurvisie, waarbij Common Ground het bredere kader vormt voor de gewenste informatiehuishouding van gemeenten.

Common Ground is voltooid als de gegevens in alle relevante informatiesystemen worden gescheiden van de applicaties en processen waarin ze worden gebruikt, om vervolgens vanuit een centrale bron voor de verschillende ICT-oplossingen beschikbaar te komen.

*Kernprincipes van Common Ground:*

1. "Seperation of concerns" - Data los van applicaties
2. Data bij de bron - Geen kopieën maar directe toegang
3. Gelaagde architectuur - Het vijf-laagsmodel
4. Open source first - Kennisdeling en vendor lock-in vermijden
5. Agile ontwikkeling - Iteratief en incrementeel werken
6. Community gedreven - Samen bouwen met gemeenten en leveranciers
7. Moderne technologie - Meegroeien met technologische ontwikkelingen

*Haven: De Technische Implementatie*

Haven implementeert specifiek het "Moderne IT" principe van Common Ground door: Gebruik te maken van moderne technologie zoals container- en serverless technologie, waarbij handelingen zoveel mogelijk worden geautomatiseerd zoals testen en deployen van software, monitoren, loggen en notificeren

#### **1.1.4 Haven+**

Haven+ is een referentiearchitectuur en implementatie die extra platformdiensten toevoegt aan de standaard Haven Kubernetes-omgeving. Het richt zich op het snel beschikbaar maken van essentiële bouwstenen voor productieomgevingen, zoals monitoring, authenticatie, certificatenbeheer en databasediensten.

Haven+ maakt het eenvoudiger voor gemeenten om een veilige, beheersbare en schaalbare cloudomgeving in te richten volgens moderne DevOps-praktijken, GitOps en open standaarden.

Met Haven+ beschik je snel over monitoring (metrics, logging en tracing), authenticatie, databases, certificaatbeheer en secret management.

#### **1.1.5 Leringen van Haven**

Haven biedt een uitstekend voorbeeld van hoe een consortium (gemeenten) succesvol een open source, multi-cloud infrastructuur heeft opgezet. De focus op standaardisatie, compliance checking en geleidelijke soevereiniteit zijn belangrijk voor succes.

De belangrijkste les: start met duidelijke specificaties en build incrementeel, net zoals Haven vanaf 2019 is gegroeid van enkele gemeenten naar 40+ organisaties.

- Definieer eerst je "Common Ground" (visie op interoperabiliteit)
- Bouw dan je "Haven" (technische implementatie)
- Plan een "+" variant voor advanced features
- Zorg voor compliance tooling vanaf dag 1
- Organiseer community governance parallel aan techniek

## 1.2 GAIA-X

Gaia-X is een initiatief om een gefedereerde veilige data-infrastructuur voor Europa te ontwikkelen, waarbij data worden gedeeld met gebruikers die controle behouden over hun data-toegang en -gebruik, om Europese digitale soevereiniteit te waarborgen. Het resultaat zal geen cloud zijn. Het wordt een gefedereerd systeem dat vele cloud service providers en gebruikers verbindt in een transparante omgeving en zal de Europese data-economie van morgen aandrijven

GAIA-X beschrijft concepten die nodig zijn om GAIA-X-conforme data- en/of infrastructuur-ecosystemen te realiseren, die Providers, Consumers en Services integreren die essentieel zijn voor deze interactie, inclusief het waarborgen van identiteiten, implementeren van trust mechanismen en het bieden van usage control over data-uitwisseling en Compliance

### 1.2.1 Kernarchitectuur van GAIA-X

GAIA-X beschrijft concepten die nodig zijn om GAIA-X-conforme data- en/of infrastructuur-ecosystemen te realiseren, die Providers, Consumers en Services integreren die essentieel zijn voor deze interactie, inclusief het waarborgen van identiteiten, implementeren van trust mechanismen en het bieden van usage control over data-uitwisseling en Compliance

*De vijf kernprincipes:*

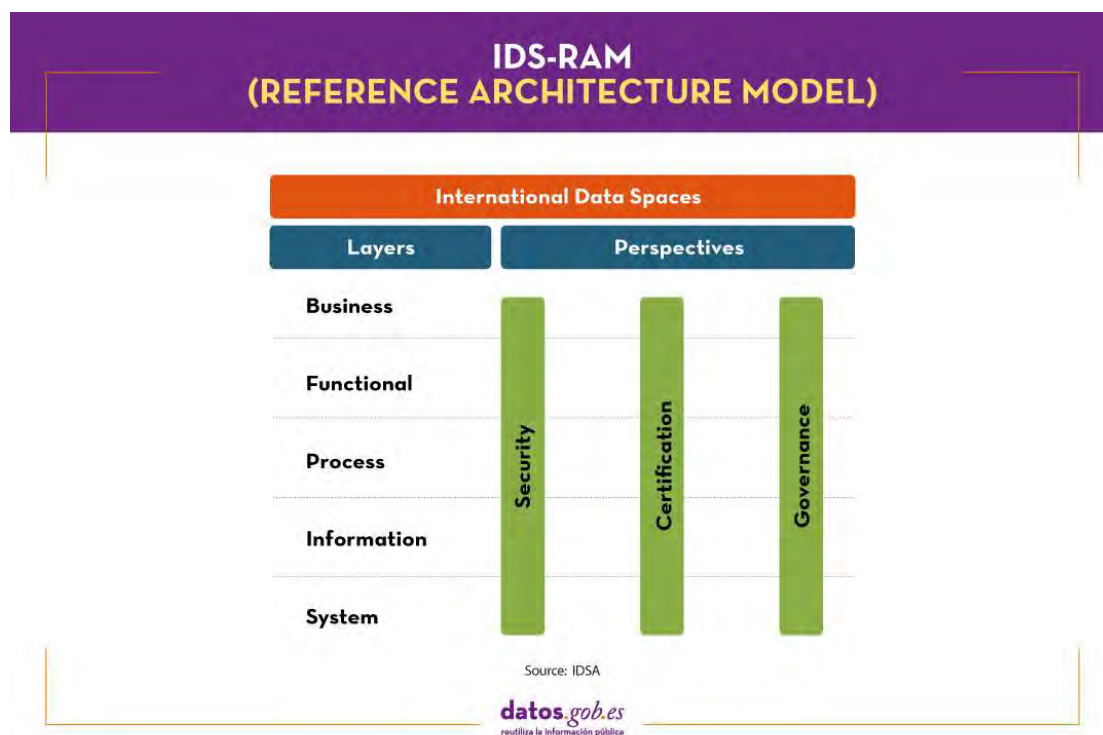
1. Data Soevereiniteit
  - Data blijft bij de eigenaar
  - Delen gebeurt onder gedefinieerde voorwaarden
  - Gebruikers behouden controle over toegang en gebruik
  - GDPR-compliant by design
2. Federatie in plaats van Centralisatie
  - Geen nieuwe cloud-infrastructuur
  - Verbinden van bestaande providers
  - Gebaseerd op gemeenschappelijke standaarden
  - Gedecentraliseerde architectuur
3. Transparantie & Verificatie
  - Self-descriptions van services
  - Verificatie framework voor compliance
  - Transparante governance
  - Certificatie mechanismen
4. Interoperabiliteit
  - Gemeenschappelijke API standaarden
  - Metadata catalogus
  - Service discovery mechanismen
  - Cross-provider orchestration
5. Trust Framework
  - Identity management
  - Access control
  - Usage policies
  - Audit trails

## 1.2.2 GAIA-X + International Data Spaces (IDS)

GAIA-X gebruikt de International Data Spaces Reference Architecture (**IDS-RAM**) om ervoor te zorgen dat data usage controls worden geboden en compliance wordt gewaarborgd. Deze relatie is fundamenteel voor het begrijpen van hoe GAIA-X data-soevereiniteit technisch realiseert. Waar GAIA-X de overkoepelende visie en governance framework levert voor Europese cloud federatie, biedt IDS de concrete technische architectuur voor veilige, gecontroleerde data-uitwisseling.

International Data Spaces (IDS) is een referentie-architectuur en een set van specificaties ontwikkeld door de Fraunhofer Gesellschaft (Duitsland) samen met meer dan 100 bedrijven en onderzoeksinstituten. Het project startte in 2015 met als doel een architectuur te creëren waarin organisaties data kunnen delen terwijl ze volledige controle behouden over hoe hun data gebruikt wordt.

Het fundamentele probleem dat IDS oplost: traditioneel betekent "data delen" dat je een kopie weggeeft en alle controle verliest. Eenmaal gekopieerd, kun je niet meer afdwingen dat data alleen voor specifieke doeleinden wordt gebruikt, niet langer bewaard wordt dan toegestaan, of niet naar derde partijen gaat. IDS introduceert "**data sovereignty by design**" - technische mechanismen die data-eigenaren controle geven over data usage, zelfs nadat data hun organisatie heeft verlaten.



IDS definieert een Clearing House, onafhankelijke service die alle data transactions logged. Denk aan het als a blockchain-achtige audit trail.

Functionaliteit:

- Provider logs: "I sent dataset X to consumer Y at timestamp T"
- Consumer logs: "I received dataset X from provider Y at timestamp T"
- Clearing House: beide logs matchen (consistency check)
- Immutable log voor geschil resolutie

Use case: Provider beweert "Consumer downloadt mijn data zonder permissie". Consumer beweert "Ik heb nooit die data ontvangen". Clearing House log toont de waarheid.

GAIA-X heeft IDS niet volledig overgenomen maar wel key componenten geadopteerd. GAIA-X gebruikt IDS als technische foundation, maar voegt lagen toe voor:

1. Federated Identity IDS had geen specifieke identiteit layer. GAIA-X voegt toe:
  - Self-Sovereign Identity (SSI) met verifiable credentials
  - Federated identity providers (niet één centrale IdP)
  - European Digital Identity (eIDAS) integratie
2. Compliance & Certification GAIA-X definieert compliance framework:
  - GAIA-X Label
  - GAIA-X Lighthouse
  - Automated compliance verificatie via credentials
3. Multi-Cloud Native IDS is ontworpen for on-premise omgevingen. GAIA-X is cloud-native:
  - Cloud service self-descriptions (VMs beschrijvingen, Opslag beschrijvingen)
  - Sovereign cloud attest
  - Cloud provider certificatie

### 1.2.3 Leringen van GAIA-X

GAIA-X demonstreert hoe een ambitieuze Europese federatie-aanpak werkt aan data-soevereiniteit en interoperabiliteit over landsgrenzen heen. De focus op trust frameworks, self-descriptions en gedecentraliseerde architectuur bieden waardevolle inzichten voor consortium-samenwerking.

De belangrijkste les: start met werkende technologie en voeg governance geleidelijk toe. GAIA-X toont aan dat te veel governance vooraf de adoptie vertraagt - begin pragmatisch, formaliseer naarmate je groeit.

#### Wat werkt:

- Federatie-architectuur in plaats van centralisatie voorkomt vendor lock-in en machtconcentratie
- Self-description mechanisme maakt datasets en services vindbaar zonder centrale catalogus
- Trust framework met verifiable credentials schept vertrouwen tussen onbekende partijen
- Usage control policies geven data-eigenaren controle over hoe hun data gebruikt wordt
- Identity federation met SSI (Self-Sovereign Identity) voor privacy-vriendelijke authenticatie

#### Wat vermijden:

- Te veel tijd besteden aan governance-structuren voordat er werkende code is
- Perfectie nastreven - start met MVP en itereer
- Technische complexiteit die kleine organisaties uitsluit
- Association-model opzetten voordat de waarde bewezen is

## 1.3 OpenStack

OpenStack is een open-source cloud platform dat gedistribueerde compute, netwerk en storage resources beheert, deze aggregeert in pools, en on-demand provisioning van virtuele resources mogelijk maakt via een self-service portal. In tegenstelling tot GAIA-X (dat een federatie-standaard is) of Haven (dat een deployment model biedt), geeft OpenStack je daadwerkelijk de technologie om je eigen cloud te bouwen - vergelijkbaar met wat AWS, Azure of Google Cloud doen, maar dan volledig open source en onder jouw controle.

OpenStack is ontstaan in 2010 uit een samenwerking tussen Rackspace en NASA, en is sindsdien uitgegroeid tot een van de grootste open source projecten ter wereld. Meer dan 500 bedrijven dragen bij, en het draait in productie bij organisaties als CERN, Walmart, Bloomberg, en talloze telecombedrijven. Deze mature codebase vertegenwoordigt vijftien jaar aan battle-tested ervaring in het bouwen van cloud infrastructuur op enterprise niveau.

Met OpenStack bouw je effectief je eigen AWS. Je hebt volledige controle, maar ook volledige verantwoordelijkheid.

### 1.3.1 OpenStack architectuur overzicht

OpenStack bestaat uit een ecosysteem van ongeveer 30+ componenten (projecten), waarvan er 6 core zijn die vrijwel elke deployment gebruikt. Elk component is modulair - je kunt ze onafhankelijk deployen, upgraden of vervangen. Deze modulariteit is een kracht (flexibiliteit) maar ook een uitdaging (complexiteit).

#### *De zes core componenten*

- Nova (Compute). Nova is het hart van OpenStack - het beheert de lifecycle van virtuele machines. Denk aan het als de AWS EC2 equivalent.
- Neutron (Networking). Neutron biedt "networking as a service" - het creëert virtuele netwerken, subnets, routers, firewalls als software-defined entities.
- Swift (Object Storage). Swift is OpenStack's object storage systeem - vergelijkbaar met AWS S3. Het is ontworpen voor durability, availability en scalability.
- Cinder (Block Storage). Terwijl Swift object storage is (files), is Cinder block storage (disks). Het is als AWS EBS - virtuele disks die je aan VMs kunt attachen.
- Keystone (Identity). Keystone is de identity service - authentication, authorization en service discovery voor alle OpenStack componenten.
- Horizon (Dashboard). Horizon is de web UI bovenop alle andere OpenStack componenten. Het biedt een self-service portal voor eindgebruikers en admins.

#### *Aanvullende belangrijke componenten*

Heat is OpenStack's Infrastructure-as-Code systeem - vergelijkbaar met AWS CloudFormation of Terraform.

Functionaliteit:

- Template-based: Declaratief definiëren van complete stacks (VMs + networks + storage + load balancers)
- Resource dependencies: Automatisch bepalen in welke volgorde resources worden gecreëerd
- Stack updates: Wijzigingen aan infrastructuur via template updates
- Auto-scaling: Templates kunnen auto-scaling groups definiëren

### 1.3.2 OpenStack Cascading

Een van OpenStack's interessantste patterns voor grote deployments is **Cascading**, een hiërarchisch model waarbij één top-level OpenStack regio meerdere onderliggende regio's aanstuurt.

*Het Probleem dat Cascading Oplost*

Stel je voor: een grote organisatie (telecom operator, overheid) wil OpenStack deployen over 50 datacenters wereldwijd. De naïeve aanpak:

#### Optie A: Eén monolithisch OpenStack

- Eén control plane voor alle 50 datacenters
- **Probleem:** Schaalbaarheidsissues. Keystone database heeft miljoenen entries. Nova scheduler moet kiezen uit duizenden hosts. Latency tussen datacenters maakt synchrone operations traag.

#### Optie B: 50 onafhankelijke OpenStack installaties

- Elk datacenter heeft eigen OpenStack
- **Probleem:** No unified view. User moet 50 verschillende endpoints kennen. Cross-datacenter workloads zijn onmogelijk. Veel complexiteit

#### Optie C: Koop/Bouw Cloud Management Platform

- Third-party platform (VMware vRealize, RedHat CloudForms) bovenop discrete clouds
- **Probleem:** Proprietary lock-in. OpenStack API ecosystem gaat verloren. Duur.

OpenStack Cascading biedt een vierde optie: hiërarchische structuur met OpenStack op elk niveau.

*Voordelen van de Cascading oplossing*

- Eén endpoint, maar workloads distributed over many regions
- Elke region heeft eigen database, eigen scheduler. Top-level hoeft alleen delegation te doen.
- Als region-2 faalt, blijven region-1 en region-3 werken.
- Verschillende regions kunnen verschillende hardware, hypervisors, storage backends hebben.

### 1.3.3 Leringen van OpenStack

OpenStack biedt vijftien jaar aan battle-tested ervaring in het bouwen van private clouds op enterprise-niveau. Hoewel het platform zelf te zwaar is voor onze use cases, zijn de architectuurprincipes en design patterns van onschatbare waarde.

De belangrijkste les: neem de architectuur-wijsheid mee, niet de implementatie. OpenStack heeft hard geleerd hoe je complexe, multi-tenant infrastructuur bouwt - leer van hun oplossingen zonder hun operationele last over te nemen.

**Wat werkt:**

- Resource abstractie - hardware-verschillen verbergen achter uniforme APIs
- Multi-tenancy design - strikte isolatie tussen verschillende organisaties op gedeelde infrastructuur
- Modulaire architectuur - onafhankelijke componenten die je kunt vervangen zonder alles te verbouwen
- API-first benadering - alles is toegankelijk via goed-gedocumenteerde REST APIs
- Quota en resource management - eerlijke verdeling van compute/storage tussen tenants
- Rolling upgrades - zero-downtime updates door componenten incrementeel te vervangen

**Wat vermijden:**

- Volledige OpenStack deployment - te complex, te duur om te runnen
- Feature bloat - OpenStack kan alles, maar dat betekent niet dat je alles nodig hebt
- Eigen ontwikkeling van infrastructuur-code - gebruik bestaande cloud providers waar mogelijk
- Zware operationele overhead - volautomatisch beheer is moeilijker dan het lijkt

## 1.4 Vergelijking

Na bestudering van Haven, GAIA-X en OpenStack wordt duidelijk dat elk initiatief unieke sterke punten heeft die voortkomen uit hun specifieke context en doelstellingen. Haven opereert op nationale schaal binnen Nederland en excelleert in praktische compliance en snelle adoptie. GAIA-X daarentegen denkt EU-breed en biedt een robuust trust framework voor cross-border samenwerking, maar worstelt met de complexiteit van multi-nationale governance. OpenStack tenslotte biedt ongekeerde technische diepte na vijftien jaar enterprise-ervaring, maar brengt een operationele last met zich mee die voor veel organisaties onhaalbaar is.

ASPECT	HAVEN	GAIA-X	OPENSTACK
Scope	NL Municipalities	EU Federation	Organization Cloud
Maturity	Growing	Emerging	Mature
Complexity	Medium	High	Very High
Governance	VNG	Association	Community
Best Feature	Compliance Checker	Trust Framework	Technical Depth

Voor ons consortium betekent dit dat een pure adoptie van één van deze modellen suboptimaal zou zijn. Haven's focus op Nederlandse gemeenten geeft een te kleine scope. GAIA-X's governance-first aanpak zou onze time-to-market te veel vertragen. En OpenStack's volledige stack zou onze operationele capaciteit overschrijden. In plaats daarvan kiezen we voor een **hybride aanpak** die het beste van drie werelden combineert, aangepast aan onze specifieke context.

### 1.4.1 Het fundament: Haven's pragmatisme

Van Haven nemen we de praktische, incrementele aanpak over. Haven heeft bewezen dat je kunt beginnen met een kleine groep early adopters en geleidelijk uitgroeien tot een substantieel netwerk - van enkele gemeenten in 2019 naar meer dan 40 organisaties vijf jaar later. Deze groeistrategie past perfect bij onze consortium-structuur waarin partners op verschillende momenten willen instappen.

Concreet betekent dit dat we Kubernetes als ons deployment platform kiezen, niet omdat het de enige optie is, maar omdat het een bewezen, vendor-neutrale standaard is waar Haven al jaren ervaring mee heeft. We adopteren ook Haven's concept van een compliance checker - een geautomatiseerde tool die controleert of implementaties aan afgesproken standaarden voldoen. Dit voorkomt dat partners per ongeluk incompatibele systemen bouwen en maakt onboarding van nieuwe partners substantieel eenvoudiger.

Cruciaal is ook Haven's community-gedreven ontwikkelmodel. In plaats van een top-down architectuur op te leggen, bouwen we samen met de partners aan oplossingen voor concrete

problemen. Dit zorgt voor buy-in en relevantie, maar vereist wel discipline om niet in feature-chaos te vervallen. Haven's ervaring met prioritering via een gezamenlijke roadmap is hier waardevol.

#### 1.4.2 De federatie-laag: GAIA-X's visie

Waar Haven ons het praktische fundament geeft, biedt GAIA-X de conceptuele architectuur voor federatie. Het fundamentele inzicht van GAIA-X - dat je geen nieuwe cloud hoeft te bouwen maar bestaande clouds moet kunnen federeren - is direct toepasbaar op onze situatie. Partners hebben al bestaande infrastructuur bij verschillende cloud providers; ons doel is om deze te verbinden, niet te vervangen.

Van bijzonder belang is GAIA-X's trust framework. In een consortium waar organisaties data moeten delen zonder volledig vertrouwen, biedt het concept van verifiable credentials een elegante oplossing. Een dataset kan zichzelf beschrijven (metadata, licentie, gebruiksvoorwaarden) op een manier die cryptografisch verifieerbaar is. Dit maakt discovery mogelijk zonder centrale autoriteit, en stelt data-eigenaren in staat om controle te behouden over hoe hun data gebruikt wordt.

We implementeren ook GAIA-X's self-description mechanisme voor onze datasets. Elke dataset krijgt een gestandaardiseerde beschrijving die niet alleen technische metadata bevat (formaat, grootte, locatie), maar ook semantische informatie (wat stelt deze data voor?) en juridische informatie (wie mag het gebruiken, onder welke voorwaarden?). Dit lost een praktisch probleem op: als een WP4-applicatie GTSM-data nodig heeft, kan deze automatisch ontdekken welke partner die data heeft, of toegang toegestaan is, en hoe de data op te halen.

Tegelijkertijd nemen we GAIA-X's waarschuwing ter harte: begin niet met governance-structuren maar met werkende code. GAIA-X's relatief trage adoptie komt deels doordat te veel energie ging naar het definiëren van besluitvormingsprocessen voordat er concrete waarde te demonstreren viel. Wij draaien dit om: eerst bouwen we iets dat werkt en waarde levert, dan formaliseren we de governance eromheen.

#### 1.4.3 De technische diepgang: OpenStack's architectuur

OpenStack draait al vijftien jaar in productie bij enkele van 's werelds grootste organisaties. Die battle-tested ervaring is een goudmijn aan lessen over hoe je multi-tenant, gedistribueerde infrastructuur bouwt die écht werkt onder stress.

Het meest waardevol is OpenStack's resource abstractie-strategie. OpenStack's Nova (compute), Neutron (network) en Cinder (storage) componenten laten zien hoe je hardware-diversiteit kunt verbergen achter uniforme APIs. Dit principe passen we toe met MinIO als storage-abstractie-laag: applicaties praten tegen één S3-compatibele API, maar daarachter kunnen we Azure Blob Storage, Google Cloud Storage, of on-premise storage aansluiten zonder dat applicaties dit merken.

Essentieel is ook OpenStack's multi-tenancy architectuur. OpenStack's hiërarchie van domains → projects → users → roles biedt een gedegen model voor hoe je meerdere organisaties veilig op dezelfde infrastructuur laat werken. We implementeren een vergelijkbaar model waar elke partner zijn eigen namespace krijgt binnen Kubernetes, met resource quotas die eerlijk gebruik garanderen en network policies die isolatie afdwingen.

Een partner kan niet per ongeluk (of opzettelijk) resources van een andere partner zien of beïnvloeden.

Van OpenStack's Heat orkestratie leren we hoe je Infrastructure-as-Code goed doet. Heat templates zijn declaratief - je beschrijft de gewenste eindtoestand, niet de stappen om daar te komen. Dit maakt deployment reproduceerbaar en maakt het eenvoudig om identieke omgevingen op te zetten bij verschillende cloud providers. We adopteren dit patroon met Helm charts en Terraform modules.

Cruciaal is echter dat we OpenStack zelf niet deployen. De operationele complexiteit van het draaien van een volledige OpenStack-installatie zou onevenredig zijn voor onze use case. In plaats daarvan "stelen" we hun architectuurprincipes en passen deze toe met lichtere, moderne tools. Waar OpenStack Keystone heeft voor identity, gebruiken wij Keycloak. Waar OpenStack Swift heeft voor object storage, gebruiken wij MinIO. We krijgen vergelijkbare functionaliteit met een fractie van de operationele overhead.

## 2 Vergelijkend onderzoek cloud storage oplossingen

### 2.1 Context

Een van de fundamentele uitdagingen in een multi-cloud architectuur is het efficiënt en betrouwbaar opslaan van data over verschillende cloud providers heen. Voor ons komt deze uitdaging met specifieke eisen: we werken met grote volumes aan simulatiedata (GTSM modelresultaten), tijdseries van sensoren en AIS-tracking, historische datasets voor reconstructies, en real-time streams van operationele systemen. Deze data moet toegankelijk zijn voor verschillende partners die mogelijk bij verschillende cloud providers gehost zijn, moet voldoen aan AVG-eisen rondom data-lokatie, en moet kostenefficiënt opgeslagen kunnen worden afhankelijk van hoe 'hot' of 'cold' de data is.

Voordat we een architectuurbeslissing konden nemen over onze storage-strategie, hebben we een systematisch vergelijkend onderzoek uitgevoerd naar vier verschillende storage backends: Azure Blob Storage, Google Cloud Storage, LeafCloud (een Nederlandse duurzame cloud provider), en on-premise storage via MinIO. Dit hoofdstuk documenteert de methodologie, bevindingen en conclusies van dat onderzoek, en legt uit hoe we op basis hiervan tot een hybride storage-architectuur zijn gekomen die het beste van meerdere werelden combineert.

### 2.2 Selectie van storage providers

De keuze voor deze vier specifieke providers was niet willekeurig, maar gebaseerd op strategische overwegingen die relevant zijn voor ons consortium.

Azure Blob Storage werd gekozen omdat verschillende partners al Azure-diensten gebruiken voor andere doeleinden, en Azure heeft een datacenter in West Europa wat latency-voordelen biedt. Bovendien heeft Deltares al ervaring met Azure, wat de learning curve verkort. Azure is ook een van de "big three" hyperscalers, wat betekent dat hun diensten mature zijn en uitgebreide functionaliteit bieden.

Google Cloud Storage werd meegenomen omdat Google bekend staat om hoge performance en omdat hun europe-west4 regio fysiek in Nederland gevestigd is, wat relevant is voor data-soevereiniteit. Google heeft ook sterke machine learning integratie, wat mogelijk nuttig is voor toekomstige data-analyse use cases. Bovendien biedt het een tweede hyperscaler optie, wat vendor lock-in risico's verkleint.

LeafCloud is een minder voor de hand liggende keuze, maar strategisch interessant om meerdere redenen. Ten eerste is het een Nederlandse, Europese provider, wat governance en AVG-compliance vereenvoudigt. Ten tweede draait LeafCloud volledig op duurzame energie en gebruikt ze restwarmte voor stadsverwarming, wat aansluit bij maatschappelijke verantwoordelijkheid. Ten derde vertegenwoordigt LeafCloud de categorie "alternatieve cloud providers" - kleinere spelers die vaak competitiever geprijsd zijn en flexibeler in partnership. Als LeafCloud werkt, werken waarschijnlijk ook andere S3-compatibele providers.

On-premise storage via MinIO werd getest omdat sommige partners eigen datacenters hebben waar ze historische data al opslaan, en omdat voor bepaalde gevoelige datasets data-soevereiniteit absoluut vereist is. On-premise storage vertegenwoordigt ook de "traditionele" aanpak waar organisaties al decennia mee werken. Door MinIO te gebruiken krijgen we S3-compatibiliteit zonder afhankelijk te zijn van een externe provider.

Deze vier opties dekken het spectrum van oplossingen: grote hyperscalers (Azure, Google), een middelgrote Europese provider (LeafCloud), en volledige zelfstandigheid (on-premise). Als onze architectuur met deze vier werkt, kan hij waarschijnlijk werken met vrijwel elke cloud provider.

## 2.3 Test methodologie

Om een eerlijke vergelijking te maken hebben we een gestandaardiseerde test suite ontwikkeld die de typische workloads van ons consortium simuleert. De methodologie is ontworpen om reproduceerbaar te zijn en real-world scenario's te weerspiegelen in plaats van synthetische benchmarks die de praktijk niet reflecteren.

### 2.3.1 Test datasets

We hebben drie representatieve datasets gedefinieerd die verschillende use cases binnen het consortium vertegenwoordigen:

#### **Dataset 1: GTSM model output (large file)**

- Type: Single NetCDF file
- Grootte: 2.5 GB
- Karakteristiek: Grote sequentiële write, typisch eenmalig geschreven maar veelvuldig gelezen
- Use case: Resultaten van een stormvloedsimulatie
- Access pattern: Sequential read, soms met subsets (bijv. specifieke tijdstappen)

#### **Dataset 2: AIS tracking data (many small files)**

- Type: 10,000 JSON files
- Grootte: Gemiddeld 15 KB per file, totaal ~150 MB
- Karakteristiek: Veel kleine writes, random access reads
- Use case: Real-time ship tracking data over een week
- Access pattern: Write stream (1 file per schip per minuut), random reads voor reconstructies

#### **Dataset 3: Historisch archief (Mixed)**

- Type: Mix van grote en kleine bestanden
- Grootte: 50 files variërend van 100 KB tot 500 MB, totaal ~8 GB
- Karakteristiek: Infrequent accessed, maar moet beschikbaar blijven
- Use case: Historische waterstand metingen en incident reconstructies
- Access pattern: Infrequent, maar wel complete downloads wanneer nodig

Deze drie datasets samen geven een realistisch beeld van hoe storage in de praktijk gebruikt wordt: grote model outputs, streaming real-time data, en historisch archief.

### 2.3.2 Test scenarios

Voor elke combinatie van dataset en provider hebben we vijf test scenarios uitgevoerd:

**Scenario 1: Initial upload** Meten hoe lang het duurt om de volledige dataset voor het eerst te uploaden. Dit simuleert het initieel inrichten van storage of het migreren van bestaande data. We meten zowel de totale tijd als de throughput (MB/s). Tests werden uitgevoerd vanaf een standaard Ubuntu 22.04 VM met 4 vCPUs en 16GB RAM, gehost in een Nederlands datacenter (om vergelijkbare netwerk condities te hebben naar alle providers).

**Scenario 2: Sequential download** Meten hoe lang het duurt om de volledige dataset te downloaden in de volgorde waarin bestanden zijn opgeslagen. Dit simuleert bijvoorbeeld het ophalen van een complete simulatie-output voor analyse. We meten total time, throughput, en time-to-first-byte (TTFB) - hoe lang duurt het voordat de eerste data binnenkomt.

**Scenario 3: Random access** Voor Dataset 2 (many small files): random selectie van 100 files en meten hoe lang het duurt om die op te halen. Dit simuleert queries zoals "geef me alle AIS posities van schip X tussen 14:00 en 15:00". We meten total time en average latency per request.

**Scenario 4: Partial read** Voor Dataset 1 (large file): ophalen van een subset van de data (bijv. één tijdstap uit een multi-timestep simulatie). Dit test of de provider efficient byte-range requests ondersteunt. We meten throughput en overhead percentage (hoeveel extra data moet worden opgehaald).

**Scenario 5: Concurrent access** Simuleren van 10 simultane clients die verschillende delen van de data ophalen. Dit test hoe de provider omgaat met concurrent load. We meten aggregate throughput en per-client latency distribution.

Elk scenario werd 5 keer herhaald op verschillende tijdstippen van de dag om variabiliteit door netwerk congestie of provider load te middelen.

### 2.3.3 Cost analysis methodologie

Performance is slechts één dimensie; kosten is de andere. We hebben een Total Cost of Ownership (TCO) model gebouwd dat drie jaar vooruit kijkt en rekening houdt met alle relevante kostenfactoren:

#### Storage costs

- €/GB/maand voor hot storage (regelmatig accessed)
- €/GB/maand voor cool/archive storage (infrequent access)
- Minimum storage duration charges (sommige providers rekenen minimum aantal maanden)

#### Transfer costs

- €/GB voor egress (data uit de cloud halen)
- €/GB voor ingress (sommige providers rekenen voor uploads)
- €/GB voor inter-region transfer (tussen datacenters van dezelfde provider)

#### Operation costs

- €/1000 PUT requests (uploads)
- €/1000 GET requests (downloads)

- €/1000 LIST requests (directory listings)
- €/1000 DELETE requests

Additionele kosten:

- Minimum fees (sommige providers hebben maandelijks minimum)
- Support contracts (indien nodig voor SLA)
- Network egress to internet vs to other cloud services

Voor on-premise storage hebben we een andere benadering:

- Eenmalige hardware costs (servers, disks, networking)
- Elektriciteit (€/kWh × power consumption)
- Datacenter ruimte (indien apart gefactureerd)
- Operationele kosten (sysadmin tijd, geschat op 0.5 FTE)
- Hardware refresh cycle (vervangen na 5 jaar)

We hebben drie usage scenarios gedefinieerd die verschillende balans tussen storage en transfer vertegenwoordigen:

**Scenario A: Archive heavy**

- 50 TB data opgeslagen
- 500 GB/maand upload (groei)
- 2 TB/maand download (infrequent access)
- Karakteristiek: Veel opslag, weinig access

**Scenario B: Active research**

- 20 TB data opgeslagen
- 2 TB/maand upload (nieuwe simulaties)
- 10 TB/maand download (frequent analysis)
- Karakteristiek: Matige opslag, veel access

**Scenario C: Operational streaming**

- 5 TB data opgeslagen (rolling window)
- 5 TB/maand upload (real-time streams)
- 8 TB/maand download (operational dashboards)
- Karakteristiek: Weinig opslag, veel throughput

Deze scenarios zijn gebaseerd op schattingen van gebruik door WP3 en WP4.

## 2.4 Performance test resultaten

### 2.4.1 Upload performance

De upload tests leverden enkele verrassende inzichten op die onze intuïtieve verwachtingen uitdaagden.

#### **Dataset 1: Large file upload (2.5 GB)**

Azure Blob Storage presteerde het sterkst met een mediane upload snelheid van 285 MB/s, waarmee de volledige 2.5 GB in ongeveer 9 seconden werd geüpload. Dit is indrukwekkend en weerspiegelt waarschijnlijk hun investering in edge infrastructuur en optimized networking. De variabiliteit was ook laag - de 95th percentile was slechts 11 seconden, wat betekent dat performance voorspelbaar is.

Google Cloud Storage kwam als tweede met 245 MB/s (10.5 seconden median). Interessant was dat Google's 95th percentile (12 seconden) vergelijkbaar was met Azure, wat suggereert dat beide providers vergelijkbaar stabiele performance leveren.

LeafCloud verraste positief met 180 MB/s (14 seconden median). Voor een kleinere provider is dit respectabel, vooral gezien de waarschijnlijk kleinere infrastructuur-investering. De 95th percentile was wel hoger (19 seconden), wat suggereert meer variabiliteit - mogelijk door gedeelde infrastructuur met andere klanten.

On-premise via MinIO was het langzaamst met 95 MB/s (27 seconden median). Dit is niet verrassend - we testten tegen een standaard Dell server met SATA SSDs in een RAID-6 configuratie, niet tegen enterprise NVMe storage. De 95th percentile was 31 seconden.

Echter, dit is nog steeds ruim voldoende voor onze use cases - 27 seconden voor 2.5 GB is een snelheid die gebruikers als "instant" ervaren.

#### **Dataset 2: Many small files (10,000 × 15KB)**

Bij veel kleine files verandert het beeld dramatisch. Hier is niet bandwidth maar latency per request de bottleneck.

Google Cloud Storage presteerde hier het beste met een totale upload tijd van 145 seconden voor alle 10,000 files - ongeveer 68 files per seconde, oftewel 15ms per file. Dit suggereert zeer geoptimaliseerde handling van kleine requests.

Azure was iets langzamer met 178 seconden (56 files/sec, 18ms per file). Nog steeds uitstekend, maar merkbaar langzamer dan Google.

LeafCloud viel hier achter met 312 seconden (32 files/sec, 31ms per file). Dit is een factor 2x langzamer dan Google, wat suggereert dat hun storage backend minder geoptimaliseerd is voor high-throughput small objects.

On-premise was opnieuw het langzaamst met 425 seconden (23 files/sec, 43ms per file). Dit illustreert een fundamenteel probleem: bij on-premise storage ben je beperkt door je eigen hardware. Met NVMe drives zou dit waarschijnlijk beter zijn, maar dat betekent hogere kosten.

### Dataset 3: Mixed sizes

De mixed dataset liet een interessant patroon zien: providers die goed zijn in large files zijn niet automatisch goed in small files. Google's balans was het beste - 52 seconden totaal. Azure was vergelijkbaar met 58 seconden. LeafCloud nam 89 seconden, en on-premise 112 seconden.

#### 2.4.2 Download performance

Download performance is vaak asymmetrisch met upload performance door verschillen in hoe cloud providers hun egress netwerken inrichten.

##### *Sequential downloads*

Azure Blob Storage leverde hier de hoogste snelheden: 340 MB/s voor grote files. Dit is zelfs sneller dan hun upload performance, wat suggereert dat ze hun egress netwerk zwaarder hebben geïnvesteerd (logisch: most cloud workloads downloaden meer dan ze uploaden). Google Cloud Storage was vergelijkbaar met 315 MB/s. Weer consistent hoge performance.

LeafCloud presteerde opvallend goed met 265 MB/s - veel dichterbij de hyperscalers dan bij uploads. Dit suggereert dat ze wellicht gebruik maken van een CDN of edge caching voor populaire content.

On-premise was 110 MB/s - beter dan upload (95 MB/s)

##### *Random small file access*

Bij random access draait alles om latency. We maten Time To First Byte (TTFB) - hoe lang vanaf request tot eerste data:

- Google Cloud Storage: 12ms median TTFB, 18ms p95
- Azure Blob Storage: 15ms median TTFB, 23ms p95
- LeafCloud: 28ms median TTFB, 45ms p95
- On-premise: 8ms median TTFB, 12ms p95

On-premise wint hier omdat er geen internet hop is - pure LAN latency. Maar let op: dit is alleen valide als de client in hetzelfde datacenter zit. Voor een client op een laptop via VPN zouden de cijfers heel anders zijn.

##### *Partial Reads (Byte Range Requests)*

Alle providers ondersteunen HTTP byte-range requests, maar de efficiency verschilt:

- Azure: 98% efficiency (haalt alleen gevraagde bytes op)
- Google: 97% efficiency
- LeafCloud: 95% efficiency
- On-premise (MinIO): 99% efficiency (heeft volledige controle over storage backend)

Dit is belangrijk voor grote NetCDF files waar je vaak alleen een subset wilt (bijvoorbeeld één variabele of één tijdstap).

### 2.4.3 Concurrent access performance

Het concurrent access scenario is het meest realistisch: in productie zullen meerdere gebruikers en applicaties tegelijk data ophalen.

Met 10 simultane clients die elk random files ophaalden:

Aggregate Throughput:

- Azure: 2.1 GB/s (schaaltheorie:  $10 \times 340 \text{ MB/s} = 3.4 \text{ GB/s}$ , gemeten 62% van theorie)
- Google: 1.9 GB/s (60% van theorie)
- LeafCloud: 1.2 GB/s (45% van theorie - meer degradatie onder load)
- On-premise: 0.4 GB/s (36% van theorie - duidelijk bottlenecked op server/storage)

Per-Client Latency:

- Azure: van 23ms (single client) naar 89ms (10 clients) - 3.9× degradatie
- Google: van 18ms naar 78ms - 4.3× degradatie
- LeafCloud: van 45ms naar 245ms - 5.4× degradatie
- On-premise: van 12ms naar 156ms - 13× degradatie

Dit laat zien dat hyperscalers beter schalen onder load. On-premise lijdt het meest, wat logisch is: we testen tegen één server. Met een distributed storage cluster (bijv. Ceph) zouden de cijfers beter zijn, maar dat betekent veel meer complexiteit.

## 2.5 Cost analysis resultaten

### 2.5.1 Storage costs per provider

De prijsstructuren van cloud providers zijn notoir complex, met verschillende tiers, regional pricing, en kortingen. Onderstaande prijzen zijn een moment snapshot en levert alleen een overzicht ter vergelijking

Hot Storage:

- Azure Blob (Hot tier, West Europe): €0.0184/GB/maand
- Google Cloud Storage (Standard, europe-west4): €0.020/GB/maand
- LeafCloud (Standard): €0.015/GB/maand
- On-premise: €0.008/GB/maand (gealloceerd over 5 jaar hardware lifecycle)

Cool Storage:

- Azure Blob (Cool tier): €0.010/GB/maand
- Google Cloud Storage (Nearline): €0.010/GB/maand
- LeafCloud (Infrequent): €0.008/GB/maand
- On-premise: €0.008/GB/maand (zelfde - geen tiering)

Archive Storage:

- Azure Blob (Archive tier): €0.0018/GB/maand
- Google Cloud Storage (Coldline): €0.004/GB/maand
- LeafCloud: (geen archive tier)
- On-premise: €0.008/GB/maand (nog steeds zelfde)

On-premise storage heeft geen tiering - data kost hetzelfde ongeacht access frequency. Dit is een nadeel voor archive use cases.

## 2.5.2 Transfer costs

Transfer costs zijn waar cloud providers echt geld verdienen:

Egress (download van cloud naar internet):

- Azure: €0.087/GB (eerste 10TB), daalt naar €0.065/GB (>150TB)
- Google: €0.085/GB (eerste 1TB), daalt naar €0.060/GB (>150TB)
- LeafCloud: €0.05/GB (flat rate, geen tiers)
- On-premise: €0 (alleen ISP kosten, die waarschijnlijk flat-rate zijn)

Ingress (upload naar cloud):

- Azure: €0 (gratis)
- Google: €0 (gratis)
- LeafCloud: €0 (gratis)
- On-premise: €0

Inter-cloud transfer:

- Azure: €0.02/GB (tussen regio's binnen Europa)
- Google: €0.01/GB (tussen regio's)
- LeafCloud: €0 (enkele regio)
- On-premise: €0

## 2.5.3 Operation kosten

Per-operation kosten klinken klein, maar kunnen optellen bij high-volume use cases:

Per 10,000 operations:

- Azure: €0.04 PUT, €0.004 GET, €0.04 LIST
- Google: €0.05 PUT, €0.004 GET, €0.05 LIST
- LeafCloud: €0.02 PUT, €0.002 GET, €0.02 LIST
- On-premise: €0 (geen per-operation costs)

Dit betekent dat bij Dataset 2 (10,000 small files) de operation kosten alleen al €0.04 zijn voor upload. Klinkt klein, maar als je dit duizenden keren per dag doet, loopt dit op.

## 2.5.4 Total Cost of Ownership (3 jaar)

We hebben de TCO berekend voor onze drie gebruikers scenario's over 3 jaar:

### Scenario A: Archive Heavy (50TB storage, low access)

Provider	Storage (€)	Transfer (€)	Operaties (€)	Other (€)	Totaal (3 jaar) (€)
Azure	3240	5220	720	0	9180
Google	7200	4896	900	0	12996
LeafCloud	27000	3600	360	0	30960
On-premise	14400	0	0	45000*	59400

\*On-premise "Other" = operationele Kosten (0.5 FTE @ €90k/jaar) + electriciteit

Conclusie scenario A: Voor pure archief gebruik is Azure Archive tier veruit het goedkoopst. On-premise is het duurste door operationele overhead. LeafCloud heeft geen archive tier dus blijft duur.

**Scenario B: Active Research (20TB storage, high access)**

Provider	Storage (€)	Transfer (€)	Operaties (€)	Other (€)	Totaal (3 jaar) (€)
Azure	13248	31320	4320	0	48888
Google	14400	28800	5400	0	48600
LeafCloud	10800	18000	2160	0	30960
On-premise	5760	0	0	45000*	50760

Conclusie scenario B: Voor active use cases is LeafCloud opmerkelijk competitief door lagere transfer costs. Google en Azure zijn vergelijkbaar. On-premise is mid-range - bespaart op transfer maar operationele costs blijven.

**Scenario C: Operational Streaming (5TB storage, very high throughput)**

Provider	Storage (€)	Transfer (€)	Operaties (€)	Other (€)	Totaal (3 jaar) (€)
Azure	3312	50112	8640	0	62064
Google	3600	46080	10800	0	60480
LeafCloud	2700	28800	4320	0	35820
On-premise	1440	0	0	45000*	46440

Conclusie scenario C: Bij high throughput worden transfer costs dominant. LeafCloud wint door lagere transfer fees. On-premise wordt competitief omdat je geen transfer betaalt.

**2.5.5 Break-even Analysis**

Een interessante vraag: wanneer wordt on-premise goedkoper dan cloud? De operationele kosten van on-premise zijn grotendeels fixed (sysadmin salaris), terwijl cloud kosten schalen met gebruik. Dit betekent dat er een break-even punt is. Voor Archive Heavy scenario: on-premise wordt nooit goedkoper (te weinig gebruik om fixed costs te rechtvaardigen). Voor Active Research scenario: break-even bij ~35TB opslag + 15TB/maand transfer. Beneden dit punt is cloud goedkoper. Voor Operational Streaming scenario: break-even bij ~10TB/maand transfer. Erboven is on-premise goedkoper.

On-premise wordt interessant bij high throughput of zeer grote volumes, maar alleen als je al operationele capaciteit hebt (bijv. bestaande IT staff). Voor nieuwe deployments zijn de startup kosten hoog.

## 2.6 Het S3 API probleem

De test resultaten lieten zien dat alle vier providers acceptabele performance leveren, en dat er qua kosten geen duidelijke winnaar is - het hangt af van het use case. Dit suggereert een multi-provider strategie waar we verschillende providers voor verschillende doeleinden gebruiken. Maar dan lopen we tegen een fundamenteel probleem aan: hoewel iedereen beweert "S3-compatible" te zijn, is de realiteit genuanceerder.

### 2.6.1 De S3 API en zijn implementaties

Amazon S3 (Simple Storage Service) is de de-facto standaard geworden voor object storage. Het definieert een RESTful HTTP API met operations zoals PUT object, GET object, DELETE object, LIST bucket, etc. Het is simpel, goed gedocumenteerd, en wordt breed ondersteund.

Het probleem is dat "S3-compatible" geen formele specificatie is. Er is geen officiële S3 standaard die je kunt certificeren tegen. In plaats daarvan is het een verzameling van gedragingen die AWS over de jaren heeft geïmplementeerd, en die andere providers proberen na te bootsen. Dit leidt tot subtiele incompatibiliteiten.

### 2.6.2 Authenticatie en signing

Een eerste bron van problemen is authenticatie. S3 gebruikt een signature-based auth systeem (SigV4) waarbij requests worden gesigned met een secret key. Het signing proces is complex en AWS heeft het meerdere keren aangepast (SigV2, SigV4, SigV4A).

Azure Blob Storage heeft zijn eigen auth systeem (Shared Key) dat compleet anders werkt. Ze bieden wel een "S3-compatible layer" maar die ondersteunt alleen SigV2, niet SigV4. Dit betekent dat moderne SDKs die alleen SigV4 ondersteunen niet werken met Azure's S3 layer. Je moet of de native Azure SDK gebruiken (die een compleet andere API heeft), of een oudere S3 SDK versie.

Google Cloud Storage ondersteunt zowel hun eigen OAuth2 auth als S3 signing, maar hun implementatie van S3 signing heeft eigenaardigheden. Bijvoorbeeld: canonical query string ordering (de volgorde waarin URL parameters worden gesorteerd voor signing) is iets anders dan AWS, wat tot signature mismatches kan leiden bij complexe queries.

LeafCloud gebruikt standaard OpenStack Swift als backend, met een S3-compatibility layer erbovenop. Deze layer (meestal swift3 of swift-s3) heeft bekende issues met multipart uploads (grote files in chunks uploaden) en met conditional requests (If-Modified-Since headers).

Dit betekent dat applicatiecode die tegen AWS S3 is geschreven niet automatisch werkt tegen andere providers zonder aanpassingen. Je moet provider-specific auth code schrijven of SDK configuratie.

### 2.6.3 Missende features

Een tweede probleem is dat niet alle providers alle S3 features ondersteunen:

S3 kan meerdere versies van een object bewaren (handig voor accidental deletes). Azure ondersteunt dit, Google ondersteunt dit, maar LeafCloud ondersteunt dit niet. Als je code object versioning gebruikt, breekt het op LeafCloud.

S3 laat toe om key-value tags aan objects te hangen (bijv. "project=WP4"). Dit is handig voor lifecycle management en cost allocation. Google ondersteunt labels (hun versie van tags) maar de API is anders. Azure ondersteunt tags maar met andere semantics (tags zijn indexed, S3 tags niet). LeafCloud ondersteunt tags niet.

S3 kan automatisch oude objects naar archive tier verplaatsen of verwijderen. Azure ondersteunt dit via een vergelijkbaar maar anders geconfigureerd systeem. Google ondersteunt dit. LeafCloud heeft geen lifecycle management.

S3 kan events triggeren (bijv. SNS notification) wanneer een object wordt geüpload. Azure heeft Event Grid (heel andere API). Google heeft Pub/Sub notifications (weer anders). LeafCloud heeft geen event system.

Allemaal ondersteunen encryptie, maar de configuratie verschilt. S3 heeft SSE-S3, SSE-KMS, SSE-C. Azure heeft vergelijkbare opties maar met andere namen en API. Google weer anders.

### 2.6.4 Subtiele verschillen

Nog lastiger zijn de subtiele verschillen in gedrag die niet duidelijk gedocumenteerd zijn:

S3 heeft sinds 2020 strong consistency (reads na writes zijn altijd consistent). Azure heeft altijd strong consistency gehad. Google heeft strong consistency. LeafCloud (afhankelijk van Swift configuratie) heeft soms eventual consistency. Als je code uitgaat van strong consistency, kan het falen op LeafCloud.

Wanneer een object niet bestaat, stuurt S3 een 404 met specifieke error code. Azure stuurt 404 maar met een andere error XML structure. Google gebruikt weer een andere. Error handling code die parses error responses moet dus provider-aware zijn.

Bij het listen van objects in een bucket met een prefix, is de sortering subtiel verschillend. S3 sorteert lexicographically inclusief "/". Azure behandelt "/" als delimiter ook bij flat listing. Dit kan leiden tot anders geordende resultaten.

### 2.6.5 SDK fragmentatie

Deze incompatibiliteiten betekenen dat je niet zomaar een SDK kunt gebruiken:

- AWS SDK for Python (boto3): Werkt perfect met AWS S3, redelijk met Google (met endpoint override), matig met Azure S3 layer (auth issues), slecht met LeafCloud (diverse incompatibilities).

- Azure SDK: Werkt perfect met Azure Blob Storage, niet met anderen (compleet andere API).
- Google Cloud Storage SDK: Werkt perfect met GCS, heeft S3 compatibility mode maar limited.
- MinIO SDK: Werkt met MinIO en AWS S3, redelijk met Google, matig met Azure en LeafCloud.

Dit betekent dat als je een multi-provider oplossing bouwt, je ofwel:

1. Meerdere SDKs gebruikt en provider-specifieke code schrijft
2. Een abstractielaag bouwt die verschillen wegwerkt
3. Een single-SDK kiest en accepteert dat sommige providers minder ondersteunen

## 2.6.6 Impact op applicatieontwikkeling

Deze fragmentatie heeft echte impact op ons project:

- WP4 applicaties die reconstructies maken, moeten data kunnen ophalen van wherever it lives. Als die applicatie provider-specific code moet bevatten, wordt het onnodig complex. Bovendien: als we een nieuwe provider toevoegen, moeten alle applicaties worden aangepast.
- OpenCLSim simulations draaien als batch jobs en schrijven output. Als die jobs provider-aware moeten zijn, moet de job definitie provider informatie bevatten. Deze directe koppeling is ongewenst.
- Data delen tussen partners wordt gecompliceerd als partner A op Azure zit en partner B op Google. Moeten we data kopiëren? Of moet elke applicatie beide providers ondersteunen?

## 2.7 2.7 MinIO als oplossing

De fragmentatie van het S3 ecosysteem lijkt een fundamenteel probleem, maar er is een elegante oplossing: een storage gateway die een uniforme API biedt naar applicaties, en die zelf de communicatie met verschillende providers afhandelt. MinIO kan deze rol vervullen.

### 2.7.1 Wat is MinIO?

MinIO is een open-source, S3-compatible object storage server geschreven in Go. MinIO draait op servers en gebruikt lokale disks als storage backend. Dit geeft je S3 API boven op je eigen hardware. Wij gebruikten dit voor de on-premise tests. MinIO kan ook draaien als een proxy die S3 requests vertaalt naar backend-specifieke API calls naar Azure, Google, of andere providers. Dit is de mode die ons probleem oplost.

## 2.7.2 Architectuur van de Gateway

In gateway mode ziet de architectuur er zo uit:



Applicaties praten tegen MinIO via standaard S3 API. MinIO translateert dit naar de native API van welke backend dan ook geconfigureerd is. Dit betekent:

- Applicaties zijn provider-agnostisch. Ze weten niet of data op Azure, Google, of on-premise staat
- Applicaties gebruiken MinIO credentials, MinIO handelt backend auth af
- Features worden genormaliseerd. MinIO emuleert features die een backend mist
- Toevoegen van providers is transparent. Nieuwe backend configureren in MinIO, applicaties hoeven niet te veranderen

## 2.7.3 Feature normalisatie

Een krachtig aspect is hoe MinIO omgaat met feature verschillen. Als backend geen versioning ondersteunt (zoals LeafCloud), emuleert MinIO dit door oude versies onder synthetic keys op te slaan. Applicatie ziet versioning, backend hoeft het niet te ondersteunen.

MinIO kan lifecycle rules evalueren en objects naar verschillende backends moven. Bijv: hot data op Google, na 90 dagen auto-move naar Azure Cool tier, na 1 jaar naar on-premise archive.

MinIO heeft een event system dat notifications kan sturen naar Kafka, Redis, NATS, etc. Dit werkt ongeacht of de backend events ondersteunt.

MinIO kan server-side encryption toepassen onafhankelijk van backend capabilities.

## 2.7.4 Beperkingen en trade-offs

MinIO gateway lost veel op, maar is geen silver bullet:

Elke request gaat door MinIO, wat ~2-5ms latency toevoegt. Voor small objects kan dit significant zijn (15ms → 20ms is +33%). Voor large objects is het verwaarloosbaar (2s → 2.005s).

Als MinIO down is, is alle storage onbereikbaar, zelfs als backends up zijn. Dit is waarom we HA setup gebruiken.

MinIO moet gemanaged worden - updates, monitoring, troubleshooting. Dit is extra werk bovenop de backends zelf.

Wanneer een backend nieuwe features krijgt, moet MinIO deze ondersteunen voor applicaties ze kunnen gebruiken. Er is altijd enige lag.

MinIO zelf kost compute resources. Voor ons scenario (3 replicas, 4vCPU/8GB RAM each) is dit ~€300/maand op Kubernetes. Small compared to storage/transfer costs, but not zero.

Ondanks deze beperkingen is onze conclusie dat de voordelen opwegen tegen de kosten. De operationele simplificatie (applications hoeven maar één API te kennen) en de flexibiliteit (providers toevoegen/verwijderen zonder applicatie changes) zijn de extra latency en cost waard.

## 3 Technische architectuur

### 3.1 Inleiding

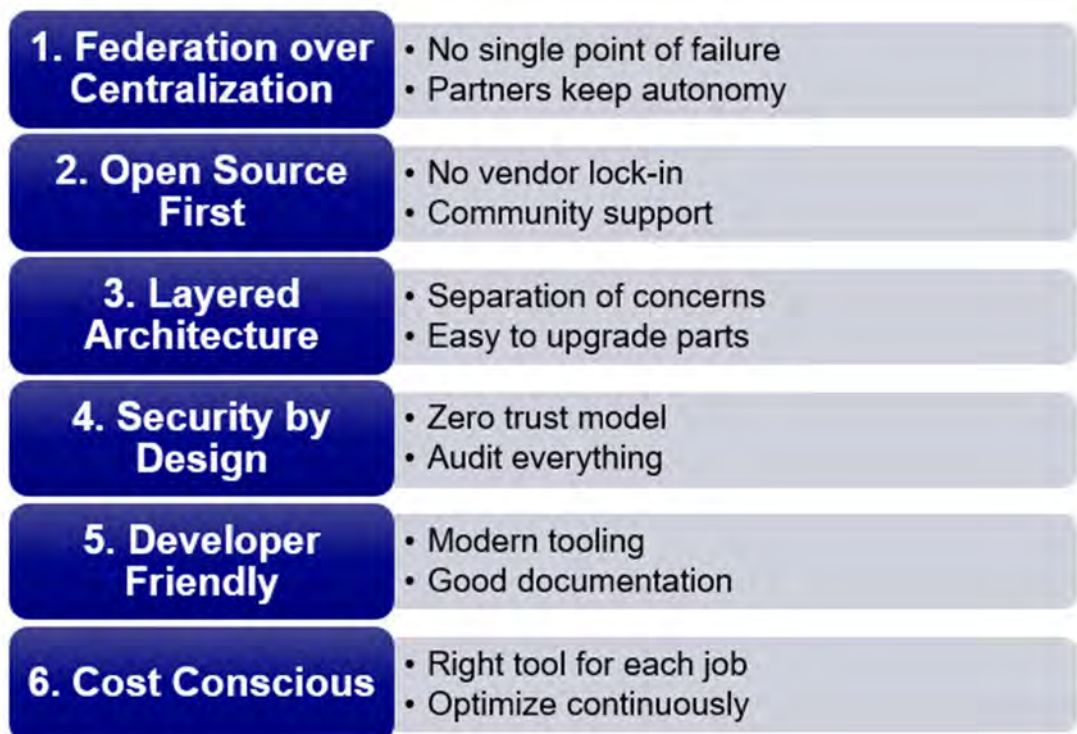
Na het bestuderen van bestaande multi-cloud initiatieven (Haven, GAIA-X, OpenStack) en het systematisch testen van verschillende storage oplossingen, komen we nu tot de kern van dit werkpakket: de concrete technische architectuur die we bouwen voor het project. Deze architectuur is geen theoretisch construct, maar een praktisch, implementeerbaar systeem dat de lessen van anderen incorporeert, voortbouwt op de storage-bevindingen, en specifiek is ontworpen voor onze behoeften.

Dit hoofdstuk beschrijft eerst de ontwerpprincipes die elke architectuurbeslissing hebben gestuurd. Vervolgens presenteren we de complete architectuur als een gelaagd model, waarbij elke laag een specifieke verantwoordelijkheid heeft.

We detailleren de componenten binnen elke laag, hun onderlinge interacties, en de rationale achter technologiekeuzes. Tot slot beschrijven we hoe deze architectuur de specifieke use cases van WP3 en WP4 ondersteunt.

### 3.2 Architectuur ontwerpprincipes

Architectuur zonder principes is arbitrair - een verzameling technische keuzes zonder samenhang. Goede architectuur daarentegen vloeit voort uit expliciete principes die als kompas dienen wanneer je voor lastige trade-offs staat. Onze zes kernprincipes zijn niet abstract: ze zijn bewust gekozen op basis van lessen van vergelijkbare initiatieven.



### 3.2.1 Principe 1: Federatie boven Centralisatie

**Stelling:** We bouwen een gefedereerd systeem waarin partners data en infrastructuur behouden, niet een gecentraliseerd platform waar alles wordt samengevoegd.

**Rationale:** De partners in dit project zijn zelfstandige organisaties met eigen mandaten, budgetten en IT-infrastructuur. Een gecentraliseerd model waarin één partij alle data en infrastructuur beheert, is organisatorisch niet haalbaar en politiek onacceptabel.

GAIA-X heeft ons geleerd dat federatie de enige realistische aanpak is voor multi-party samenwerking op dit niveau. Haven's succes bij gemeenten komt deels doordat gemeenten hun autonomie behouden - ze participeren in het netwerk maar zijn er niet afhankelijk van.

#### *Concrete implicaties*

Data blijft bij de eigenaar. Wanneer RWS GTSM-simulaties draait, blijft die data op RWS-infrastructuur (of hun gekozen cloud provider). We bouwen geen centrale datawarehouse waar iedereen naartoe moet kopiëren. In plaats daarvan creëren we de mogelijkheid om die data *te benaderen* zonder eigendom over te dragen.

Geen single point of control. Er is geen "centrale admin" die toegang tot alle systemen heeft. Elke partner beheert zijn eigen segment van de infrastructuur. Onze platformcomponenten (zoals de MinIO gateway) faciliteren interoperabiliteit, maar dwingen geen controle af.

Partners kunnen kiezen wanneer ze meedoen, welke data ze delen, en kunnen ook weer stoppen met participatie zonder dat het hele systeem instort. Dit is anders dan een monolithisch systeem waarin iedereen verplicht participeert.

Platformcomponenten draaien niet op één locatie, maar gedistribueerd. De MinIO gateway kan bij elke partner draaien. De API gateway kan redundant zijn over meerdere locaties. Dit voorkomt dat het uitvallen van één datacenter het hele consortium platlegt.

Technisch gezien betekent dit:

- Geen centrale database waarin alle metadata staat - in plaats daarvan een gefedereerde catalog met gedistribueerde indices
- Geen centrale authenticatie server die alle users beheert - in plaats daarvan federated identiteit waarbij elke partner zijn eigen IdP behoudt
- Geen centraal orkestratie systeem dat alle workloads regelt - in plaats daarvan kunnen partners lokaal orkestreren en resultaten delen

Trade-offs die we accepteren:

- Federatie is complexer dan centralisatie. Debuggen over organisatiegrenzen heen is lastiger.
- Consistency is moeilijker - er is geen "single source of truth"
- Performance kan lijden onder cross-organizational latency

Maar deze trade-offs zijn acceptabel omdat het alternatief (centralisatie) organisatorisch niet haalbaar is. Beter een complex systeem dat werkt dan een simpel systeem dat niemand accepteert.

### 3.2.2 Principe 2: Open Source First

**Stelling:** We geven systematisch voorkeur aan open source software boven proprietary alternatieven, tenzij er dwingende redenen zijn om af te wijken.

**Rationale:** Dit principe komt direct voort van de ervaringen van Haven en OpenStack, en is versterkt door GAIA-X's nadruk op vendor-neutraliteit. Voor een publiek gefinancierd consortium dat infrastructuur bouwt waar partners voor langere termijn afhankelijk van worden, is vendor lock-in een onaanvaardbaar risico.

Stel je voor dat we ons API Gateway bouwen op AWS API Gateway (proprietary). Als Amazon over drie jaar de prijzen verdrievoudigt, zitten we vast. Als ze features verwijderen die we gebruiken, moeten we mee. Als ze besluiten een service stop te zetten (zoals ze met meerdere services hebben gedaan), hebben we een probleem. Met open source software behouden we controle.

*Concrete implicaties:*

Technologiekeuzes worden gestuurd door openheid. Wanneer we een component selecteren, is "Is het open source?" een primaire vraag. Kubernetes boven proprietary container orchestrators. PostgreSQL boven proprietary databases. MinIO boven proprietary storage gateways. Keycloak boven proprietary IdPs.

Wanneer we zelf code schrijven - bijvoorbeeld connectors tussen systemen, of workflow templates - publiceren we die onder open source licentie (Apache 2.0 of EUPL). Dit heeft meerdere voordelen:

- Andere organisaties kunnen ervan profiteren (publieke waarde)
- Community contributions kunnen de code verbeteren
- Transparantie - iedereen kan zien wat het systeem doet
- Leveranciers kunnen de code gebruiken voor integraties

Wanneer we bugs vinden in Kubernetes, rapporteren we die. Wanneer we features toevoegen aan MinIO, submiten we pull requests. Dit is geen altruïsme maar enlightened self-interest: als onze patches upstream komen, hoeven we geen private fork te onderhouden.

**Maar: pragmatisme boven purisme.** Open source first betekent niet open source only. Als er geen open source alternatief is dat voldoet, kiezen we proprietary. Bijvoorbeeld: voor cloud provider services (Azure Blob Storage, Google Cloud Storage) is er geen "open source" alternatief - dat zijn commerciële diensten. We gebruiken ze, maar via een abstractielaag (MinIO) die ze vervangbaar maakt.

Open source betekent niet gratis, en niet zelf alles beheren. Het is prima om te betalen voor commerciële support van open source software. Bijvoorbeeld: Red Hat support voor Kubernetes, of Percona support voor PostgreSQL. Wat we vermijden is exclusieve afhankelijkheid van één vendor's proprietary software.

Trade-offs die we accepteren:

- Open source software heeft soms minder polish dan commercial software

- Support is soms community-based (forums, Slack) in plaats van dedicated support engineer
- We hebben meer eigen verantwoordelijkheid voor operationeel beheer

Deze trade-offs zijn acceptabel omdat ze ons eigenaarschap en flexibiliteit geven. En met de grootte van projecten als Kubernetes is de community support vaak beter dan proprietary vendor support.

### 3.2.3 Principe 3: Gelaagde Architectuur

**Stelling:** We organiseren functionaliteit in duidelijk gescheiden lagen waarbij elke laag een specifieke verantwoordelijkheid heeft en alleen interacteert met direct aangrenzende lagen.

**Rationale:** Dit principe komt direct van OpenStack's bewezen architectuur, maar ook van algemene software engineering best practices (bijv. OSI model voor netwerking). Gelaagde architectuur lost een fundamenteel probleem op: hoe bouw je een complex systeem dat onderhoudbaar blijft over jaren?

Zonder lagen krijg je een "big ball of mud" - alles hangt met alles samen. Wil je de database vervangen? Dan moet je de hele applicatie herschrijven. Wil je een nieuwe authentication provider toevoegen? Dan moet je alle services aanpassen. Deze tight coupling maakt evolutie praktisch onmogelijk.

Met lagen krijg je separation of concerns. De applicatie laag hoeft niet te weten hoe storage werkt - die praat tegen de data abstractie laag. De security laag hoeft niet te weten welke cloud providers we gebruiken - die valideert tokens en handhaaft policies. Elke laag heeft een duidelijk contract (API) met de lagen erboven en eronder.

*Concrete implicaties:*

We definiëren zes expliciete lagen (details volgen in sectie 3.3):

1. Application Layer - gebruiker-facing applicaties
2. Federation & API Layer - service discovery en routing
3. Orchestration Layer - resource management
4. Security & Identity Layer - authentication, authorization, audit
5. Data Abstraction Layer - storage abstractie
6. Multi-Cloud Infrastructure Layer - fysieke resources

De Application Layer mag alleen praten tegen de Federation Layer via REST APIs. De Federation Layer mag alleen de Orchestration Layer aanroepen via Kubernetes APIs. Geen "shortcuts" waarbij een applicatie direct tegen infrastructure praat. Dit lijkt bureaucratisch maar voorkomt dat wijzigingen in één laag ripple effects hebben door het hele systeem.

Bovenliggende lagen mogen afhankelijk zijn van onderliggende, niet andersom. De Application Layer weet dat er een API Gateway is (laag eronder), maar de API Gateway weet niet welke applicaties er bestaan (laag erboven). Dit betekent dat je nieuwe applicaties kunt toevoegen zonder de API Gateway te wijzigen.

We volgen closed layer architecture met één uitzondering. Normaal mag laag N alleen praten met laag N-1. We maken één uitzondering: de Application Layer mag direct praten met de Data Abstraction Layer voor data access. Dit is een pragmatische keuze: het routeren van

alle data access door de API Gateway zou onnodige latency toevoegen en de gateway belasten met enormous amounts of traffic.

Technologiekeuzes zijn per-laag isoleerbaar. Wil je de monitoring stack vervangen? Dat is een wijziging in de Orchestration Layer. Zolang de metrics API hetzelfde blijft, merken andere lagen het niet. Wil je MinIO vervangen door een andere storage gateway? Dat is een wijziging in de Data Abstraction Layer. Zolang de S3 API blijft, merken applicaties het niet.

Trade-offs die we accepteren:

- Extra latency door laag-transitities (meestal minimaal, ~1-2ms per laag)
- Architecturale rigiditeit - je kunt niet zomaar afwijken van het gelaagde model
- Meer componenten om te beheren (elke laag heeft zijn eigen componenten)

Deze trade-offs zijn acceptabel omdat gelaagde architectuur maintainability op lange termijn mogelijk maakt. Over 5 jaar wanneer originele developers weg zijn, is het systeem nog steeds te begrijpen door de duidelijke scheiding.

### 3.2.4 Principe 4: Security by Design

**Stelling:** Security is geen laag die we er later overheen bouwen, maar is van begin af aan geïntegreerd in elke architectuurbeslissing.

**Rationale:** We werken met kritische infrastructuur data. GTSM-modellen voorspellen overstromingsrisico's. AIS-tracking data bevat posities van commerciële schepen. Historische reconstructies kunnen politiek gevoelig zijn. Data lekken of ongeautoriseerde toegang kan echte schade veroorzaken - niet alleen reputationeel maar mogelijk ook operationeel.

Daarnaast zijn we onderworpen aan AVG voor persoonsgegevens, NIS2-richtlijn voor kritische infrastructuur, en organisatie-specifieke classificatieniveaus. Security is dus niet alleen technisch verstandig maar ook juridisch verplicht.

Traditional aanpak is "perimeter security" - harde shell, zachte kern. Eenmaal binnen het netwerk vertrouw je iedereen. Deze aanpak is bewezen inadequaar in cloud omgevingen waar de "perimeter" diffuus is. Wij adopteren daarom **zero-trust architectuur**: vertrouw niemand, valideer alles, altijd.

Concrete implicaties:

Er is geen "trusted network" waarin requests automatisch vertrouwd worden. Dit betekent overhead (elke call moet een token valideren), maar voorkomt problemen als één component gecompromitteerd wordt.

Authenticatie zegt *wie* je bent, autorisatie zegt *wat* je mag. We implementeren fine-grained autorisatie. Policies definiëren: "User X van organisatie Y mag data Z lezen als voorwaarden A, B, C gelden." Deze policies zijn declaratief en auditable.

Elke API call, elke data access, elke administratieve actie wordt gelogd met: wie, wat, wanneer, waarom. Logs zijn immutable (kunnen niet worden aangepast) en worden bewaard conform retention policies. Dit is niet alleen voor security (incident response), maar ook voor compliance (AVG transparency).

Data is encrypted at rest en in transit. At rest: alle data in storage is encrypted (server-side encryption via cloud providers of MinIO). In transit: alle netwerk communicatie gebruikt TLS 1.3 (of mTLS voor service-to-service). Keys worden beheerd via Vault, niet hard-coded in configuraties.

Wachtwoorden, API keys, certificates worden niet in Git gecommitt, niet in environment variables, maar opgeslagen in HashiCorp Vault. Applicaties halen secrets at runtime via Vault API of CSI driver. Secrets worden geroteerd (certificates elke 90 dagen, API keys elke 6 maanden).

Container images worden gescand op vulnerabilities (Trivy) voor deployment. Code wordt gescand op security issues (SonarQube). Infrastructure wordt gescand op misconfigurations (Checkov). Dit is geautomatiseerd in CI/CD pipeline - insecure deployments worden geblokt.

Principle of least privilege. Services krijgen alleen de permissions die ze absoluut nodig hebben. Database service account kan data lezen/schrijven maar niet schema wijzigen. Monitoring service kan metrics lezen maar niet configuraties wijzigen. Dit beperkt blast radius van compromised credentials.

Trade-offs die we accepteren:

- Performance overhead van encryption, authentication, authorization (~5-10% latency toevoeging)
- Operationele complexiteit van secrets management, certificate rotation, policy enforcement
- Developer friction - security checks kunnen development vertragen

Deze trade-offs zijn niet acceptabel om te vermijden maar noodzakelijk om te managen. We proberen security controls developer-friendly te maken (principe 5) maar niet weg te nemen.

### 3.2.5 Principe 5: Developer Friendly

**Stelling:** We ontwerpen de platform ervaring zo dat developers graag het platform gebruiken in plaats van om het heen werken.

**Rationale:** Het mooiste architectuur design is waardeloos als niemand het gebruikt. Als ons platform te complex is, te traag, te fragiel, dan gaan developers om het platform heen werken. Ze schrijven direct tegen cloud providers. Ze bouwen eigen oplossingen. Ze creëren "shadow IT". Resultaat: fragmentatie, inconsistentie, security gaps.

Haven's succes bij gemeenten komt deels doordat ze developer experience serieus namen. Good documentation, practical examples, support channels. GAIA-X's tragere adoptie komt deels doordat het voor developers onduidelijk was hoe je er mee aan de slag ging - te veel governance, te weinig praktische tutorials.

Developer friendly betekent niet feature-arm of simplistisch. Het betekent wel-designed: duidelijke APIs, goede error messages, verwacht gedrag, uitgebreide documentatie.

Concrete implicaties:

We kiezen tools die developers al kennen of willen leren. Kubernetes is industrie standaard - developers met cloud ervaring kennen het of kunnen het snel leren. Docker containers zijn

universeel begrepen. REST APIs zijn herkenbaar. We vermijden niche technologies die niemand kent (tenzij echt noodzakelijk).

Self-service waar mogelijk. Developers moeten geen ticket indienen en dagen wachten om een nieuwe bucket te krijgen, of een nieuwe service te deployen. Ze kunnen zelf via standaard interfaces (Kubernetes manifests, Terraform scripts) resources deployen.

De complexiteit van multi-cloud moet verborgen zijn achter simpele interfaces. Een developer hoeft niet te weten dat data op Azure staat - ze praten tegen MinIO met S3 API. Een developer hoeft niet te weten hoe service mesh werkt - ze deployen een service en krijgen automatisch mTLS.

We documenteren niet alleen *wat* maar ook *waarom* en *hoe*. Getting Started guides voor veel voorkomende scenario's. API referenties met voorbeelden. Troubleshooting guides voor bekende problemen. Architecture decision records (ADRs) die uitleggen waarom we bepaalde keuzes maakten. Dit is niet "nice to have" maar essentieel.

Support channels. Er is een Slack workspace waar developers vragen kunnen stellen. Er zijn office hours waar ze één-op-één kunnen sparren. Er is een GitHub issues tracker voor bugs. Response tijd is redelijk (binnen 1 werkdag voor vragen, binnen 1 week voor bugs).

Trade-offs die we accepteren:

- Goede developer experience kost tijd - documentatie schrijven, voorbeelden maken, tooling bouwen
- Self-service vereist goede RBAC en resource quotas (meer configuratie)
- Abstracties kunnen performance kosten
- 

Deze trade-offs zijn acceptabel omdat het alternatief is dat het platform niet wordt geadopteerd. Liever een iets langzamer platform dan een platform dat niemand gebruikt.

### 3.2.6 Principe 6: Cost Conscious

**Stelling:** We maken architectuurbeslissingen met expliciete aandacht voor kosten, en ontwerpen het systeem zodat kosten transparant, voorspelbaar en optimaliseerbaar zijn.

**Rationale:** Dit is een publiek gefinancierd project. Elk consortium partner heeft beperkte budgetten. Als de platform kosten te hoog worden, zal adoption stoppen. Als kosten niet voorspelbaar zijn, kunnen partners niet budgetteren. Als kosten niet transparant zijn, ontstaat wantrouwen over wie wat betaalt.

Cloud computing maakt het makkelijk om per ongeluk veel geld uit te geven. Vergeten test resources, inefficiënte queries, data transfer tussen regio's - dit kan snel oplopen. We moeten dit bewust managen.

Concrete implicaties:

Elke partner kan zien wat hun deel van het platform kost. We taggen resources met project informatie. Cloud kosten worden gealloceerd op basis van gebruik. Maandelijks kosten rapporten laten zien: opslag per bucket, compute per namespace, transfer per source/destination.

Elke partner/project krijgt quota: maximaal X GB opslag, Y vCPUs compute, Z GB transfer per maand. Dit voorkomt onverwachte kosten. Als je quota nadert, krijg je waarschuwingen. Over quota gaan is mogelijk maar daar is akkoord voor nodig.

We monitoren actuele resource usage en optimaliseren waar nodig. Als een service 4 vCPUs requested maar gemiddeld 0.5 vCPU gebruikt, verkleinen we de request. Dit maakt resources vrij voor anderen en verlaagt kosten.

Services kunnen auto-scale onder load waar nodig, maar hebben max replicas ingesteld (voorkomt cost explosion). Bijvoorbeeld: min 2 replicas, max 10 replicas, scale based on CPU >70%.

Data wordt automatisch verplaatst naar goedkopere storage tiers op basis van access patterns. Hot data op Google (duur maar snel), cold data op Azure Archive (goedkoop maar traag). Dit is geautomatiseerd via lifecycle policies, niet manueel.

We cachen frequente data in MinIO gateway om te voorkomen dat we steeds weer opnieuw data van cloud downloaden (met egress costs). Cache heeft ROI (return on investment) - de kosten van cache storage zijn lager dan bespaarde transfer costs.

Spot instances worden gebruikt waar mogelijk. Voor kritische services (API Gateway, databases) gebruiken we reguliere instances.

Voor baseline capaciteit die we 24/7 nodig hebben, committen we via reserved instances (1-3 jaar) met ~40% discount. Voor burst capacity gebruiken we on-demand.

Trade-offs die we accepteren:

- Cost optimization kost tijd en effort (monitoring, analyzing, optimizing)
- Resource quotas kunnen innovatie beperken (developers worden geconfronteerd met constraints)
- Spot instances kunnen interrupted worden (tradeoff availability vs cost)

Deze trade-offs zijn acceptabel omdat ze ons financieel behapbaar maken. Een platform dat te duur is overleeft niet, hoe technisch goed het ook is.

### 3.3 De Zes-Lagen Architectuur

Nu we de principes hebben gedefinieerd, beschrijven we de concrete architectuur die deze principes realiseert. We gebruiken een zes-lagen model waarbij elke laag een specifieke verantwoordelijkheid heeft en een duidelijk contract heeft met aangrenzende lagen. Dit is een referentie architectuur. De componenten in de lagen zijn mogelijke componenten of worden al gebruikt ter ondersteuning van de andere werkpakketten.



<http://digishape-architecture-platform.s3-website-eu-west-1.amazonaws.com/>

Deze ordening is opzettelijk van boven naar beneden - vanuit gebruikersperspectief.

Gebruikers interacteren met Application Layer (6), die gebruik maakt van Federation Layer (5), enzovoort naar beneden tot aan de fysieke infrastructuur (1).

We beschrijven nu elke laag van onder naar boven - van fundament naar gebruiker.

## 3.4 Layer 1: Multi-Cloud Infrastructure

### 3.4.1 3.4.1 Verantwoordelijkheid en Scope

De Multi-Cloud Infrastructure Layer vormt het fysieke fundament van de hele architectuur. Deze laag bevat de daadwerkelijke compute, storage en networking resources die alle andere lagen mogelijk maken. Het is de enige laag die niet software-defined is, maar bestaat uit hardware en cloud provider services.

**Primaire verantwoordelijkheid:** Het leveren van computing resources (CPU, memory, storage, network) op een manier die betrouwbaar, performant en kosteneffectief is.

**Scope:** Deze laag omvat vier verschillende soorten infrastructuur providers, elk met hun eigen karakteristieken en use cases.

## 3.5 Layer 2: Data Abstraction Layer

### 3.5.1 Verantwoordelijkheid en Scope

De Data Abstraction Layer vormt de kritieke schakel tussen applicaties en de onderliggende multi-cloud infrastructuur. Deze laag abstraheert de heterogeniteit van verschillende storage backends en presenteert een uniforme interface naar bovenliggende lagen. Het is de laag die het mogelijk maakt dat applicaties cloud-agnostisch blijven terwijl ze toch optimaal gebruik maken van verschillende storage providers.

**Primaire verantwoordelijkheid:** Het bieden van een consistente, performante en betrouwbare data access interface die de complexiteit van multi-cloud storage verbergt en intelligente data placement, caching en replicatie mogelijk maakt.

**Scope:** Deze laag omvat storage abstraction (MinIO Gateway), data beweging en replicatie (Rclone), performance optimization (Redis cache), metadata management (STAC).

## 3.6 Layer 3: Security & Identity Layer

### 3.6.1 Verantwoordelijkheid en Scope

De Security & Identity Layer implementeert het zero-trust security model door elke interactie te authentifieren, autoriseren en auditen. Deze laag zorgt ervoor dat multi-party samenwerking veilig kan plaatsvinden zonder dat partners elkaar volledig hoeven te vertrouwen. Het is de laag die juridische compliance (AVG, NIS2) technisch mogelijk maakt.

**Primaire verantwoordelijkheid:** Het waarborgen dat alleen geauthentiseerde en geautoriseerde entiteiten toegang hebben tot resources, dat alle acties worden gelogd voor audit en compliance, en dat secrets veilig worden beheerd zonder dat ze in code of configuratie terechtkomen.

**Scope:** Deze laag omvat identity federation (Keycloak), secrets management (Vault), policy-based authorization (Open Policy Agent), audit logging (ELK Stack), en certificate lifecycle management (Cert-Manager).

## 3.7 Layer 4: Orchestration Layer

### 3.7.1 Verantwoordelijkheid en Scope

De Orchestration Layer beheert de lifecycle van alle gedistribueerde applicaties en services over de multi-cloud infrastructuur heen. Deze laag maakt het mogelijk om declaratief te definiëren wat er moet draaien, en zorgt automatisch voor deployment, scaling, health monitoring en recovery. Het is de laag die chaos tempert tot voorspelbaar gedrag.

**Primaire verantwoordelijkheid:** Het automatisch deployen, schalen en herstellen van applicaties op basis van declaratieve specificaties, het monitoren van de gezondheid van het gehele systeem, en het mogelijk maken van zero-downtime updates via GitOps workflows.

**Scope:** Deze laag omvat container orchestration (Kubernetes control plane), package management (Helm), continuous deployment (ArgoCD), metrics en monitoring (Prometheus), observability dashboards (Grafana).

## 3.8 Layer 5: Federation & API Layer

### 3.8.1 Verantwoordelijkheid en Scope

De Federation & API Layer implementeert de GAIA-X-geïnspireerde federatie-architectuur door services vindbaar te maken, requests intelligent te routeren, en asynchrone communicatie te faciliteren. Deze laag maakt het mogelijk dat gedistribueerde services van verschillende partners naadloos samenwerken alsof ze één coherent systeem vormen.

**Primaire verantwoordelijkheid:** Het bieden van een uniform toegangspunt voor alle platform services (API Gateway), het mogelijk maken dat services elkaar ontdekken en veilig communiceren (Service Mesh), het catalogiseren en vindbaar maken van datasets (Data Catalog), het orchestreren van complexe workflows (Workflow Engine), en het faciliteren van event-driven architectuur (Message Broker).

**Scope:** Deze laag omvat API gateway en routing (Kong/Traefik), service-to-service communicatie (Istio service mesh), dataset discovery (CKAN-based catalog), workflow orchestration (Apache Airflow), en event streaming (Apache Kafka).

## 3.9 3.9 Layer 6: Application Layer

### 3.9.1 3.9.1 Verantwoordelijkheid en Scope

De Application Layer bevat de daadwerkelijke gebruiker-facing applicaties en domein-specifieke tools die de waarde van het platform realiseren. Deze laag is waar expertise wordt gecombineerd met de technische kennis van de onderliggende lagen. Het is de laag die eindgebruikers zien en waarmee ze interacteren.

**Primaire verantwoordelijkheid:** Het leveren van domein-specifieke functionaliteit voor use cases door gebruik te maken van de mogelijkheden van alle onderliggende lagen, zonder dat applicaties directe kennis hoeven te hebben van de complexiteit van multi-cloud infrastructuur, security, of data management.

**Scope:** Deze laag omvat WP4 reconstructie applicaties (web-based visualisaties van historische gebeurtenissen), simulatie tools (OpenCLSim voor logistiek, OpenTNSim voor scheepvaart), data science omgeving (Jupyter Hub), en visualisatie frameworks (deck.gl, QGIS integraties).

Deltares is een onafhankelijk kennisinstituut voor toegepast onderzoek op het gebied van water en ondergrond. Wereldwijd werken we aan slimme oplossingen voor mens, milieu en maatschappij.

**Deltares**

[www.deltares.nl](http://www.deltares.nl)