Net Actuate

Architecting for Cloud Openness

A Guide for Avoiding Hyperscaler Lock-in

Hyperscaler (cloud provider) lock-in is not always apparent. Hyperscalers diminish portability through their comprehensive set of pre-packaged and proprietary services for managing infrastructure and building applications. These hyperscaler-native services are typically only compatible in the provider's environment and make it difficult to migrate workloads to other providers.

Hyperscalers often provide venture-backed start-ups with generous free credits. Growing companies can certainly take advantage of these credits in their early days while also future-proofing their infrastructure OPEX by designing, building, and deploying applications ***without*** locking themselves in. By designing in architectural portability, once the credits run out, they can make informed decisions about their infrastructure instead of being forced into their existing cloud ecosystem for the long term.

The best way to increase cloud options down the road is to architect your cloud environment using non-proprietary or open source alternatives that can easily be run in other environments. This guide describes portable alternatives to hyperscaler proprietary services based on third-party and open-source tooling.

This guide will cover some of the most common infrastructure and service setups used by cloud-native organizations, which include:

1. **Identity and Access Management**
2. **CI/CD and DevOps Pipelines for Application Development**
3. **Programmable Infrastructure**
4. **Kubernetes-based Compute**
5. **VM-based Compute**
6. **Networking Layer**
7. **Data Layer (databases)**
8. **Storage Layer (object, block, file storage)**
9. **Monitoring and Visibility**

How to Read this Guide

As tempting as it may be (and as easy as the hyperscalers make it) to build your application stack using hyperscaler-proprietary services, a small investment in researching and using third-party, open-source tooling today will pay huge dividends in freedom tomorrow. Some of these open source tools are used by the hyperscalers themselves, and most are available in the hyperscalers' marketplaces.

For each infrastructure component and service listed above, we will define a hyperscaler-proprietary way (using AWS as an example) of setting up the base infrastructure and associated services. These will be labeled as ***less portable***, since migrating them to a different environment will require some degree of rearchitecting.

We will also present an alternative option based on open-source solutions, labeled as ***portable***. These open-source alternatives are suitable to be used in a hyperscaler-native environment, and then can also be more easily migrated to a non-hyperscaler environment or even across hyperscalers.

Infrastructure and Service Setups

Identity and Access Management

Identity and Access Management (IAM) governs who can access which resources, under what conditions, and with what level of permission. This includes user authentication, service-to-service authorization, and policy enforcement mechanisms. In hyperscaler environments, IAM is typically deeply integrated into the platform and optimized for convenience.

Less Portable	Portable
WORKLOAD IDENTITY AND ROLE ASSIGNMENT	
Identity and access management revolves entirely around **AWS IAM** with deeply integrated, provider-specific concepts. For instance, IAM instance profiles for EC2 instances, ECS task roles for containers, and execution roles for Lambda functions.	Instead of AWS IAM, you can use open source solutions like **Keycloak** or **OpenIAM** that provide comparable identity management capabilities. For container workloads that need service identity, you can implement: • **Vault by HashiCorp** with its Kubernetes auth method to provide dynamic secrets • **SPIFFE/SPIRE** for workload identity across distributed systems.
ACCESS POLICY DEFINITION AND ENFORCEMENT	
Access policies are commonly written in AWS's policy language with resource patterns using AWS-specific ARN formats. These policies might include AWS-specific condition keys like **`aws:SourceVpc`** or **`aws:PrincipalOrgID`** that have no direct equivalents in other providers. Resource-based policies attached to S3 buckets or SQS queues would use AWS-specific principal identifiers and condition operators.	You can minimize dependency on AWS's policy language and ARN format with the following: • Open Policy Agent (OPA) for flexible, declarative policy definitions across diverse resources • Casbin for fine-grained, model-based access control with various storage adapters These systems use standardized expressions and can be adapted to work with non-AWS resource naming schemes.

Less Portable	Portable
USER AUTHENTICATION AND FEDERATION	
User authentication would typically be implemented through **Amazon Cognito** with AWS-specific user pools, identity pools, and authentication flows. Applications would be designed around Cognito's specific JWT format and claims, AWS-specific social identity provider integration, and Cognito's approach to custom authentication challenges and triggers. The application's security model would often incorporate Cognito-specific concepts like groups and custom attributes that may not map cleanly to other identity providers	Open source alternatives include: • Keycloak, a full-featured identity provider with social login support, MFA, and user federation, or Logto for developer-friendly authentication platforms; • **OpenID Connect** providers like Dex that work with existing identity systems; • **Authentik** for a modern approach to identity management with flexible flows. These solutions provide standard OAuth2/OIDC flows with JWT tokens using standardized claims that aren't tied to a specific cloud provider. They handle social identity provider integration through standard protocols and support custom authentication flows and user attributes in ways that can be migrated between environments. Note that custom flows and user attributes often still require manual mapping and adjustments between systems. The portability is protocol-level, not implementation-level

CI/CD and DevOps Pipelines for Application Development

This is a toolset for integrating code changes, building artifacts, running tests, and deploying to staging or production environments. While hyperscalers offer fully integrated CI/CD tools that streamline this process within their platforms, this is an area full of widely used open source and third-party alternatives.

Less Portable	Portable
PIPELINE DEFINITION AND EXECUTION ENGINE	
A CI/CD architecture on **AWS** would revolve around AWS developer tools like **CodePipeline**, **CodeBuild**, and **CodeDeploy** with numerous AWS-specific integrations. Build pipelines would be defined using AWS CodePipeline's proprietary	Open-first CI/CD implements pipeline definitions as code using provider-agnostic tools like **GitLab CI/CD**, **Jenkins**, or **GitHub Actions**. Pipeline definitions would be stored in the application repository rather than in a provider-specific

Less Portable	Portable
JSON structure with stages and actions that are tightly coupled to AWS services.	service, allowing them to be executed in any environment.
BUILD ENVIRONMENT CONFIGURATION	
Build environments would be configured using CodeBuild's **`buildspec.yml`** format with AWS-specific environment variables, phases, and artifacts definitions. The pipeline would likely use CodeBuild's specific behavior around caching, VPC connectivity, and privileged mode execution that differs from other CI systems.	Build environments can be defined as Docker containers with explicit dependencies, ensuring consistent behavior regardless of where the pipeline runs. These containers would include all necessary build tools, language runtimes, and dependencies, allowing builds to execute identically across different CI systems.
DEPLOYMENT MECHANISM	
Deployment using AWS CodeDeploy involves AWS-specific elements such as AppSpec files, deployment configurations, and deployment groups that are tightly integrated with AWS services like EC2, ECS, or Lambda. Pipelines built with AWS CodePipeline may incorporate manual approval actions and SNS-based notifications, leveraging features that are native to AWS.	Deployment would use provider-agnostic approaches like Helm or Kustomize for Kubernetes resources or provider-neutral infrastructure as code tools. Deployment scripts would externalize environment-specific configuration through a configuration management system like Ansible, Chef, or Puppet, allowing the same deployment process to target different environments.
SOURCE CODE AND ARTIFACT STORAGE	
Source code would likely be stored in CodeCommit with AWS-specific triggers and integration patterns, or would use CodeStar connections to other repositories with AWS-specific authentication mechanisms. Artifact storage would rely on S3 with CodePipeline's specific artifact format and encryption approach.	Artifact storage would use standard repositories appropriate to the artifact type—container images would be stored in OCI-compliant registries like **Harbor**, **Docker Hub**, or **Quay.io**, while language-specific packages would use standard repositories like **npm**, **PyPI**, or **Maven Central**. These repositories can be self-hosted or used as services, providing consistency across environments.

Programmable Infrastructure

Programmable infrastructure enables teams to define, provision, and manage cloud infrastructure using machine-readable configuration files. In a cloud-native environment, infrastructure-as-code supports repeatability, scalability, and automation by treating infrastructure as software—version-controlled, testable, and modular. While hyperscalers like AWS provide proprietary IaC solutions such as CloudFormation, these are tightly coupled with their platforms and often contain unique behaviors and syntax. To avoid lock-in and maintain flexibility across cloud providers or hybrid environments, teams

can adopt provider-agnostic tools like **OpenTofu** or **Pulumi**, which allow for reusable, modular, and portable infrastructure configurations.

Less Portable	Portable
INFRASTRUCTURE DEFINITION AND SYNTAX	
Traditional infrastructure on **AWS** would be defined using **CloudFormation** templates with heavy use of intrinsic functions like **`!Ref`**, **`!GetAtt`**, and **`!Sub`** that only work in AWS. Templates would use AWS-specific pseudo-parameters like **`AWS::Region`** and **`AWS::AccountId`** and would incorporate CloudFormation-specific concepts like nested stacks and custom resources.	Open-first CI/CD implements pipeline definitions as code using provider-agnostic tools like **GitLab CI/CD**, **Jenkins**, or **GitHub Actions**. Pipeline definitions would be stored in the application repository rather than in a provider-specific service, allowing them to be executed in any environment.
RESOURCE ABSTRACTIONS AND INPUTS	
Resource definitions would use AWS-specific properties and return values, often with subtle differences from similar resources in other clouds. For example, an **`AWS::EC2::Instance`** resource has unique approaches to user data, network interfaces, and IAM role attachment that don't directly map to other providers.	A "database" module would present consistent inputs like **`engine_type`**, **`version`**, **`size`**, and **`high_availability`**, while implementing the appropriate resources for each provider underneath. This approach allows application teams to define infrastructure requirements in provider-neutral terms while enabling actual deployment to any supported provider.
STACK REFERENCES	
The infrastructure would likely use CloudFormation exports and imports for cross-stack references, which creates dependencies that are difficult to restructure. Change management would rely on CloudFormation change sets and stack policies that have AWS-specific semantics and limitations.	Resource references would use consistent variable and output naming conventions across modules, creating a uniform interface regardless of the underlying provider. For instance, database connection details would be exposed with consistent output names like **`host`**, **`port`**, and **`connection_string`** regardless of whether the database is running on **AWS RDS**, **Azure Database**, or a self-hosted option.
DEPLOYMENT AND CONFIGURATION INTEGRATION	
The deployment process might incorporate AWS-specific services like **Systems Manager Parameter Store** for configuration and **AWS Service Catalog** for template distribution, creating additional coupling to the AWS ecosystem.	In contrast, portable infrastructure designs focus on separating core functional requirements from provider-specific optimizations. This allows infrastructure to be deployed across different environments with consistent baseline functionality, while enabling advanced

Less Portable	Portable
	features—such as enhanced networking, encryption, or monitoring—where supported. Configuration systems are designed to gracefully degrade when certain features aren't available, ensuring compatibility across clouds and on-prem systems.

Kubernetes-Based Compute

Kubernetes-based compute is the foundation for deploying, managing, and scaling containerized applications in cloud-native environments. It abstracts underlying infrastructure, automates orchestration tasks, and provides consistent APIs for deployment, networking, and storage. In hyperscaler platforms like AWS, managed Kubernetes offerings such as **Amazon EKS** simplify cluster operations but often integrate deeply with proprietary services, making migration challenging. By contrast, open-first approaches focus on running Kubernetes in a provider-neutral way, using certified distributions and portable interfaces that allow workloads to be moved seamlessly between clouds or to on-premises environments.

Less Portable	Portable
CLUSTER ACCESS AND IDENTITY MANAGEMENT	
In a traditional **AWS** approach, **Kubernetes** workloads would be deployed on **Amazon EKS** with numerous AWS-specific integrations. The cluster would use **AWS IAM authenticator** for Kubernetes, requiring AWS credentials for all cluster access. Pod identity would be managed through **IAM Roles for Service Accounts (IRSA)**, an AWS-specific mechanism that injects AWS credentials into pods.	Open-first **Kubernetes** deployments use certified Kubernetes distributions that maintain compatibility with the standard Kubernetes API. Applications would be deployed to vanilla Kubernetes clusters without dependencies on provider-specific extensions or controllers, allowing them to run consistently across environments.
STORAGE	
Storage would be provisioned using the **EBS CSI** driver with volume types like gp3 or io2. Persistent volume claims would result in **Amazon EBS** volumes with AWS-specific lifecycle and snapshot capabilities.	For storage, applications would use the **Container Storage Interface (CSI)**The specific CSI driver would be selected based on the environment (e.g., **AWS EBS CSI Driver in AWS, Azure Disk CSI Driver in Azure**), but the application's `PersistentVolumeClaim` objects would remain identical across environments.

Less Portable	Portable
NETWORKING	
Networking would use the **AWS VPC CNI** plugin with AWS-specific IP address management and security group integration. Ingress controllers would be implemented using the **AWS Load Balancer Controller**, which creates AWS-specific annotations to control Application Load Balancers or Network Load Balancers.	Networking would use the **Container Network Interface (CNI)** with portable implementations like Calico or Cilium that work across environments. Network policies would be defined using standard Kubernetes NetworkPolicy resources rather than provider-specific annotations or extensions, ensuring consistent security enforcement regardless of the underlying infrastructure. Layer 7 Service exposure for HTTP and other application traffic would use standard Kubernetes Ingress resources with portable controllers like **NGINX** or **Envoy**, rather than provider-specific ingress implementations. Layer 4 network load balancing would use appliances such as MetalLB.
OBSERVABILITY	
The Kubernetes cluster would be tightly integrated with AWS services like **CloudWatch** for logging and monitoring and **AWS X-Ray** for tracing.	Observability can be architected using open source technologies such as **OpenSearch**, **Logstash**, **Garylog**, **Kibana**, **Prometheus**, **OpenTelemetry** and **Grafana**.

VM-based Compute

In cloud-native environments, VM-based compute is used for workloads that require long-running processes, full OS-level control, or legacy systems that aren't yet containerized. These virtual machines are commonly used for stateful applications, bespoke system configurations, or environments where compliance or performance tuning is critical. While hyperscalers like AWS offer tightly integrated services around VMs through EC2 and its ecosystem, these implementations often involve provider-specific formats, tools, and workflows that hinder portability. A portable approach leverages open tools and standards to define and manage VM lifecycle, configuration, and scaling across multiple cloud and on-premises environments.

Less Portable	Portable
IMAGE BUILDING AND INITIALIZATION	
A traditional VM-based architecture in **AWS** would use EC2 instances with AWS-specific **Amazon**	Open-first VM architectures use tools like **HashiCorp Packer** to build machine images

Less Portable	Portable
Machine Images (AMIs) that contain AWS-specific initialization code and tools. Instances would be launched with EC2-specific user data scripts that handle bootstrapping and configuration, often relying on the particular format and timing guarantees of EC2 user data execution.	from a single template that can target multiple providers. Image definitions would include provider-specific builders but share common provisioning scripts, ensuring consistent configuration regardless of the target environment. For instance, a single Packer template might include builders for **AWS AMI**, **Azure Image**, **Google Cloud Image**, and **VMware OVA formats**, but would use the same shell scripts, **Ansible** playbooks, or other provisioning tools for all targets. This approach ensures consistent system configuration and application deployment across environments.
VM SCALING AND ORCHESTRATION	
The architecture would use **EC2 Auto Scaling Groups** with AWS-specific launch templates, scaling policies based on **CloudWatch** metrics, and lifecycle hooks that integrate with other AWS services. Applications would be designed around EC2 instance types with specific performance characteristics and AWS's approach to ephemeral storage.	VM orchestration would use provider-agnostic tools like **HashiCorp Nomad** or **Terraform** to manage VM lifecycle and scaling. Instance type requirements would be expressed in terms of CPU, memory, and storage, with mapping to appropriate instance types for each provider handled by the orchestration tool or through configuration.
METADATA ACCESS AND INSTANCE CONFIGURATION	
Instances would use EC2 instance metadata service at the 169.254.169.254 endpoint for configuration data, credentials, and user data. Applications might directly query the EC2 metadata service for information about the instance's identity, network configuration, or attached IAM role.	Configuration and bootstrapping would use **`cloud-init`** with provider-agnostic syntax for common operations like user creation, package installation, and file writing. **`Cloud-init`**'s data source abstraction allows it to obtain configuration from different providers while presenting a consistent interface to the initialization scripts.
MANAGEMENT, MONITORING, AND DEPLOYMENT	
Instance management would rely on AWS Systems Manager for patching, configuration, and command execution, using the **AWS-specific SSM Agent** that must be installed on all instances. Monitoring would use CloudWatch with the CloudWatch agent for metrics and logs collection in AWS-specific formats.	Application deployment to VMs would use standard configuration management tools like **Ansible**, **Chef**, or **SaltStack** that work consistently across environments. These tools would use inventory and configuration that can be dynamically generated based on the target environment, allowing the same playbooks or recipes to be used anywhere. Prometheus Node Exporter can be used for

Less Portable	Portable
	monitoring, which collects detailed system metrics including CPU, memory, disk, and network utilization from each virtual machine. The Node Exporter runs as a lightweight daemon on each VM and exposes metrics in a format that Prometheus can scrape.

Networking Layer

In a cloud-native environment, the networking layer provides the fundamental communication backbone that connects services, workloads, and users. It handles traffic routing, isolation, firewalling, and secure connectivity across distributed components and hybrid infrastructure. Hyperscaler platforms like AWS abstract and manage this layer with deeply integrated, proprietary tools (e.g., VPC, security groups, NAT gateways), which can make portability challenging. A portable approach replaces these vendor-specific constructs with open-source networking tools and protocols, allowing organizations to design and operate cloud-agnostic networks with consistent behavior across environments.

Less Portable	Portable
VIRTUALIZED NETWORKING	
Traditional **AWS** networking architecture revolves around the Virtual Private Cloud (VPC) with numerous AWS-specific constructs. The network would be divided into subnets with AWS-specific routing tables, network ACLs, and security groups that use AWS's particular rule formats and behavior.	Replace AWS VPC with standard networking concepts using tools like **OpenStack Neutron** or open-source virtualization platforms. Create network segments with standard VLANs or overlay networks using technologies like **VXLAN**. Define security using standard Linux iptables or nftables for firewall rules, and implement standard network access controls using tools like **Linux** namespaces or containers with defined network policies.
CROSS-NETWORK CONNECTIVITY AND VPN	
Connectivity between VPCs would use **VPC Peering** or **Transit Gateway**, both AWS-specific services with their own configuration models and limitations. External connectivity would use Internet Gateways, NAT Gateways, and VPN Connections with AWS-specific configuration and routing requirements.	For connectivity between network segments, use standard routing protocols like BGP with tools such as **FRRouting** or **Bird**. Direct peering can be established through Internet Exchange Points (IXs) using BGP sessions. For secure connections, deploy open-source VPN solutions like **WireGuard** or **OpenVPN**, or use IPsec with **strongSwan** or **Libreswan** to create encrypted tunnels between networks. Cross-network connectivity would use standard VPN

Less Portable	Portable
	protocols like IPsec or WireGuard that can be implemented anywhere, rather than provider-specific constructs like VPC peering or Transit Gateway.
LOAD BALANCING AND TRAFFIC ROUTING	
Load balancing would use **Elastic Load Balancing** with Application Load Balancers, Network Load Balancers, or Gateway Load Balancers, each with AWS-specific target groups, listener rules, and integration with other AWS services.	Instead of AWS's Elastic Load Balancing service, you can use **NGINX** or **HAProxy** as powerful open-source load balancers. NGINX provides excellent performance for HTTP/HTTPS traffic with features like SSL termination, keepalive connections, request routing, and health checks. HAProxy excels at TCP and HTTP load balancing with detailed metrics and high-availability configurations. Both can be deployed on standard **Linux** servers and configured for automatic scaling.

Data Layer (Databases)

In a cloud-native environment, the data layer—comprising databases and associated services—ensures persistent, durable, and reliable storage of application data. Traditional cloud-native designs often tie databases tightly to managed services from a specific cloud provider, but a portable approach uses open-source databases and Kubernetes-native tooling to allow consistent operation across any environment.

Less Portable	Portable
DATABASE ENGINES AND DEPLOYMENT MODELS	
A traditional **AWS** database architecture would use fully managed services like **RDS**, **Aurora**, or **DynamoDB** with deep integration to AWS-specific features. For relational workloads, applications would typically use **Amazon Aurora** with proprietary extensions for **MySQL** or **PostgreSQL** that provide AWS-specific capabilities like Global Database, serverless scaling, or zero-ETL integration with **Redshift**.	Open-first database architectures use standard database distributions deployed in a provider-agnostic way. For example, **PostgreSQL**, **MySQL**, or **MongoDB** would be deployed using Kubernetes operators like the **PostgreSQL Operator** or the **MongoDB Community Kubernetes Operator**, providing consistent deployment and lifecycle management across environments. These operators handle provisioning, high availability, backup, and scaling in a consistent manner regardless of the underlying infrastructure. For instance, the PostgreSQL Operator can create a highly available PostgreSQL cluster with

Less Portable	Portable
	streaming replication on any Kubernetes cluster, whether running in AWS, Azure, or on-premises.
CONNECTION MANAGEMENT AND HIGH AVAILABILITY	
Connection management would rely on AWS-specific endpoints and DNS naming conventions, often using the AWS-specific RDS Proxy for connection pooling. High availability would use Aurora's specific implementation of read replicas and automatic failover that doesn't directly map to standard database clustering.	Connection management would use standard connection pooling solutions like PgBouncer Connection management would use standard connection pooling solutions like **PgBouncer** for PostgreSQL or **ProxySQL** for MySQL, deployed alongside the application in any environment. These provide consistent connection handling, pooling, and failover behavior regardless of the database deployment.

Storage Layer

In a cloud-native environment, the storage layer provides the foundational capabilities for storing and retrieving data in various formats: object, block, and file. This layer supports diverse workloads—from serving static assets to providing persistent volumes for stateful applications. A cloud-native approach emphasizes decoupling storage from any single provider, enabling applications to remain portable and infrastructure-agnostic while still meeting performance, scalability, and durability requirements.

Less Portable	Portable
OBJECT STORAGE	
Traditional storage architecture in **AWS** would use S3 for object storage with numerous AWS-specific features. Applications would use bucket policies with AWS IAM principals and conditions, lifecycle rules specific to S3's storage tiers, and event notifications with AWS-specific targets like **SNS**, **SQS**, or **Lambda**.	Open-first object storage implementations use S3-compatible APIs that are widely supported across providers and tools. Solutions like **MinIO** provide S3-compatible object storage that can be deployed anywhere, while libraries like the **AWS SDK** with the S3-compatible endpoint configuration allow applications to use any S3-compatible storage without code changes. Ceph RADOS Gateway offers another robust S3-compatible solution that integrates with Ceph's distributed storage cluster, providing enterprise-grade features like multi-tenancy, data replication, and erasure coding.

Less Portable	Portable
BLOCK STORAGE	
For block storage, applications would use EBS volumes with AWS-specific volume types, IOPS provisioning models, and snapshot capabilities. Attachment and detachment would follow EC2-specific behavior and limitations, and data persistence would rely on AWS's specific guarantees around availability zones and regions.	For block storage in containerized environments, applications use **Kubernetes PersistentVolumeClaims (PVC)** with the **Container Storage Interface (CSI)** to request storage through a consistent API that works across all environments. Open-source solutions like Rook's Ceph RBD integration and Longhorn provide **CSI drivers** that can provision block storage from distributed storage clusters, while the same underlying storage can also serve VMs and bare-metal workloads through Ceph's RBD daemon. This approach allows organizations to use the same storage infrastructure and APIs whether deploying on cloud platforms or self-managed clusters.
SHARED FILE STORAGE	
File storage would typically use **EFS** with AWS-specific mount targets, access points, and	File storage would use standard protocols like **NFS** or **SMB** that are supported across environments.
lifecycle management. Applications would be designed around EFS's particular performance characteristics, throughput modes, and integration with other AWS services through file system policies.	In **Kubernetes**, the **Rook** operator with **Ceph** provides a consistent way to provision file storage in any environment, with the same **CephFS** PersistentVolumeClaim being usable regardless of the underlying infrastructure.
DATA MANAGEMENT	
Data transfer between storage services would often use **AWS Transfer Family** or **AWS DataSync** with AWS-specific configuration and behavior. Access patterns would be optimized for AWS's specific pricing models, such as using S3 Transfer Acceleration or Direct Connect for cost-effective data movement.	Data lifecycle management would be implemented using standard policies and tools rather than provider-specific features. For example, object lifecycles could be managed by application logic or by scheduled jobs that apply consistent policies across any S3-compatible storage.

Monitoring and Visibility

In a cloud-native environment, monitoring and visibility encompass the collection of metrics, logs, and traces; the analysis of system behavior; and the setup of alerts and dashboards for real-time observability.

Less Portable	Portable
METRICS AND LOG COLLECTION	
Traditional AWS monitoring architecture would center on **CloudWatch** with AWS-specific metrics, logs, and alarms. Applications would publish custom metrics using the CloudWatch API with AWS's specific namespaces, dimensions, and statistics. Log collection would use CloudWatch Logs with AWS-specific log groups, log streams, and retention policies.	Open-first observability implements the **OpenTelemetry** framework for metrics, logs, and traces that can be collected consistently across environments. Applications would be instrumented using the OpenTelemetry SDK, producing telemetry data in standard formats that can be processed by various backends. It can forward data to **Prometheus** as a backend, maintaining consistent labeling and naming conventions so the same PromQL queries work regardless of where the application is running. Traces can be sent to backends like Jaeger or Zipkin for distributed tracing analysis.
INFRASTRUCTURE AND DATABASE MONITORING	
Infrastructure monitoring would often use AWS Config with AWS-specific rules and remediation actions. Performance insights for databases would use the AWS-specific **RDS Performance Insights** or similar service-specific monitoring tools.	Open-first environments monitor infrastructure using standard tools such as Prometheus exporters, OpenTelemetry collectors, or agents that expose metrics in open formats. For databases, standard exporters or OpenTelemetry SDKs can collect performance metrics that work across self-hosted or managed instances in any environment, providing consistent visibility without relying on provider-specific APIs.
DISTRIBUTED TRACING	
For distributed tracing, applications would use **AWS X-Ray** with the **X-Ray SDK** for instrumentation, producing traces in AWS's specific format with X-Ray-specific concepts like segments, subsegments, and annotations. Sampling and trace collection would follow X-Ray's specific behavior and limitations.	Distributed tracing would use OpenTelemetry with consistent span naming and attribute conventions. The same traces would be collected regardless of the environment, allowing for consistent visualization and analysis in backends like **Jaeger** or **Zipkin**.
ALERTING AND DASHBOARDING	
Alerting would use **CloudWatch Alarms** with integration to SNS for notification or to other AWS services for automated remediation. Dashboards would be created in the CloudWatch console with AWS-specific widgets and visualization options.	Dashboarding and alerting would use **Grafana** or **Icinga** with dashboards defined as code, allowing them to be version-controlled and deployed consistently across environments. Alerts would be defined using standard mechanisms like Prometheus **AlertManager** rules, with notification through standard channels like email, Slack, or PagerDuty.

Migrating to the Open Network Edge

An open approach to infrastructure development allows fast-growing companies to leverage best-in-class open source tooling while also ensuring portability between environments. This provides the critical flexibility to evolve and scale your technology stack and optimize for performance, costs, and new revenue opportunities as your company evolves.

NetActuate's **Open Network Edge (ONE)** is a globally distributed infrastructure-as-a-service (Iaas) built using open source technologies. It provides out-of-the-box implementations of widely deployed open source software, including operating system, network operating systems, monitoring software, orchestration tools, and everything else needed to scale your platform. Our team of experts can help you re-architect your current cloud environment and set up your infrastructure in such a way that it is easily extensible to other environments.

Schedule a pressure-free 30-minute call with one of our engineers to learn how NetActuate can help you build and deploy your solution at scale.

linkedin.com/company/netactuate-inc

github.com/netactuate

youtube.com/@NetActuate

netactuate.com