# DevOps Landscape:

Linux

Shell scripting

Python

SSL certs

DNS

APIS and api management - apigee , swagger UI , apigateway , rate limit and pagination

Middleware - applicationservers, webservers and queuing systems , kafka

Application servers - tomcat, httpd, weblogic, jboss,

Webservers - Nginx

Queuing - Apache kafka, Sqs, Redis

Config server - to maintain the application properties

IAC - Cloudformation, Terraform, boto3 or CDK , diff between the aws sdk, cdk and boto3

Vault - for secrets management

Docker

K8

Otel concepts

Monitoring and observability - how are they different from each other ?

Logging

Loadbalancing concepts

Backups

JAVA concepts - jre, jdk, jvm , GC, threads, thread dumps

AWS

Application Architecture patterns: Monolithic, Microservices, Serverless, Event Driven , SOA, Serverless Saga Pattern, pub-sub , polling

<https://hookdeck.com/webhooks/guides/when-to-use-webhooks#:~:text=Polling%20requests%20are%20made%20by,a%20new%20event%20or%20not>.

# DevOps TOOLS:

**Version control:** github, gitlab and bitbucket

**Build Tools:**  Maven and Gradle

**CI:** Github Actions (github pipeline), Jenkins, GitOps (Gitlab pipeline)

**CD:** Jenkins, UrbanCode Deploy, Ansible, Harness

**Configuration Management:** Ansible, Chef, Puppet

Loadbalancing concepts

Backups

# Otel Agents:

APM - Application performance monitoring

Cloudwatch - cloudwatch agent, Metrics, Traces and Cloudwatch Synthesis

Dynatrace - Oneagent, it also monitor logs

Opensource Logg Shipper - FluentBit, Fluend

ELK - logshipper

Splunk -

Prometheus - Node exporter

DataDog:

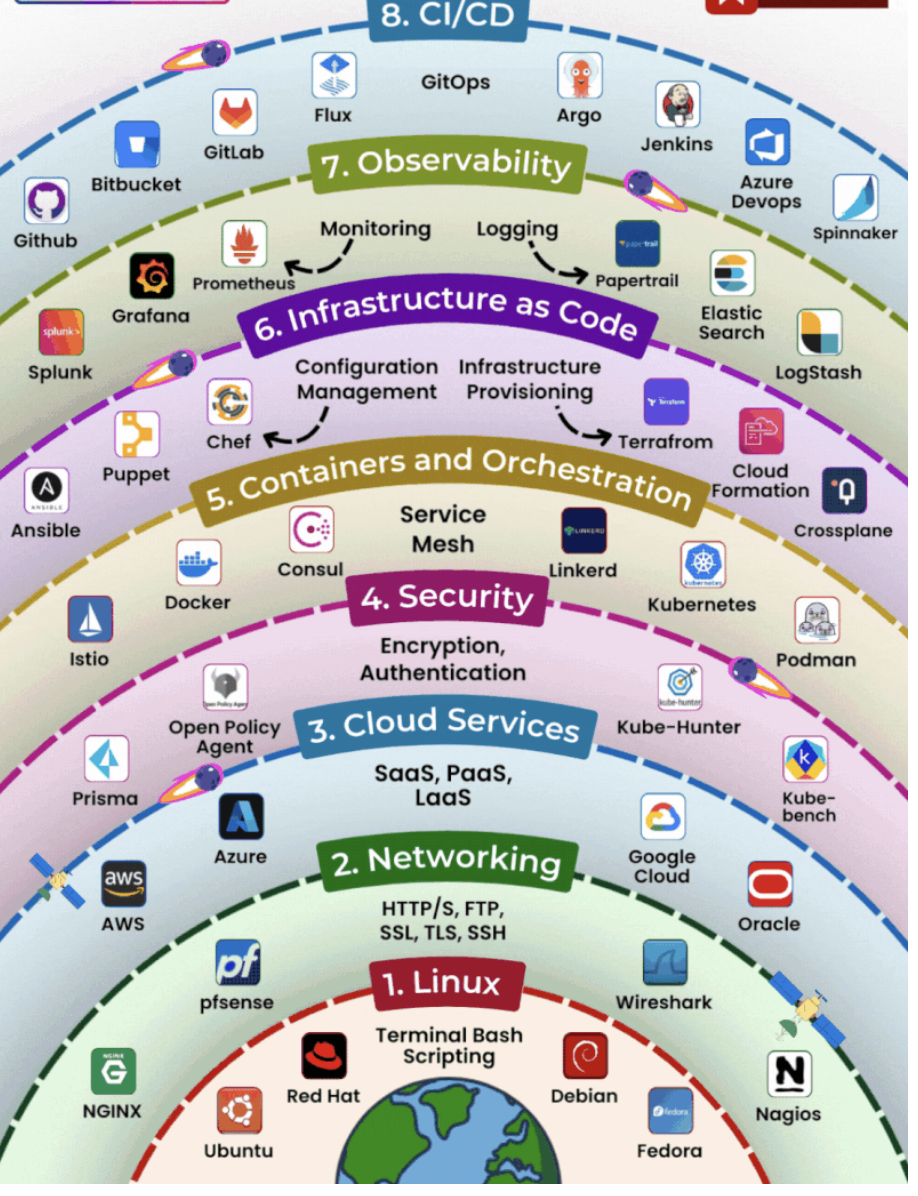
**Pipelines and Architecture Patterns:**

**Single branch pipeline - deployment**

**Multibranch pipeline - delivery**

**IAC pipeline with terraform and Jenkins**

**How to integrate OPA with terraform for infrastructure scan**



# DevOps includes:

Continuous Deployment and Delivery ?

Logging -

Monitoring -

DNS ?

LoadBalancing ?

Autoscaling ?

Networking with protocols - OSI

Automated SSL Certificate Rotation

Secret Management (e.g., Vault, SOPS)

Canary Deployments with Progressive Rollouts

Autoscaling with Metrics-Driven Policies

Immutable Infrastructure with Blue-Green Deployments

Debugging with Distributed Tracing

Cluster Autoscaler Tuning for Workload Spikes

Self-Healing Infrastructure (e.g., Health Checks & Restart Policies)

GitOps with Advanced Reconciliation Loops

Service Mesh Implementations (e.g., mTLS, Traffic Shaping)

I could go on...

Operating tools is not Devops - just a part of it.

Be open minded - learning becomes easy.

# 80% of DevOps Engineers are terrible at troubleshooting.

# **Here's the trick to improve:**

### Master Linux first.

### - You can handle the command line better.

### - You can grasp process management better.

### - You can understand networking fundamentals better.

### - You can approach system architecture better.

### - You can automate repetitive tasks better.

### As a whole, you can troubleshoot better.

### Remember, DevOps is 20% building and 80% operating.

### It's not about shiny tools.

### It's about understanding what's under the hood.

### Want to be an outstanding DevOps pro?

### Master Linux first.

# Pipeline Types:

Monolithic - deployment to aws ec2 - either windows or linux

Serverless - Deployment to lambda

Microservices - Deployment to ECS, EKS nodegroup/Fargate

StaticWebsite - Deployment to S3, Amplify

# explain about devops as its not a technology where we maintain the different tools but it also a culture in an organization to improve the application delivery and business improvement

DevOps is more than just a collection of tools; it is a **philosophy and cultural shift** aimed at breaking down silos between development (Dev) and operations (Ops) teams. It fosters **collaboration, communication, and continuous improvement** to streamline the delivery of software and enhance business outcomes.

### **Key Aspects of DevOps Culture:**

1. **Collaboration and Shared Responsibility**:
   * In a traditional setup, development and operations work in isolation, leading to misunderstandings, delays, and inefficiencies. DevOps bridges this gap by encouraging **cross-functional teams** to work together throughout the software lifecycle.
   * Developers take ownership of their code beyond deployment, and operations teams contribute to improving infrastructure for rapid delivery.
2. **Continuous Feedback and Learning**:
   * DevOps emphasizes learning from failures and successes. Teams adopt a mindset of experimentation, where small and incremental changes are tested and evaluated frequently.
   * Feedback loops—automated or manual—help improve code quality, infrastructure reliability, and customer satisfaction.
3. **Customer-Centric Approach**:
   * The ultimate goal of DevOps is to deliver value to customers faster and more reliably. By aligning technical goals with business objectives, DevOps ensures that teams focus on solving customer problems effectively.
4. **Automation and Process Standardization**:
   * While DevOps is not a tool, it heavily relies on tools to automate repetitive tasks, ensure consistency, and reduce human error. For example, CI/CD pipelines, automated testing, and infrastructure-as-code are key practices in DevOps.
5. **Continuous Improvement**:
   * A core tenet of DevOps is the ongoing optimization of processes. Teams analyze data, learn from incidents, and make iterative changes to enhance efficiency and performance.
6. **End-to-End Responsibility**:
   * Teams in a DevOps culture own their products end-to-end—from development and deployment to monitoring and incident resolution. This accountability drives better quality and reliability.

### **How DevOps Impacts Business:**

1. **Faster Time to Market**:
   * With automation and streamlined processes, businesses can release features, updates, and bug fixes faster, giving them a competitive edge.
2. **Improved Reliability**:
   * Practices like automated testing, monitoring, and rollback strategies ensure that changes are safe and downtime is minimized.
3. **Cost Efficiency**:
   * Early detection of issues, efficient resource utilization, and reduced manual work contribute to lower costs.
4. **Enhanced Innovation**:
   * By reducing the time spent on mundane tasks, DevOps frees up resources for innovation and experimentation.
5. **Increased Customer Satisfaction**:
   * Delivering high-quality products quickly and reliably translates to better user experiences and loyalty.

### **Tools Supporting the Culture:**

While tools like Jenkins, GitHub, Docker, SonarQube, Twistlock, and Artifactory play a role in implementing DevOps, they are only enablers of the culture. Their true value is realized when they are combined with practices like:

* **Continuous Integration/Continuous Delivery (CI/CD)**,
* **Monitoring and Observability**,
* **Infrastructure as Code (IaC)**,
* **Version Control and Collaboration**.

By adopting DevOps as a culture, organizations do more than maintain tools—they foster an environment where **innovation, collaboration, and efficiency** thrive, ultimately improving both application delivery and business outcomes.

# Diff between the continuous deployment and continuous delivery , how it impacts the devops and when to use each of them ?

### **Difference Between Continuous Delivery and Continuous Deployment**

| **Aspect** | **Continuous Delivery** | **Continuous Deployment** |
| --- | --- | --- |
| **Definition** | Ensures code changes are automatically tested and prepared for deployment, but requires **manual approval** for production deployment. | Automates the entire process, including deployment to production, without any manual intervention. |
| **Automation Scope** | Automated until staging; manual steps for production release. | Fully automated, including production deployment. |
| **Deployment Frequency** | Controlled; depends on when approvals are provided. | High frequency; changes are deployed immediately after passing tests. |
| **Human Involvement** | Requires human approval or oversight before production release. | No manual oversight; relies entirely on automated testing and monitoring. |
| **Use Cases** | For organizations needing more control or regulatory/compliance checks before deployment. | For organizations focusing on speed, with strong automation and robust testing processes. |
| **Risk** | Lower, as manual reviews act as a safety net. | Potentially higher, relying heavily on automated systems for quality assurance. |
| **Business Impact** | Slower but safer releases, balancing speed with oversight. | Fast, frequent releases enabling rapid feedback and iteration. |

### **How Continuous Delivery and Continuous Deployment Impact DevOps**

#### **1. Continuous Delivery Impact:**

* **Improved Collaboration**: Developers and operations work together to maintain a deployable system at all times.
* **Increased Control**: Manual approval allows business stakeholders to align releases with strategic goals.
* **Builds Trust**: Teams gain confidence in the system as every release undergoes automated testing and validation.
* **Slower Feedback Loops**: While faster than traditional methods, the manual intervention may delay gathering feedback from end users.

#### **2. Continuous Deployment Impact:**

* **Accelerated Delivery**: Fully automated pipelines enable rapid delivery of features and bug fixes, reducing time-to-market.
* **Faster Feedback**: Real-time deployment allows teams to learn from user behavior and iterate quickly.
* **Higher Reliance on Automation**: Strong test coverage and monitoring become critical to ensuring stability and reliability.
* **Potential Risks**: Faulty releases could go live if there are gaps in the testing or monitoring processes.

### **When to Use Each Practice**

#### **Use Continuous Delivery When:**

1. **Regulatory or Compliance Requirements**:
   * Industries like healthcare, banking, or government often require audits or approvals before deploying changes to production.
2. **Higher Risk Tolerance**:
   * Manual approval ensures that critical releases or high-stakes changes undergo thorough review.
3. **Stakeholder Involvement**:
   * When business teams or product owners want to control the timing of new feature rollouts.
4. **Teams New to DevOps**:
   * Continuous Delivery provides a stepping stone to full automation by focusing on a deployable state without forcing immediate production deployment.

#### **Use Continuous Deployment When:**

1. **Focus on Speed and Agility**:
   * Organizations that prioritize rapid innovation and iterative development (e.g., startups or SaaS providers).
2. **Mature Automation Practices**:
   * Teams with high confidence in their automated testing, monitoring, and rollback strategies.
3. **Low Risk of Failure**:
   * Applications with low criticality or those that can quickly roll back changes without impacting users.
4. **Continuous Feedback**:
   * When immediate feedback from end users is a priority for fine-tuning features or resolving issues.

### **DevOps Alignment:**

Both Continuous Delivery and Continuous Deployment align with the DevOps goal of shortening the software development lifecycle while maintaining high quality. Here's how:

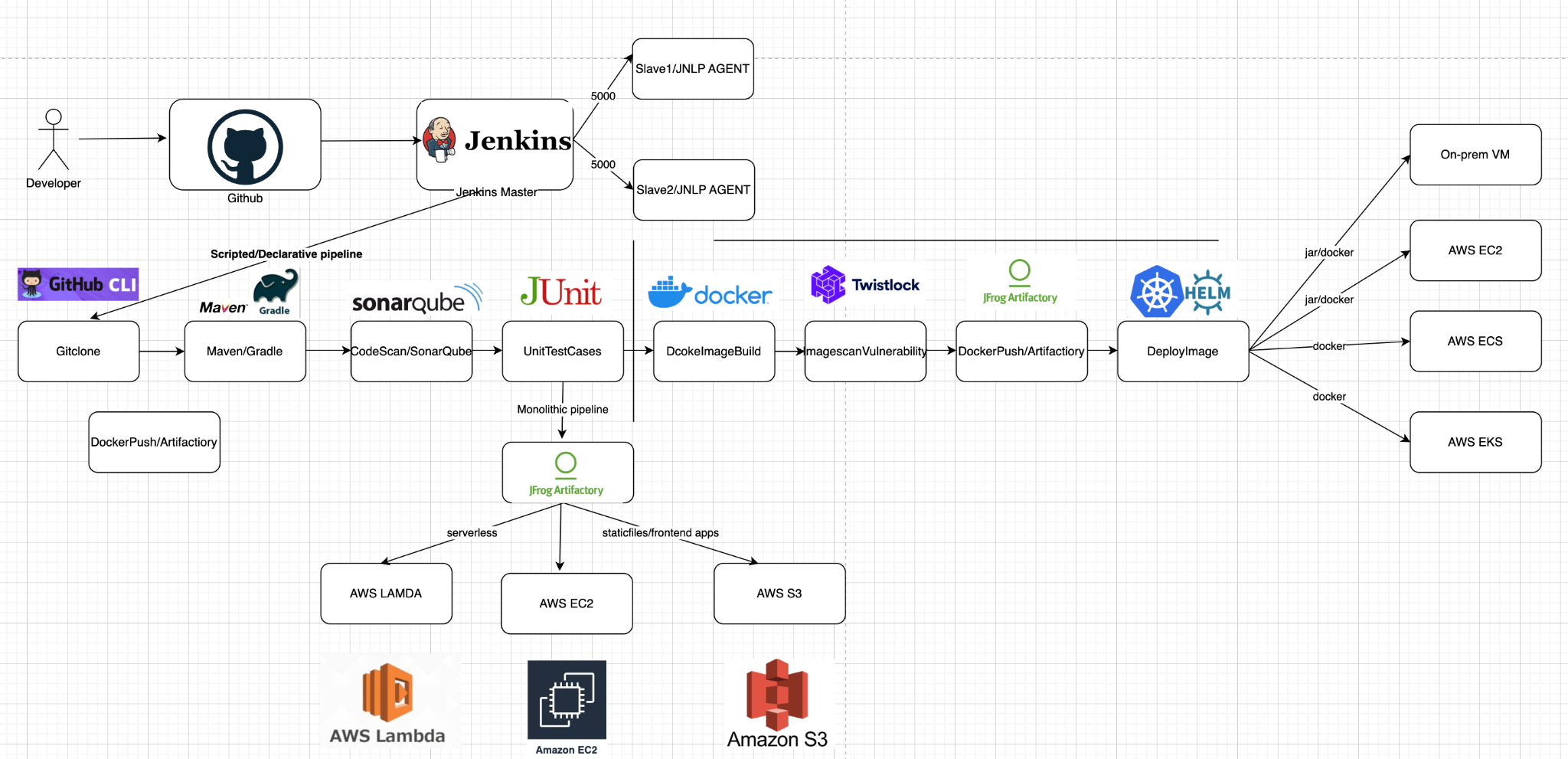
* **Continuous Delivery** supports a **controlled DevOps culture**, balancing automation with manual checkpoints.
* **Continuous Deployment** represents the pinnacle of **DevOps maturity**, embracing full automation for rapid and reliable software delivery.

### **Key Considerations for Implementation:**

* Start with **Continuous Delivery** if you’re transitioning from traditional practices to DevOps. It’s easier to implement and builds confidence in automation.
* Gradually evolve to **Continuous Deployment** once your team has robust testing, monitoring, and rollback capabilities in place.

# An Enterprise DevOps Pipeline contain below stages:

# 



**Version control:** github, gitlab and bitbucket

**Build Tools:**  Maven and Gradle

**CI:** Github Actions (github pipeline), Jenkins, GitOps (Gitlab pipeline)

**CD:** Jenkins, UrbanCode Deploy, Ansible, Harness

**Configuration Management:** Ansible, Chef, Puppet

Loadbalancing concepts

Backups

# 

# 

# Otel Agents:

APM - Application performance monitoring

Cloudwatch - cloudwatch agent, Metrics, Traces and Cloudwatch Synthesis

Dynatrace - Oneagent, it also monitor logs

Opensource Logg Shipper - FluentBit, Fluend

ELK - logshipper

Splunk -

Prometheus - Node exporter

DataDog:

# Understanding Application Servers such as WebSphere, WebLogic, JBoss, Tomcat, IIS, etc. why to use them and what is the significance of these. why the modern applications such as the latest version of spring boot application are not using any application servers in enter prices applications ?

### **Understanding Application Servers: WebSphere, WebLogic, JBoss, Tomcat, IIS**

An **Application Server** is a software platform used to develop, deploy, and run applications, typically web-based. These servers provide an environment for running various types of applications (including web applications, enterprise applications, and service-oriented applications) and manage essential services like security, transaction management, threading, and more.

Popular examples of traditional application servers are **IBM WebSphere**, **Oracle WebLogic**, **Red Hat JBoss**, **Apache Tomcat**, and **Microsoft IIS**.

Let's dive deeper into these application servers:

### **1. IBM WebSphere**

**IBM WebSphere** is a suite of middleware products for building and deploying web-based applications. It is widely used in large enterprises and supports Java EE (Enterprise Edition), which includes components like Servlets, JSPs (Java Server Pages), EJBs (Enterprise JavaBeans), and others.

* **Key Features**:
  + Enterprise-class features (scalability, security, etc.).
  + High availability, clustering, and load balancing.
  + Java EE support, including JNDI, JMS, and Web Services.
  + WebSphere Commerce for e-commerce platforms.
  + Integration with IBM tools for monitoring, management, and optimization.
* **Why Use WebSphere?**:
  + WebSphere is ideal for enterprises that require a robust, secure, and scalable platform.
  + It provides sophisticated transaction management, security, and service-oriented architecture (SOA) features, which are crucial in large, complex enterprise environments.

### **2. Oracle WebLogic**

**Oracle WebLogic** is a Java EE application server from Oracle. It is used for deploying and running enterprise applications, especially those that require Java-based technologies.

* **Key Features**:
  + Full support for Java EE standards, including EJB, JPA, JMS, and web services.
  + High availability and clustering.
  + Integration with Oracle’s enterprise products (e.g., Oracle Database).
  + Support for various types of applications: Web applications, Enterprise Applications, and Services.
  + Integrated tools for monitoring, debugging, and managing deployments.
* **Why Use WebLogic?**:
  + WebLogic is best suited for organizations that already rely on Oracle products and need an enterprise-grade application server.
  + It provides comprehensive features for clustering, transaction management, and scalability, making it suitable for mission-critical applications.

### **3. Red Hat JBoss (Now WildFly)**

**Red Hat JBoss**, now rebranded as **WildFly**, is an open-source Java EE application server.

* **Key Features**:
  + Lightweight and modular compared to WebSphere and WebLogic.
  + Open-source, which makes it cost-effective and customizable.
  + Full Java EE support.
  + Integration with Red Hat's ecosystem (e.g., for containerization and orchestration).
  + A microservices-friendly version, which can be run in containers and on cloud environments.
* **Why Use JBoss/WildFly?**:
  + Ideal for organizations that prefer open-source solutions.
  + JBoss is known for flexibility and scalability. It supports both traditional enterprise applications as well as modern microservices-based architectures.
  + It provides an excellent choice for organizations that are seeking a balance between enterprise-class features and cost-effective, open-source solutions.

### **4. Apache Tomcat**

**Apache Tomcat** is an open-source, lightweight web server and servlet container. Unlike the other application servers listed here, Tomcat is specifically designed to handle **servlets** and **JSPs** and does not support the full Java EE stack (such as EJBs or JMS).

* **Key Features**:
  + Lightweight and fast.
  + Primarily used to run Java Servlets and JSP applications.
  + Easy to configure and manage.
  + Ideal for smaller-scale applications that do not require full Java EE support.
* **Why Use Tomcat?**:
  + Tomcat is often chosen for web applications that require **less overhead** than full Java EE application servers like WebSphere or WebLogic.
  + It is suitable for small-to-medium-sized applications or those that rely purely on servlets and JSPs.
  + Cost-effective and open-source, making it a good choice for startups and smaller businesses.

### **5. Microsoft IIS (Internet Information Services)**

**IIS** is a web server from Microsoft used to host .NET applications, web services, and websites. It supports HTTP, HTTPS, FTP, and several other protocols.

* **Key Features**:
  + Supports web applications developed using **ASP.NET**, **PHP**, and other technologies.
  + Deep integration with Microsoft technologies (e.g., SQL Server, Active Directory).
  + Scalability, security, and manageability for enterprise environments.
* **Why Use IIS?**:
  + Ideal for enterprises using Microsoft technologies (Windows Server, .NET, SQL Server).
  + IIS is tightly integrated with the Microsoft ecosystem and provides excellent support for ASP.NET and .NET Core applications.
  + Strong security, ease of integration with other Microsoft services, and performance tuning make IIS popular in enterprise environments.

### **Why to Use These Application Servers?**

* **Deployment of Enterprise Applications**: Traditional application servers are designed to host large-scale, enterprise-grade applications with a variety of features such as security, high availability, clustering, and transaction management. For mission-critical applications, these features are essential.
* **Scalability and Load Balancing**: Application servers such as WebSphere, WebLogic, and JBoss provide features like clustering and load balancing, making it easier to scale applications horizontally. This is vital for businesses with high traffic or complex architectures.
* **Integration with Middleware**: These application servers often come with tools for **enterprise integration**, such as message queues (JMS), object-relational mapping (JPA), and enterprise service buses (ESBs), which simplify communication between distributed applications.
* **Support for Java EE**: If your application is designed to use the **Java EE stack** (e.g., EJB, JPA, JMS, etc.), traditional application servers provide full support for these technologies, ensuring seamless integration.
* **Security and Compliance**: Many of these servers offer enterprise-grade security features, including **SSL/TLS encryption**, **role-based access control**, **authentication mechanisms**, and more, which are essential for enterprise applications dealing with sensitive data.

### **Why Modern Applications (e.g., Spring Boot) Do Not Use Traditional Application Servers**

In recent years, modern application architectures, especially those built using frameworks like **Spring Boot**, **Micronaut**, **Quarkus**, and others, have moved away from traditional, heavyweight application servers. There are several reasons for this shift:

### **1. Lightweight and Self-Contained Applications**

* **Spring Boot** allows developers to create **standalone applications** with an embedded web server (e.g., **Tomcat**, **Jetty**, or **Undertow**), removing the need for a traditional application server like WebLogic or WebSphere.
* These applications are **self-contained** and come with all necessary dependencies, simplifying deployment. This is much simpler than deploying a WAR file to a traditional application server.

### **2. Microservices Architecture**

* Modern applications often follow a **microservices** architecture, where each service is **small**, **independent**, and **deployable** separately. Traditional application servers like WebSphere and WebLogic are **monolithic** and designed to handle large, complex applications with multiple layers.
* Microservices benefit from smaller, **more modular components** that can be deployed independently, often in **containers** (e.g., using **Docker** or **Kubernetes**). This is more flexible and efficient than the centralized approach of traditional application servers.

### **3. Containerization and Cloud-Native Architectures**

* **Containers** and **cloud-native technologies** (e.g., Kubernetes, Docker, and serverless frameworks) have become the default for modern application deployment. Containers provide the **lightweight, portable, and consistent environment** needed for cloud applications.
* Traditional application servers are not well-suited for containerized environments because they require more configuration and overhead. Modern frameworks like **Spring Boot** are designed to be **container-friendly** and can run efficiently in cloud platforms and Kubernetes clusters.

### **4. No Need for Full Java EE Stack**

* **Spring Boot** applications, for example, can run with minimal dependencies. You don't need the full **Java EE** stack (like EJB, JMS, or JPA), which traditional application servers provide. Spring Boot integrates only the specific components you need (e.g., Spring Data, Spring Security, etc.), making it more **flexible** and **lightweight**.
* The modern **Spring Framework** and other similar frameworks focus on simplicity, providing libraries that can be included directly in the application rather than relying on a complex application server.

### **5. Simpler Deployment and Maintenance**

* Deploying a **Spring Boot** application is as simple as running a jar/war file, whereas traditional application servers often require additional configuration (e.g., setting up data sources, security policies, clustering).
* With **Spring Boot**, you get **zero configuration** deployment with auto-configuration and embedded web servers. This **reduces the operational overhead** for enterprises and speeds up development and deployment cycles.

### **Conclusion**

**Traditional application servers** like WebSphere, WebLogic, JBoss, Tomcat, and IIS continue to be essential for large-scale enterprise applications that require robust, scalable, and secure deployment environments. These application servers support a wide range of features such as transaction management, clustering, and full Java EE support, which are crucial for mission-critical applications.

However, **modern applications** like those built with **Spring Boot** have moved away from relying on heavy application servers due to the rise of **microservices architecture**, **containerization**, and **cloud-native approaches**. These modern frameworks provide lightweight, flexible, and efficient deployment options, reducing the need for traditional, monolithic application servers in many enterprise environments.

##### 

# What is the functionality difference between the application servers and nginx, only backend applications need the application servers like tomcat and web logic ? if that's the case where nginx can be used ?

### **Difference Between Application Servers and NGINX**

While **application servers** like **Tomcat**, **WebLogic**, **JBoss**, and **WebSphere** are designed to manage and run application logic (mainly backend), **NGINX** is a **web server** and a **reverse proxy** that primarily deals with HTTP/S traffic management, load balancing, and content serving. They serve different purposes in an architecture, and understanding their specific roles can clarify their use cases and interactions.

### **1. Functionality of Application Servers (e.g., Tomcat, WebLogic, JBoss)**

**Application Servers** are platforms for running business logic and are primarily responsible for:

* **Hosting Backend Applications**:
  + Application servers host and manage **dynamic backend applications**. These could be **Java-based** (e.g., **Java EE** applications), **Python**, **Ruby**, or other language-based applications.
  + Example: **Tomcat** is used for running **Java Servlets** and **JSPs**, while **WebLogic** and **WebSphere** are used for running Java EE applications with full features like **EJB**, **JPA**, **JMS**, and **Web Services**.
* **Business Logic Execution**:
  + They contain the core **business logic**, **data management**, **transaction management**, and **message handling**. For example, EJB (Enterprise Java Beans) in WebLogic or WebSphere are used for enterprise business logic execution.
* **Database Connections**:
  + They manage database connections, **transaction management**, and **persistence**. They are also responsible for **database pooling** (e.g., **JDBC**).
* **Session Management**:
  + Application servers often provide session management and clustering to ensure high availability and load balancing for stateful applications.
* **Security**:
  + They offer security services like **authentication**, **authorization**, and **role-based access control** for web applications and backend services.
* **Integration with Backend Services**:
  + They provide integration with other systems, like **message queues** (JMS), **enterprise services**, **legacy systems**, and **middleware**.
* **Protocol Support**:
  + Application servers typically support **HTTP(S)**, **JMS**, **RMI**, **SOAP**, and other protocols necessary for enterprise communication.

### **2. Functionality of NGINX**

**NGINX** is a high-performance, lightweight web server, reverse proxy, and load balancer. It is primarily focused on the **frontend** (though it can handle some backend tasks) and performs functions such as:

* **Web Server**:
  + **NGINX** serves static content such as HTML files, images, videos, CSS, JavaScript, etc. It is incredibly fast and efficient at serving static files.
  + It also supports serving **dynamic content**, especially when acting as a **reverse proxy** to backend application servers or services (like Tomcat, Node.js, etc.).
* **Reverse Proxy**:
  + NGINX acts as a **reverse proxy** for **backend application servers** (e.g., Tomcat, WebLogic, etc.). This means NGINX can forward HTTP requests to the appropriate backend application server.
  + It can load balance requests across multiple backend servers to distribute traffic evenly and improve scalability and fault tolerance.
* **Load Balancing**:
  + NGINX can distribute client requests to multiple backend servers based on various algorithms (e.g., round-robin, least connections).
  + This helps in improving scalability, fault tolerance, and high availability of web applications.
* **SSL Termination**:
  + NGINX can handle **SSL termination**, meaning it decrypts SSL/TLS traffic and forwards unencrypted requests to the backend servers, offloading the decryption work from application servers.
* **Caching**:
  + NGINX supports caching of static and dynamic content, which can greatly improve response times by reducing load on backend servers.
* **Request Routing**:
  + It can route requests based on the URL, HTTP method, or headers. For instance, certain types of requests (such as those for API endpoints) can be forwarded to specific backend servers.
* **Content Compression and Optimization**:
  + NGINX can compress content using algorithms like **gzip** to reduce response size and improve performance.
* **Security**:
  + NGINX provides basic **rate limiting**, **IP filtering**, and **access controls** to protect web servers and applications from malicious attacks.

### **Key Differences Between Application Servers and NGINX**

| **Feature** | **Application Servers (Tomcat, WebLogic, JBoss, etc.)** | **NGINX** |
| --- | --- | --- |
| **Primary Role** | Host and run backend business logic applications (e.g., Java EE apps) | Serve static content, reverse proxy, load balancer |
| **Content Served** | Backend business logic, APIs, and dynamic content (e.g., JSP, Servlets) | Static content, reverse proxy for dynamic content |
| **Protocol Support** | HTTP(S), JMS, SOAP, EJB, RMI, etc. | HTTP(S), TCP/UDP, WebSocket, SSL, etc. |
| **Session Management** | Yes (especially in Java EE applications) | No direct session management (works with backend) |
| **Scaling & Load Balancing** | Often provides built-in clustering and load balancing | Provides reverse proxy and load balancing |
| **Security** | Advanced security management (e.g., authentication, authorization) | Basic security features (e.g., rate limiting, IP filtering) |
| **Use Case** | Enterprise-grade applications requiring complex backend logic | Web front-end, reverse proxy, and content caching |

### **3. Where NGINX Can Be Used**

NGINX is typically used in frontend scenarios, but it can also be used in various ways to complement backend application servers:

#### **Use Case 1: Reverse Proxy to Application Servers**

NGINX is often placed **in front of** backend application servers like **Tomcat**, **WebLogic**, or **Spring Boot** applications to handle incoming HTTP(S) requests. It will forward the requests to the appropriate backend server.

* **Example**: NGINX can serve as a reverse proxy that routes requests to **Tomcat** for Java-based web apps or to **Node.js** for JavaScript applications.
* **Benefit**: This approach allows NGINX to offload tasks such as SSL termination, request routing, and load balancing while leaving business logic and data management to the application server.

#### **Use Case 2: Load Balancing**

NGINX is frequently used to **load balance** requests across multiple backend servers (application servers or microservices).

* **Example**: If a company runs multiple instances of **Tomcat** to handle a web app, NGINX can balance the load by distributing traffic evenly between the Tomcat instances.
* **Benefit**: This helps scale the application horizontally, distribute traffic, and increase redundancy in case of server failure.

#### **Use Case 3: Caching for Static Content**

NGINX excels at serving **static content** like HTML, CSS, JavaScript, and image files, making it an ideal choice for **content delivery**.

* **Example**: A web application might have static assets (e.g., images or style sheets) that rarely change. NGINX can cache these static assets and serve them directly to clients, reducing the load on the backend servers.
* **Benefit**: Caching static content with NGINX reduces the need for backend application servers to process each request, speeding up response times and reducing the load on backend systems.

#### **Use Case 4: API Gateway**

NGINX can act as an **API Gateway** for microservices, handling **API requests** and forwarding them to the appropriate service backend.

* **Example**: In a microservices architecture, NGINX can route API requests from clients to different microservices based on the URL path or request type (e.g., /user-service, /payment-service).
* **Benefit**: This enables centralized routing, security policies, and logging for API requests.

### **Do Backend Applications Need Application Servers Like Tomcat or WebLogic?**

Not necessarily. The decision to use an application server depends on the complexity and type of application you're building:

* **For Enterprise Applications**: If your application requires **Java EE** features (like EJBs, JMS, JPA) and needs advanced **transaction management**, **clustering**, and **security**, you will likely still need a traditional application server like **WebLogic**, **WebSphere**, or **JBoss**.
* **For Simpler or Modern Applications**:
  + With modern frameworks like **Spring Boot** (or **Node.js** for JavaScript-based applications), you may **not need a traditional application server**. These frameworks provide embedded servers like **Tomcat** or **Jetty**for serving HTTP requests and managing application lifecycle.
  + **Spring Boot**, for example, allows running applications as **self-contained** JAR files with an embedded Tomcat, Jetty, or Undertow server.
* **For Microservices**: Many microservices are deployed in **containers** (using **Docker** or **Kubernetes**) and do not require full-fledged application servers. Instead, they run on lightweight embedded servers or frameworks that manage their own HTTP traffic.

### **Conclusion**

* **Application Servers** (like **Tomcat**, **WebLogic**, and **WebSphere**) are designed to handle **backend business logic**, provide **transaction management**, **clustering**, and **session management**, and host enterprise applications.
* **NGINX**, on the other hand, is a **web server**, **reverse proxy**, and **load balancer** that is primarily used in the **frontend** to manage HTTP(S) traffic, serve static content, provide caching, and load balance requests to backend application servers.
* While backend applications may not always need traditional application servers (depending on the architecture), **NGINX** can still play a critical role in **scaling**, **securing**, and **optimizing** web traffic for modern, cloud-native applications, especially in microservices and containerized environments.

##### 

# diff between the application server and web Server with a real time scenario in an enterprise application ?

### **Difference Between Application Server and Web Server:**

Both **Application Servers** and **Web Servers** are essential components in web application architectures, but they have different roles, capabilities, and functionalities. Let’s explore their differences in more detail and illustrate with a real-time scenario in an enterprise application.

### **1. Definition and Core Functionality:**

#### **Web Server:**

* A **Web Server** is primarily responsible for handling **HTTP requests** from clients (like browsers), serving **static content** (e.g., HTML, CSS, JavaScript, images), and forwarding requests for dynamic content to the backend server (or application server).
* It mainly handles **client-side requests** and serves **static resources** without much processing.

**Core Functions**:

* Serve **static content** (HTML, CSS, JavaScript, images, etc.).
* Handle **HTTP requests**.
* Route requests for dynamic content (via CGI, PHP, etc.) to an application server or backend service.
* Provide **basic security** features (e.g., SSL/TLS termination, access control).

#### **Application Server:**

* An **Application Server** provides a platform for running **backend applications**. It executes **business logic**, processes requests, handles **database interactions**, manages transactions, and generates **dynamic content**.
* It can host complex business applications, often using **enterprise frameworks** like **Java EE**, and provide services such as **session management**, **transaction management**, and **messaging**.

**Core Functions**:

* Run **business logic** (dynamic content generation).
* Provide **transaction management**, **security**, and **application-level services**.
* Manage **database connections** and **session persistence**.
* Host **enterprise applications** (e.g., Java EE applications, Spring-based applications).

### **2. Key Differences:**

| **Feature** | **Web Server** | **Application Server** |
| --- | --- | --- |
| **Primary Role** | Serve static content and handle HTTP requests | Host business logic and dynamic content |
| **Content Served** | Static content (HTML, images, CSS, JS) | Dynamic content generated by business logic |
| **Dynamic Content Handling** | Forwards dynamic content requests to application server | Generates dynamic content on its own |
| **Protocols Supported** | HTTP, HTTPS (for secure connections) | HTTP, HTTPS, JMS, RMI, EJB, SOAP, etc. |
| **Session Management** | Limited (basic session handling) | Advanced session management and clustering |
| **Transaction Management** | No transaction management | Yes, with full transaction management (e.g., **JTA**) |
| **Database Integration** | Limited (via CGI, PHP, etc.) | Full database interaction and connection pooling |
| **Example Technologies** | Apache HTTP Server, NGINX, Microsoft IIS | Tomcat, JBoss, WebLogic, WebSphere, WildFly |

### **3. Real-Time Scenario in an Enterprise Application:**

Let’s consider a **real-time enterprise web application** for an **online retail store** with various components and services, such as an **e-commerce platform**, **user management**, **product catalogs**, and **order processing**.

#### **Architecture:**

* **Web Server**: NGINX or Apache HTTP Server
* **Application Server**: Tomcat, WebLogic, JBoss, or Spring Boot

#### **Flow of a Request in the Application:**

1. **Client Request**:
   * A user in the online store enters the website (e.g., https://www.store.com) and requests to view a product (e.g., https://www.store.com/product/123).
   * The **browser** (client) sends an HTTP request to the **web server** (e.g., **NGINX**).
2. **Web Server Handling**:
   * The **web server** receives the HTTP request for the /product/123 URL.
   * If the request is for static content (e.g., a product image or a CSS file), the **web server** immediately serves it.
   * If the request requires **dynamic content** (e.g., fetching product details from the database), the **web server**forwards the request to the **application server**.
3. **Application Server Handling**:
   * The **application server** (e.g., **Tomcat** or **Spring Boot** with an embedded server) takes the request and invokes business logic to retrieve product details from the **database**.
   * The **application server** may involve **transaction management** (e.g., ensuring the consistency of product stock levels), **security checks** (e.g., user authentication and authorization), and **API calls** (e.g., connecting to an external payment gateway).
   * The application server generates **dynamic content** (e.g., HTML page with product details, pricing, availability).
4. **Response Handling**:
   * Once the **application server** generates the dynamic content, it sends it back to the **web server**.
   * The **web server** then sends the final **HTML response** to the client browser, which is rendered to the user.
5. **Session Management**:
   * The **application server** might manage the **user’s session** (e.g., tracking cart items or user login state).
   * The **web server** may use **cookies** to maintain session IDs, but the actual session data and logic reside in the **application server**.
6. **Database Interaction**:
   * If the product request involves fetching data from a database, the **application server** (e.g., via **JDBC** or **JPA**) interacts with the database to retrieve product details, availability, and pricing.
   * The **application server** handles **transactions**, ensuring that the database interaction is consistent and follows ACID principles (Atomicity, Consistency, Isolation, Durability).

### **4. When Do You Use Web Servers and Application Servers in the Enterprise?**

In an enterprise environment, both **web servers** and **application servers** are often used together to create a scalable, efficient architecture.

#### **When to Use a Web Server:**

* **Serving Static Content**: A **web server** like NGINX or Apache HTTP Server is well-suited for serving **static content** (e.g., images, CSS, JavaScript).
* **Load Balancing**: Web servers can act as a **reverse proxy** to forward requests to application servers and distribute the load.
* **SSL Termination**: Web servers can handle **SSL/TLS termination**, offloading this resource-intensive task from application servers.
* **Caching**: Web servers can cache frequently accessed static content, reducing the load on backend servers.

#### **When to Use an Application Server:**

* **Running Business Logic**: An **application server** is required for executing complex backend business logic, such as handling user authentication, processing orders, interacting with databases, and managing transactions.
* **Enterprise Application Features**: If the application requires **enterprise features** like **JMS** (Java Message Service), **EJBs** (Enterprise Java Beans), **JPA** (Java Persistence API), and **web services**, you need a robust **application server**.
* **Session Management**: If the application involves complex session handling (e.g., user shopping carts, stateful sessions), the application server manages these sessions more effectively.
* **Enterprise Integration**: An application server integrates various systems like **message queues**, **transactional services**, and **legacy systems** into a cohesive application.

### **5. Example Use Case in a Large E-commerce System:**

Let’s say the e-commerce platform has the following requirements:

* **Frontend (Web Server)**: Users access the site and browse products, view their cart, and complete orders. The web server (NGINX) serves static content such as images, CSS, and JavaScript, and also acts as a **reverse proxy** to forward requests to the application server.
* **Backend (Application Server)**: The **application server** (e.g., **Tomcat** or **Spring Boot**) handles business logic such as:
  + Processing orders.
  + Managing user sessions (login, shopping cart).
  + Interfacing with databases to fetch product information.
  + Handling transactions (e.g., payment gateway integration).
  + Generating dynamic content (e.g., product pages).

In this scenario:

* **NGINX** is configured to handle static requests and balance the load across multiple backend **application servers**(e.g., Tomcat or Spring Boot).
* The **application servers** are responsible for processing the requests, executing business logic, accessing the database, and generating dynamic content.

### **Conclusion:**

* **Web Servers** (e.g., NGINX, Apache HTTP Server) are responsible for handling **HTTP requests**, serving **static content**, and forwarding dynamic requests to the application server. They act as **gateways** between the user and the backend services.
* **Application Servers** (e.g., Tomcat, WebLogic, JBoss) handle **backend business logic**, **database interactions**, **session management**, and **enterprise-level functionality** like transactions and security. They are used to run **dynamic applications** that require complex processing.

In an enterprise application, both servers are typically used together to create a **scalable** and **efficient** architecture, where the web server handles the frontend interactions, and the application server takes care of backend processing and business logic.

# Monitoring vs. observability: What’s the difference between monitoring and observability?

Is observability really monitoring by another name? In short, no. While observability and monitoring are related—and can complement one another—they are actually different concepts.

### **Monitoring**

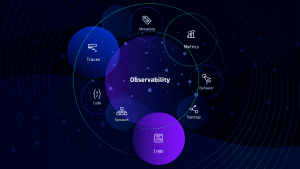
In a monitoring scenario, you typically preconfigure dashboards to alert you about performance issues you expect to see later. However, these dashboards rely on the key assumption that you’re able to predict what kinds of problems you’ll encounter before they occur.

Cloud-native environments don’t lend themselves well to this type of monitoring because they are dynamic and complex, which means you have no way of knowing in advance what kinds of problems might arise.

### **Observability**

In an observability scenario, where teams have fully instrumented an environment to provide complete observability data, you can flexibly explore what’s going on and quickly figure out the root cause of issues you may not have been able to anticipate.

Traditionally, the industry defines observability as logs, metrics, and traces. In more complex cloud environments, however, observability must encompass more, including metadata, user behavior, topology and network mapping, and access to code-level details.



Observability pillars include logs, metrics, and traces. Modern observability also includes metadata, user behavior, topology and network mapping, and code-level details.

# Why is observability important?

In enterprise environments, observability helps cross-functional teams understand and answer specific questions about what’s happening in highly distributed systems. Observability enables you to understand what is slow or broken and what you need to do to improve performance. With an observability solution in place, teams can receive alerts about issues and proactively resolve them before they impact users.

# Understanding “unknown unknowns”

Because modern cloud environments are dynamic and constantly changing in scale and complexity, teams neither know about nor can monitor most problems. Observability addresses this common issue of “unknown unknowns,” enabling you to continuously and automatically understand new types of problems as they arise.

# Backup:

We create the backup policies in the parent account which is the control tower account and those backup policies to OU’s. so that the backup policies get inherited to all the member accounts.

Deploy the member account resources like vault, vault key, iam role and kms key in the member accounts using stackset in the payer account.

In the member account all users need to do is attach the tag/key values to resources they want backup. Once tags are added aws backup will take the backup of resources and then store them in the vault in the same account.

The tag/key values be added to the resources:

| **Policy / Plan Name** | **Tag Name** | **Value** | **Schedule** | **Remarks** | **Retention in US-West 2 (Oregon)** | **Copied and Retented in US-East (N. Virginia)** |
| --- | --- | --- | --- | --- | --- | --- |
| Dish-wl-DB-Daily-p | Backup-Plan | DB-Default | Every 12 Hours | Production Database Point in Time Recovery (PITR) backups | 35 Days | No |
| Dish-wl-EC2-Daily-p | Backup-Plan | EC2-Default | Daily(10PM) | Production EC2 instance backups | 7 Days | No |
| Dish-wl-EBS-Daily-p | Backup-Plan | EBS-Default | Daily(10PM) | Production Stand Alone EBS Volume backups | 7 Days | No |
| Dish-wl-EFS -Daily-p | Backup-Plan | EFS-Deault | Daily(10PM) | Production EFS volume backups | 7 Days | No |
| Dish-wl-FSX-Daily-p | Backup-Plan | FSX-Default | Daily(10PM) | FSX File Systems | 7 Days | No |
| Dish-wl-S3-Daily-p | Backup-Plan | S3-Default | Daily(10PM) | S3 Bucket backups | 7 Days | No |
| Dish-wl-EC2-Daily-1yr-p | Backup-Plan | Dish-wl-EC2-Daily-1yr-p | Daily(10PM) | Production EC2 instance backups | 7 Days | 1 Year |

### **Give the implementation process to implement the aws backup solution in control tower environment from creating the backup policies in control tower, attaching them to OU and in the member account create a vault and vault key to protect the snapshots. aws backup will create the backup snapshots and store them in the vault ?**

Implementing an AWS Backup solution in a Control Tower environment involves a series of steps to ensure backup policies are created, applied, and managed properly across accounts and organizational units (OUs) while ensuring snapshots are stored in a secure vault with the appropriate encryption. Below is a step-by-step guide on how to implement AWS Backup in an AWS Control Tower environment.

### Overview

1. Create Backup Policies in AWS Control Tower: These policies define what to back up and when, across all accounts in your AWS organization.
2. Attach Backup Policies to Organizational Units (OUs): Apply these policies to specific OUs or individual AWS accounts within Control Tower.
3. Create a Backup Vault in Member Accounts: Each account should have a designated backup vault to store the backup snapshots securely.
4. Create and Protect Backup Vault with a Vault Key: Ensure the backup vault is encrypted using a custom AWS KMS key.
5. Configure AWS Backup to Create Snapshots and Store in the Vault: Set AWS Backup to manage and create backups and store them in the vault.

### Prerequisites

* AWS Control Tower environment set up.
* Access to the AWS Backup service.
* Permissions to create policies, vaults, and KMS keys in AWS accounts.
* Ensure you are operating in the Control Tower management account and have delegated permissions to the member accounts.

### Step 1: Create Backup Policies in AWS Control Tower

AWS Control Tower provides a managed service for governing your AWS environment, including backup policies. You need to create backup policies through AWS Backup and then attach them to OUs.

1. Sign in to the AWS Management Console as an admin user in the Control Tower management account.
2. Navigate to the AWS Backup console.
3. On the left-hand side, under Backup plans, click Create backup plan.
4. Choose a Backup plan template:
   * Select from predefined templates such as Daily backup, Weekly backup, etc., or create a custom plan.
5. Configure the backup plan:
   * Name the backup plan.
   * Specify Backup frequency (e.g., daily, weekly).
   * Set Retention period (e.g., 30 days, 90 days).
   * Choose the backup vault (you can create a vault in Step 3).
6. Define Resources:
   * Add resources to the backup plan by either using tags (e.g., tag-based backup selection) or directly specifying the AWS resources (e.g., EC2 instances, EBS volumes).
7. Save the Backup Plan.
8. Once the plan is created, you can attach the backup plan to specific organizational units (OUs) in Control Tower:
   * From the Backup Plans screen, select your newly created backup plan.
   * Under Assign backup plan, choose Attach to organizational unit and select the OUs you want to apply the policy to.
   * AWS Control Tower will ensure that the policy is enforced across all member accounts in the selected OUs.

### Step 2: Attach Backup Policies to Organizational Units (OUs)

1. Navigate to the AWS Control Tower console.
2. On the left panel, click Organizational Units (OUs).
3. Select the OU that you want to apply the backup policy to.
4. Under the Backup section (if integrated), assign the relevant backup policy.
5. If the OU is already linked to the AWS Backup service, the policy you created in Step 1 will automatically be applied to all member accounts under this OU.
6. Ensure that the member accounts (which will be using the backup policy) are onboarded to Control Tower and have necessary permissions.

### Step 3: Create a Backup Vault in Member Accounts

Each member account needs a dedicated Backup Vault to store the backup data.

1. Sign in to the AWS Management Console for one of your member accounts.
2. Open the AWS Backup console.
3. In the Backup Vaults section, click Create Backup Vault.
4. Enter the following details:
   * Name: Provide a name for the vault (e.g., MyBackupVault).
   * Encryption: Choose AWS Key Management Service (KMS) for encryption. You can use the default KMS key or create a custom key (discussed below).
5. Click Create to create the vault.

### Step 4: Create a Vault Key to Protect the Snapshots (Using KMS)

To ensure that your backup data is encrypted and properly secured, create an AWS KMS key and associate it with the backup vault.

1. Create a Custom KMS Key:
   * Navigate to the KMS console in your member account.
   * Click Create key and choose Symmetric encryption.
   * Provide a Key alias and Key description.
   * Set the Key policy to allow AWS Backup and the necessary accounts to use the key for encryption.
2. Attach the Key to the Backup Vault:
   * In the AWS Backup console, go to Backup Vaults and select the vault created in Step 3.
   * Under Encryption settings, click Edit and choose your newly created KMS key from the list.
   * This key will be used to protect all backups stored in this vault.
3. Optionally, you can create IAM policies to control which users and services can access the vault and KMS key.

### Step 5: Configure AWS Backup to Create Snapshots and Store in the Vault

Now that the backup policy, vault, and KMS key are configured, you can ensure that AWS Backup automatically creates and stores backup snapshots in the vault.

1. Assign Backup Plan to Resources:
   * In the AWS Backup console, go to Backup Plans.
   * Choose your backup plan.
   * Under Assign resources, define the resources that should be backed up (e.g., EC2 instances, EBS volumes).
   * Add tags to these resources if you're using tag-based resource selection.
2. Schedule the Backups:
   * AWS Backup will automatically schedule backups based on the frequency and retention settings you defined earlier.
   * AWS Backup will start creating snapshots of your EC2 instances, RDS databases, and other resources based on the policy.
3. Verify the Backups:
   * Go to the Backup Vault to verify that backups are being stored properly.
   * You should see the snapshots being listed under the respective vault.
   * You can check the backup status in the AWS Backup Console under Backup Jobs to ensure everything is functioning as expected.

### Step 6: Monitor and Audit Backups

After backup policies are applied, you can monitor the backup jobs and ensure that they are being completed successfully.

1. Monitoring Backup Jobs:
   * Use AWS Backup Dashboard to monitor backup job status (e.g., success, failure).
   * Check the Backup Job Logs to troubleshoot issues if any backups fail.
2. Audit with AWS CloudTrail:
   * Use AWS CloudTrail to track who created or modified backup plans, vaults, and jobs.
   * Ensure compliance by monitoring backup events.
3. Set Up Notifications:
   * You can use Amazon SNS to set up notifications for backup job status (success or failure).

### Step 7: Backup Retention and Cleanup

1. Define Retention Periods: As part of the backup policy, define the retention period (e.g., 30 days, 90 days) for backup snapshots.
2. Automated Cleanup: AWS Backup will automatically delete old backups that exceed the retention period, ensuring that your storage costs are optimized.
3. Manual Cleanup: If necessary, you can manually delete backups from the vault that are no longer needed.

### Conclusion

By following these steps, you will have successfully implemented AWS Backup in an AWS Control Tower environment, ensuring that backup policies are applied at the Organizational Unit (OU) level, backups are securely stored in encrypted vaults, and backup snapshots are created and protected with KMS encryption.

This solution helps ensure consistent backup policies across all AWS accounts in the Control Tower environment, providing the organization with reliable, automated backup and recovery solutions while maintaining security and compliance.

##### 

# Service Catalog:

**enable the Shared Services account as the delegated administrator for AWS Service Catalog, you can run the following command:**

* **aws organizations register-delegated-administrator --account-id <YOUR\_AWS\_ACCOUNT\_ID> --service-principal servicecatalog.amazonaws.com**

### We have the code commit, code build and code pipeline process for our delegate account. We are adding new products or portfolios to the service catalog.

We have 3 portfolios in our environment which are:

Network services: all network related products

Vpc - 24

Vpc - 22

Additionalvpc - 22

Additional vpc - 24

Reference Architecture: Apart from networks and Iam related resources are this.

Ec2,

ecs,

eks,

autoscaling,

loadbalancers,

s3,

cloudfront

Database Services:

RDS, DynamoDB, Postgres, DocumentDB

Each portfolio contains the products related to them.

We share the portfolios with aws control tower organization id, so that all the products under portfolios can be shared across the accounts.

### Give the implementation process to implement the aws service catalog in the control tower environment and how can we share the products and portfolios to member accounts under control tower ?

**Implementing AWS Service Catalog in an AWS Control Tower environment involves creating and managing portfolios and products that can be shared across organizational units (OUs) and member accounts. AWS Service Catalog helps manage and govern the deployment of AWS resources by allowing you to create and manage collections of AWS resources, called products, which can be shared with other AWS accounts.**

**The process for integrating AWS Service Catalog with AWS Control Tower involves several steps: setting up AWS Service Catalog, creating portfolios and products, and then sharing those products and portfolios with member accounts under Control Tower.**

### **Overview of Key Concepts**

1. **Portfolio: A container for organizing and managing collections of AWS Service Catalog products.**
2. **Product: A versioned collection of AWS resources defined in Service Catalog, such as EC2 instances, VPCs, or CloudFormation templates.**
3. **Constraints: Rules that control how a product can be provisioned (e.g., IAM roles, permissions, or limits on resource configuration).**
4. **Shared Portfolio: AWS Service Catalog allows sharing portfolios with other accounts or organizational units (OUs) within the same AWS Organization. This feature is useful for managing centralized product catalogs.**

### **Prerequisites**

* **AWS Control Tower environment set up.**
* **Access to AWS Service Catalog and permissions to manage it (admin permissions).**
* **AWS Organization enabled in your AWS Control Tower environment (Service Catalog leverages AWS Organizations to manage account boundaries).**
* **Ensure that all accounts are onboarded to Control Tower and have necessary IAM permissions to access and provision Service Catalog products.**

### **Step 1: Set Up AWS Service Catalog in the Control Tower Environment**

#### **1.1 Access AWS Service Catalog**

* **Sign in to the AWS Management Console as an admin user in the AWS Control Tower management account.**
* **Navigate to the AWS Service Catalog console.**

#### **1.2 Configure AWS Service Catalog (if not already configured)**

* **AWS Control Tower automatically provisions Service Catalog and integrates it with AWS Organizations, but you may need to configure specific settings to align with your organization’s policies.**

1. **In the AWS Service Catalog Console, click on Administration and select Organizations.**
2. **Ensure that Service Catalog is enabled for the AWS Organization. This allows you to share products and portfolios across different accounts and OUs within Control Tower.**

### **Step 2: Create a Portfolio in AWS Service Catalog**

**A portfolio is used to organize and manage a collection of products (e.g., EC2 instances, RDS databases, or VPC configurations). You can create portfolios and associate them with your AWS accounts under AWS Control Tower.**

#### **2.1 Create Portfolio**

1. **Navigate to AWS Service Catalog.**
2. **Under Portfolios, click Create portfolio.**
3. **Enter the following details:**
   * **Portfolio name: Choose a name that reflects the purpose of the portfolio (e.g., SharedProductsPortfolio).**
   * **Description: Provide a brief description (e.g., Portfolio for shared resources).**
   * **Owner: Specify the owner or administrative contact for the portfolio.**
4. **Click Create to create the portfolio.**

#### **2.2 Create Product(s) and Associate with Portfolio**

1. **Create Product:**
   * **After creating the portfolio, you can create a product. A product is a collection of AWS resources (CloudFormation templates, AMIs, etc.).**
   * **Click Create product.**
   * **Provide the product name, description, and CloudFormation template (or specify existing AWS resources like AMIs, or EC2 instances).**
   * **Choose any constraints (e.g., limits on resource sizes, user permissions).**
2. **Associate Product with Portfolio:**
   * **Under the portfolio, click Add product.**
   * **Select the product you just created and click Add.**

#### **2.3 Set Constraints (Optional)**

* **AWS Service Catalog allows you to set constraints for your products to control how they are provisioned. Constraints can include:**
  + **IAM roles**
  + **Allowed configurations**
  + **AWS resource limits**

**You can create and attach constraints to the portfolio or product based on the organization's security or compliance requirements.**

### **Step 3: Share the Portfolio with Member Accounts under AWS Control Tower**

**Once you have a portfolio with products, you can share the portfolio with specific AWS accounts, organizational units (OUs), or all accounts within your AWS Organization.**

#### **3.1 Enable Portfolio Sharing**

1. **In the Service Catalog Console, go to the Portfolio you want to share.**
2. **Click on the Share button under the Actions dropdown.**
3. **You’ll be prompted to share the portfolio with one or more AWS accounts or OUs. You have the following options:**
   * **Share with specific AWS accounts: You can manually add member accounts (for example, by using the AWS Account ID).**
   * **Share with OUs: Share the portfolio with all accounts within a specific organizational unit (OU). This is useful when you want to provide access to a set of accounts that belong to a specific business unit or environment (e.g., dev, staging, production).**
   * **Share with all accounts in AWS Organization: You can choose to make the portfolio available to all accounts within your AWS Organization.**
4. **Select the desired accounts or OUs to share the portfolio.**
5. **Set permissions for the shared accounts (e.g., Admin, PowerUser, or User). This defines what actions the accounts can take on the portfolio (e.g., create products, view products, or manage resources).**

#### **3.2 Share the Portfolio to an OU (Control Tower Integration)**

**To share the portfolio with OUs within Control Tower:**

1. **When selecting the Share with OUs option, you can choose from organizational units (OUs) that are part of your Control Tower landing zone.**
   * **OU selection: Ensure that your OUs (e.g., SharedServices, Dev, Prod) are available as part of the organizational structure in Control Tower.**
   * **Once selected, the products and portfolios will be made available to all accounts under that OU, providing centralized access to Service Catalog products.**
2. **Click Share to initiate the sharing process.**

#### **3.3 Cross-Account Access Control (Optional)**

**You can use IAM policies to manage fine-grained access to Service Catalog resources in member accounts. You can specify who can launch or view the products shared across accounts.**

* **Ensure that accounts have appropriate IAM roles that allow them to access the shared portfolio.**
* **Example IAM policy to allow users in a member account to provision a Service Catalog product:**

**json**

**Copy code**

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Effect": "Allow",**

**"Action": [**

**"servicecatalog:SearchProductsAsAdmin",**

**"servicecatalog:ProvisionProduct"**

**],**

**"Resource": "\*"**

**}**

**]**

**}**

### **Step 4: Provision Products from Shared Portfolios**

**Once the portfolio is shared with member accounts or OUs, users in those accounts can provision products from the Service Catalog:**

1. **Sign in to the member account that has access to the shared portfolio.**
2. **Navigate to the AWS Service Catalog Console.**
3. **Go to Portfolios and find the shared portfolio.**
4. **Select a product from the portfolio and click Launch to provision it.**
5. **The product will be provisioned based on the CloudFormation templates or AWS resources defined in the product.**

### **Step 5: Monitor and Manage Products in the Shared Portfolio**

**You can monitor product provisioning and use AWS Service Catalog’s reporting tools to track the usage and status of products provisioned in member accounts. You can also revoke access to portfolios as necessary by managing sharing settings.**

### **Conclusion**

**By following the steps outlined above, you will have successfully implemented AWS Service Catalog in an AWS Control Tower environment. You can now:**

* **Create and manage portfolios containing predefined products (such as EC2 instances, RDS databases, and other resources).**
* **Share portfolios with specific AWS accounts or organizational units (OUs) within Control Tower.**
* **Monitor product provisioning and manage access control via IAM policies.**

**This integration allows for consistent governance, streamlined resource provisioning, and compliance within a Control Tower environment while leveraging the power of AWS Service Catalog to standardize and manage AWS resource deployments.**

# PrivateCA:

Instead of creating private certificate authority in all the member account of control tower, we created the private certificate authorities for prod and nonprod account in one of the shared service account and shared them with aws RAM to all the OU’s organization units.

While creating the private CA

This way we achieved the cost optimization as each private certificate authority cost is 400$ per month.

Private CA created are subdomains for our root domain

Ex: example.com is the root domain

Saishiva.example.com - private created one for prod and nonprod

So all the member accounts will create the certificates with this private CA and the private certificates are auto approved every year.

# Private Marketplace:

Notes:

Go daddy, amazon , azure, gcp , domain.com , bluehost

.edu, org, .edu, .net ., com , in

example.com - primary domain

[www.example.com](http://www.example.com) - subdomain

[www.saishiva.example.com](http://www.saishiva.example.com) - subdomain

Ssl hand shaking

Authoritative servers - telephone directory

Recursive servers

Name servers

Resolvers