**DevOps**

[**Linux and Shell scripting**](#Linux)

1. [Basic Linux Commands](#Linux_Basic_Commands)

[**Python**](#PYTHON)

1. [Shell Vs Python](#PYTHON_Shell_Vs_Python)
2. [Keywords, Identifiers, Statements, Variables, Constants](#PYTHON_Keywords_identifiers_statements)
3. [Name space and Scope](#PYTHON_NameSpace)
4. [Input & Output](#PYTHON_Input_and_OutPut)
5. [Operators](#PYTHON_Operators)
6. [Data Types](#PYTHON_Data_types)

[**AWS for DevOps**](#AWS_FOR_DEVOPS)**:**

1. [AWS IAM (Identity and Access Management)](#AWS_IAM)
2. [AWS EC2(Elastic compute cloud)](#AWS_EC2)
3. [VPC](#AWS_VPC)
4. [VPC(Deep Dive Notes)](#AWS_VPC_DEEP_DIVE)
5. [AWS Security Groups & NACLs](#AWS_SECURITY_GROUP_NACLs)
6. [Route53](#AWS_Route53)
7. [AWS S3[Simple Storage Service]](#AWS_S3)
8. [AWS CLI [Command Line Interface]](#AWS_CLI)
9. [AWS CFT [CloudFormation Template]](#AWS_CFT)
10. [AWS CodeCommit](#AWS_CodeCommit)
11. [AWS Code Pipeline](#AWS_CodePipeline)
12. [AWS CloudWatch](#AWS_CloudWatch)
13. [AWS LAMBDA](#AWS_LAMBDA)
14. [AWS LAMBDA – CLOUD COST OPTIMIZATION DEMO](#AWS_LAMBDA_COST_OPTIMIZATION_DEMO)
15. [AWS LAMBDA DEVOPS COMMON REAL-TIME ACTIVITIES WITH EXAMPLES](#AWS_LAMBDA_DEVOPS_ACTIVITIES)
16. [AWS CloudFront](#AWS_CloudFront)
17. [AWS ECR [Elastic Container Registry]](#AWS_ECR)
18. [AWS ECS [Elastic Container Service]](#AWS_ECS)
19. AWS EKS [Elastic Kubernetes Service]
20. [AWS Secrets Management Techniques](#AWS_SecretManagement)
21. [AWS Config](#AWS_Config)
22. [AWS Load Balancers](#AWS_Load_Balancers)
23. [Migrating applications to AWS Cloud](#AWS_Migration)
24. [AWS 3-Tire Architecture](#AWS_3Tire_architecture)

[**Kubernetes**](#Kubernetes)

1. [Why Kubernetes?](#WhyKubernetes)
2. [Kubernetes architecture?](#KubernetesArchitecture)
3. [Popular Kubernetes Distributions](#PopularKubernetesDistributions)
4. [Organizations preferences on using Kubernetes and Kubernetes distribution.](#Org_preferrences_k8s)
5. [How and what the DevOps engineers do with Kubernetes in production in organization(Life cycle)](#HowandwhattheDevOpsengineersdo)
6. [Kubernetes Installation Using KOPS on EC2](#K8sInstallation)
7. [What is Pod?](#WhatisPod)
8. [Multi-Container Pod Design Patterns in Kubernetes](#k8s_Multi_Container_PodDesign)
9. [What is Deployment? And Deployment vs ReplicaSet vs Pod](#k8s_Deployment_ReplicaSet_Pod)
10. [What is service? And problems without service and Solutions with service?](#k8s_services)
11. [Step by Step procedure on how the service(ClusterIP, NodePort, LoadBalancer) will be created?](#k8s_Service_creations)
12. [What is ingress? Services with and without ingress?](#k8s_ingress)
13. [INGRESS TLS methods(Secure ingress)](#k8s_ingress_TLS)
14. [RBAC(Role based access control)](#k8s_RBAC)
15. [Custom Resources | Custom Controllers | Custom resource Definition](#k8s_custom_resources)
16. [CONFIGMAPS & SECRETS](#k8s_CONFIGMAPS_SECRETS)
17. [MONITORING USING PROMETHEUS & GRAFANA](#k8s_MONITORING)
18. [STEP BY STEP SET-UP OF PROMETHEUS AND GRAFANA](#k8s_MONITORING_STEP_BY_STEP)
19. [Daemon and DaemonSet Concepts in Kubernetes](#k8s_DaemonSet)

[**Kubernetes Troubleshooting**](#k8s_Trouble)

1. [Image pull backoff](#k8s_Trouble_ImagePullBack)
2. [CrashLoop BackOff](#k8s_Trouble_CrashLoop_BackOff)
3. [Issues with Scheduler](#k8s_Trouble_Scheduler)
4. [Realtime Issues with Stateful Sets and Persistent Volumes in Kubernetes](#k8s_Trouble_StatefulSets)
5. [Real-time Security Access Issue in DevOps with Kubernetes Network Policies](#k8s_Trouble_Network_Policies)
6. [Kubernetes Real-Time challenges in production](#k8s_Trouble_RealTime_Challenges)

[**Github Actions**](#Github_Actions)

**LINUX & SHELL SCRIPTING**

**1.** **Basic Linux Commands**

* **ls -ltr**: To list all the files and directories along with the details
* **pwd**: present working directory
* **cd**: change directory
* **touch**: to create a file / Change file time stamps
* **mkdir**: to create a directory
* **nproc**: to know the number of CPUs
* **df –h**: To know the disk size
* **free**: To know the memory(similar to df –h)
* **top**: It will give the complete information regarding disk and processors running
* **man <any linux command>**: Provides the details of the command
* **chmod**: To grant permissions to files and folders
* **sed**:is a stream editor used for performing basic text transformations on an input stream, such as searching, find and replace, insertion, or deletion.
* **Wget:** is a utility used for non-interactive downloading of files from the web.
* **Curl**: The "curl" command in Linux is a tool used to transfer data from or to a server using various protocols such as HTTP, HTTPS, FTP, and others.
* **rm**: Remove files or directories.
* **rmdir**: Remove empty directories.
* **mv**: Move or rename files or directories.
* **chown**: Change file owner and group.
* **ln:** Create hard and symbolic links.
* **less**: View file content interactively.
* **head**: Display the beginning of a file.
* **tail**: Display the end of a file.
* **grep**: Search text using patterns.
* **find**: Search for files in a directory hierarchy.
* **locate**: Find files by name.
* **ps**: Report a snapshot of current processes.
* **kill**: Send signals to processes.
* **tar**: Archive files.
* **gzip**: Compress files.
* **gunzip**: Decompress files.
* **zip**: Package and compress files.
* **unzip**: Extract compressed files.
* **scp**: Secure copy (remote file copy program).
* **ssh**: OpenSSH SSH client (remote login program).
* **rsync**: Remote file and directory synchronization.
* **iptables**: Administration tool for IPv4 packet filtering and NAT.
* **systemctl**: Control the systemd system and service manager.
* **journalctl**: Query and display messages from the journal.
* **crontab**: Schedule periodic background tasks.
* **env**: Display or set environment variables.
* **export**: Set an environment variable.
* **alias**: Create an alias for a command.
* **nano**: Simple text editor.
* **vi**/**vim**: Advanced text editor.
* **docker**: Manage Docker containers.
* **kubectl**: Control Kubernetes clusters.
* **helm**: Manage Kubernetes applications.
* **aws**: AWS Command Line Interface.
* **terraform**: Infrastructure as Code tool.
* **ansible**: Automation tool for configuration management.
* **git**: Version control system command-line tool.
* **Free –g**: displays the amount of free and used memory in the system, with the memory sizes shown in gigabytes.

**Python For DevOps**

**1.** **Shell Vs Python**

Certainly! The choice between using shell scripting and Python in DevOps depends on the specific task or problem you're trying to solve. Both have their strengths and are suitable for different scenarios. Here are some guidelines to help you decide when to use each:

**Use Shell Scripting When:**

1. **System Administration Tasks:** Shell scripting is excellent for automating routine system administration tasks like managing files, directories, and processes. You can use shell scripts for tasks like starting/stopping services, managing users, and basic file manipulation. Works in linux environments.
2. **Command Line Interactions:** If your task primarily involves running command line tools and utilities, shell scripting can be more efficient. It's easy to call and control these utilities from a shell script.
3. **Rapid Prototyping:** If you need to quickly prototype a solution or perform one-off tasks, shell scripting is usually faster to write and execute. It's great for ad-hoc tasks.
4. **Text Processing:** Shell scripting is well-suited for tasks that involve text manipulation, such as parsing log files, searching and replacing text, or extracting data from text-based sources.
5. **Environment Variables and Configuration:** Shell scripts are useful for managing environment variables and configuring your system.

**Use Python When:**

1. **Complex Logic:** Python is a full-fledged programming language and is well-suited for tasks that involve complex logic, data structures, and algorithms. If your task requires extensive data manipulation, Python can be a more powerful choice.
2. **Cross-Platform Compatibility:** Python is more platform-independent than shell scripting, making it a better choice for tasks that need to run on different operating systems. (Both windows and linux)
3. **API Integration:** Python has extensive libraries and modules for interacting with APIs, databases, and web services. If your task involves working with APIs, Python may be a better choice.
4. **Reusable Code:** If you plan to reuse your code or build larger applications, Python's structure and modularity make it easier to manage and maintain.
5. **Error Handling:** Python provides better error handling and debugging capabilities, which can be valuable in DevOps where reliability is crucial.
6. **Advanced Data Processing:** If your task involves advanced data processing, data analysis, or machine learning, Python's rich ecosystem of libraries (e.g., Pandas, NumPy, SciPy) makes it a more suitable choice.

Note: We can even use Ansible for both windows and linux environments to execute the scripts.

**2.** **Keywords, Identifiers, Statements, Variables, Constants**

**Keywords**:

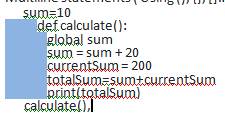
1. Keywords are the words those are reserved. There are 35 keywords those are reserved. **Ex: if, and, or, in…etc**
2. We can’t use these reserved keywords as variables, classes, functions…etc
3. Keywords are case sensitive.

**Identifiers**:

1. Identifier is the name given to identify a variable, class, function…etc
2. We shouldn’t use special symbols(@#$) in identifiers and it can’t starts with digits

**Statements**:

1. Any executable instruction that tells computer to perform a specific task is called statement
2. Single line statements
3. Multiline statements ( Using (), {}, []….in python we will use indentation



**Variables**:

1. Variables are used to store data
2. When you declare a variable, it occupy some data
3. Can be changed at any time

Declaring a Variable:

In other programs like java, we use data types to declare a variable type

Ex: *int a = 10*

*Float percentage = 85.5*

*String name = “Sarath”*

But, In python we don’t need to specify since python is dynamically programmed.

1. Variables cannot start with a number
2. Variables cannot contains special characters
3. Variable names cannot be similar to keywords
4. It is always a good practice to use camel Case or snake casing [Ex: camelCase, Snake\_Casing]

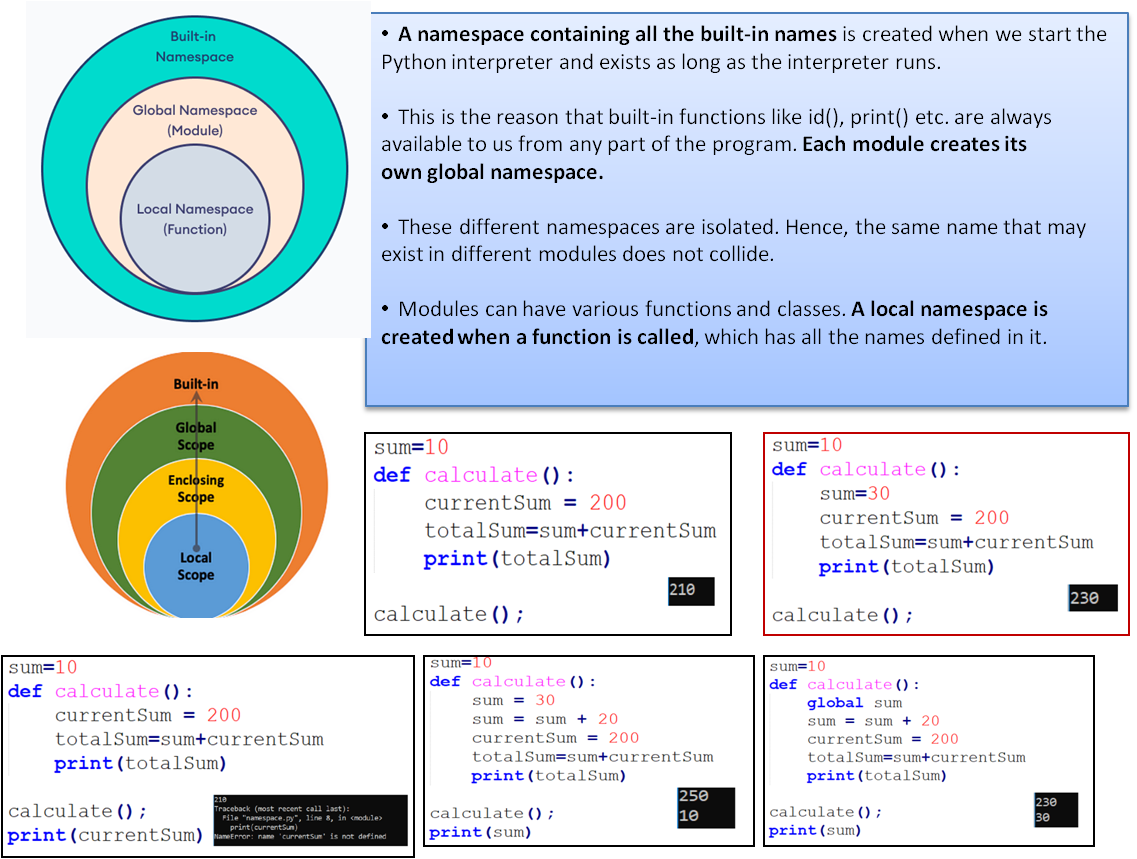
**Constants**:

This will not change once assigned.

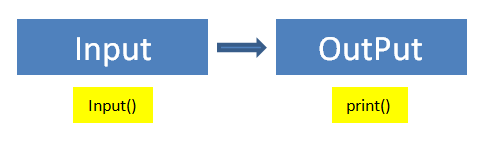
In Java we define constants like following example: *final string MY\_NAME = “Sarath”*

But, in python we can’t define constants. But we can use capital letters for easy identification.

**3.** **Name space and scope**



**4.** **Input & OutPut**

****

\* All inputs in python are by default act as strings.

Example: ATM Cash withdrawal

Cash withdrawn is: 10000 Remaining Amont is: 40000

Thank You, Visit again

cash = input(“Enter the cash amount: “)

cashAmount = int(cash)

TotalAmount = 50000

remaingAmount = 50000-cashAmount

print(“Cash withdrawn is: “, cash, “Remaining Amont is: “, remaingAmount)

print(“Thank You, Visit again”)

cash = input(“Enter the cash amount: “)

Cash withdrawn is: & 10000 & Remaining Amont is: & 40000 | Thank You, Visit again

cashAmount = int(cash)

TotalAmount = 50000

remaingAmount = 50000-cashAmount

print(“Cash withdrawn is: “, cash, “Remaining Amont is: “, remaingAmount, sep=’ & ’, end=’ | ‘)

print(“Thank You, Visit again”)

If we observe both the outputs, there are outputs separated by & and second print function output came in 1st line it self.

**Keyword arguments:**

**Sep=’ ‘**

Specifies the separator between items when using the print function. If we mention any specific argument like &, as mentioned above, then the outputs will be separated with that argument.

**end=’\n’**

Specifies what to print at the end of the output, default is a newline character, unless we manually add any arguments like above

**File**

this will write the output to a new file (Added the example below)

**flush**

Flushing the output buffer means forcing the written output to be sent immediately to its destination (such as a file or the console) instead of being stored temporarily in an internal buffer. This ensures that the output is available right away, which can be useful in real-time applications where immediate feedback is required.

In the context of the print function, setting flush=True ensures that the output is immediately written to the specified file or displayed on the console without delay.

# Example of using sep, end, file, and flush in the print function

# Open a file in write mode

with open('example\_output.txt', 'w') as file:

# Print multiple items with a custom separator and end character

Python - is - awesome!!!

Learning \* Python \* is \* fun.

print('Python', 'is', 'awesome', sep=' - ', end='!!!\n', file=file, flush=True)

# Print another line to the file

print('Learning', 'Python', 'is', 'fun', sep=' \* ', end='.\n', file=file, flush=True)

This will write the following lines to **example\_output.txt** in write mode.

**Multiple inputs:**

* When we need to pass multiple inputs, we will use **split method.**
* split() method in Python is also used to split a string into a list of substrings based on a specified delimiter. By default, it splits the string by whitespace (spaces, tabs, newlines).

cash, pin=input('Enter amout value and pin')**.split(',')**

print('pin and cash are',pin,cash)

cash=int(cash)

remainingAmount=50000-cash

print('take your cash',cash,'remainingAmount',remainingAmount,end=', ')

print('Thank you for using this ATM, visit again')

# Example of split() method

sentence = "Hello, how are you?"

words = sentence.split() # Split by whitespace by default

print(words) # Output: ['Hello,', 'how', 'are', 'you?']

# Splitting by a specific delimiter

csv\_data = "apple,orange,banana"

fruits = csv\_data.split(',') # Split by comma

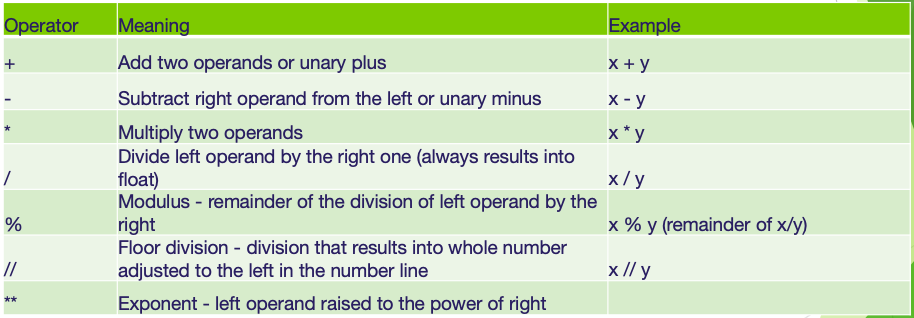
print(fruits) # Output: ['apple', 'orange', 'banana']

**5.** **Operators**

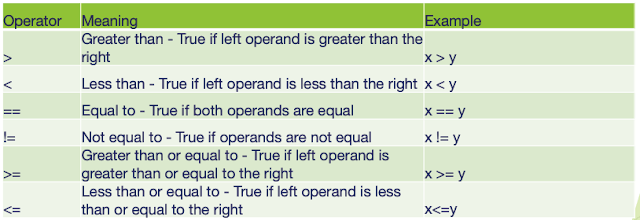
Few more

* identity operators(is, is not)
* Membership operators(in, not in)
* Ternary operator([on\_true] if [expression] else [on\_false])
* Arithmetic
* Relational[Comparison]
* Assignment
* Bitwise
* Logical

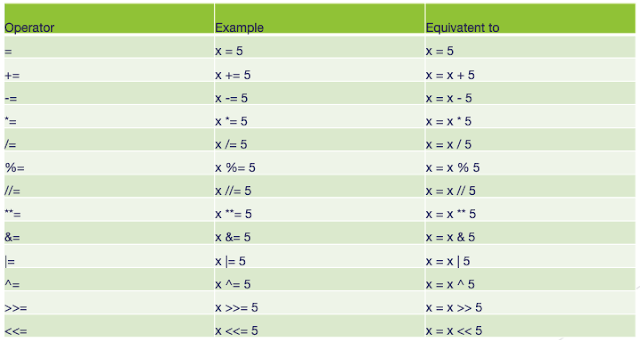
**Arthimetic Operators:** Used to perform mathematical operations.



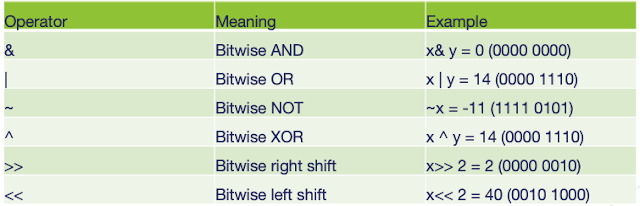
**Relational/Comparison Operators:** Used to compare values. Returns true of false according to the condition.



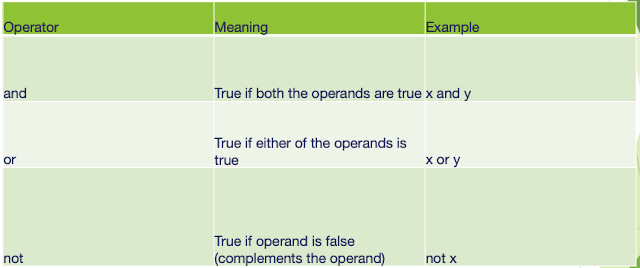
**Assignment Operator**: Used to assign values to variables



**Bitwise Operators**: Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit, hence the name.



**Logical Operators**: And, or, not. These are widely used in if statements and loops.



**6.** **Data Types**

In programming, a data type is a classification or categorization that specifies which type of value a variable can hold. Data types are essential because they determine how data is stored in memory and what operations can be performed on that data. Python, like many programming languages, supports several built-in data types. Here are some of the common data types in Python:

1. **Numeric Data Types:**
   * **int**: Represents integers (whole numbers). Example: x = 5
   * **float**: Represents floating-point numbers (numbers with decimal points). Example: y = 3.14
   * **complex**: Represents complex numbers. Example: z = 2 + 3j
2. **Sequence Types:**
   * **str**: Represents strings (sequences of characters). Example: text = "Hello, World"
   * **list**: Represents lists (ordered, mutable sequences). Example: my\_list = [1, 2, 3]
   * **tuple**: Represents tuples (ordered, immutable sequences). Example: my\_tuple = (1, 2, 3)
3. **Mapping Type:**
   * **dict**: Represents dictionaries (key-value pairs). Example: my\_dict = {'name': 'John', 'age': 30}
4. **Set Types:**
   * **set**: Represents sets (unordered collections of unique elements). Example: my\_set = {1, 2, 3}
   * **frozenset**: Represents immutable sets. Example: my\_frozenset = frozenset([1, 2, 3])
5. **Boolean Type:**
   * **bool**: Represents Boolean values (True or False). Example: is\_valid = True
6. **Binary Types:**
   * **bytes**: Represents immutable sequences of bytes. Example: data = b'Hello'
   * **bytearray**: Represents mutable sequences of bytes. Example: data = bytearray(b'Hello')
7. **None Type:**
   * **NoneType**: Represents the None object, which is used to indicate the absence of a value or a null value.
8. **Custom Data Types:**
   * You can also define your custom data types using classes and objects.

# Strings:

**1. String Data Type in Python:**

* In Python, a string is a sequence of characters, enclosed within single (' '), double (" "), or triple (''' ''' or """ """) quotes.
* Strings are immutable, meaning you cannot change the characters within a string directly. Instead, you create new strings.
* You can access individual characters in a string using indexing, e.g., my\_string[0] will give you the first character.
* **Strings support various built-in methods, such as len(), upper(), lower(), strip(), replace(), and more, for manipulation.**

**2. String Manipulation and Formatting:**

* Concatenation: You can combine strings using the + operator.
* Substrings: Use slicing to extract portions of a string, e.g., my\_string[2:5] will extract characters from the 2nd to the 4th position.
* String interpolation: Python supports various ways to format strings, including f-strings (f"...{variable}..."), %-formatting ("%s %d" % ("string", 42)), and str.format().
* Escape sequences: Special characters like newline (\n), tab (\t), and others are represented using escape sequences.
* String methods: Python provides many built-in methods for string manipulation, such as split(), join(), and startswith().

In Python, strings support a variety of built-in methods for manipulation. Here are some common string methods along with small examples for each:

### 1. str.upper()

Converts all characters in the string to uppercase.

s = "hello"

print(s.upper()) # Output: "HELLO"

### 2. str.lower()

Converts all characters in the string to lowercase.

s = "HELLO"

print(s.lower()) # Output: "hello"

### 3. str.capitalize()

Capitalizes the first character of the string.

s = "hello world"

print(s.capitalize()) # Output: "Hello world"

### 4. str.title()

Converts the first character of each word to uppercase.

s = "hello world"

print(s.title()) # Output: "Hello World"

### 5. str.strip()

Removes leading and trailing whitespace.

s = " hello "

print(s.strip()) # Output: "hello"

### 6. str.lstrip()

Removes leading whitespace.

s = " hello "

print(s.lstrip()) # Output: "hello "

### 7. str.rstrip()

Removes trailing whitespace.

s = " hello "

print(s.rstrip()) # Output: " hello"

### 8. str.replace()

Replaces occurrences of a substring with another substring.

s = "hello world"

print(s.replace("world", "Python")) # Output: "hello Python"

### 9. str.split()

Splits the string into a list of substrings based on a delimiter.

s = "hello world"

print(s.split()) # Output: ["hello", "world"]

### 10. str.join()

Joins elements of an iterable into a single string, separated by the string.

words = ["hello", "world"]

print(" ".join(words)) # Output: "hello world"

### 11. str.find()

Returns the lowest index of the substring if found, otherwise returns -1.

s = "hello world"

print(s.find("world")) # Output: 6

### 12. str.index()

Returns the lowest index of the substring if found, otherwise raises a ValueError.

s = "hello world"

print(s.index("world")) # Output: 6

### 13. str.startswith()

Returns True if the string starts with the specified prefix.

s = "hello world"

print(s.startswith("hello")) # Output: True

### 14. str.endswith()

Returns True if the string ends with the specified suffix.

s = "hello world"

print(s.endswith("world")) # Output: True

### 15. str.isdigit()

Returns True if all characters in the string are digits.

s = "12345"

print(s.isdigit()) # Output: True

### 16. str.isalpha()

Returns True if all characters in the string are alphabetic.

s = "hello"

print(s.isalpha()) # Output: True

### 17. str.isalnum()

Returns True if all characters in the string are alphanumeric.

s = "hello123"

print(s.isalnum()) # Output: True

### 18. str.islower()

Returns True if all cased characters in the string are lowercase.

s = "hello"

print(s.islower()) # Output: True

### 19. str.isupper()

Returns True if all cased characters in the string are uppercase.

s = "HELLO"

print(s.isupper()) # Output: True

### 20. str.swapcase()

Swaps the case of all characters in the string.

s = "Hello World"

print(s.swapcase()) # Output: "hELLOwORLD"

### 21. str.zfill()

Pads the string with zeros on the left, to fill the specified width.

s = "42"

print(s.zfill(5)) # Output: "00042"

### 22. str.center()

Centers the string, using a specified character for padding.

s = "hello"

print(s.center(10, '\*')) # Output: "\*\*hello\*\*\*"

### 23. str.ljust()

Left-justifies the string, using a specified character for padding.

s = "hello"

print(s.ljust(10, '\*')) # Output: "hello\*\*\*\*\*"

### 24. str.rjust()

Right-justifies the string, using a specified character for padding.

s = "hello"

print(s.rjust(10, '\*')) # Output: "\*\*\*\*\*hello"

### 25. str.len()

Gives the length of the string

s = "hello world"

print(len(s)) # Output: 11

# Numeric Data Type

**1. Numeric Data Types in Python (int, float):**

* Python supports two primary numeric data types: int for integers and float for floating-point numbers.
* Integers are whole numbers, and floats can represent both whole and fractional numbers.
* You can perform arithmetic operations on these types, including addition, subtraction, multiplication, division, and more.
* Be aware of potential issues with floating-point precision, which can lead to small inaccuracies in calculations.
* Python also provides built-in functions for mathematical operations, **such as abs(), round(), and math module** for advanced functions.

# Regex

**1. Regular Expressions for Text Processing:**

* Regular expressions (regex or regexp) are a powerful tool for pattern matching and text processing.
* The re module in Python is used for working with regular expressions.
* Common metacharacters: . (any character), \* (zero or more), + (one or more), ? (zero or one), [] (character class), | (OR), ^ (start of a line), $ (end of a line), etc.
* Examples of regex usage: matching emails, phone numbers, or extracting data from text.
* re module functions include **re.match(), re.search(), re.findall(),** and **re.sub()** for pattern matching and replacement.

………………..

……………………………………..

**AWS for DevOps:**

**1.** **AWS IAM (Identity and Access Management):**

IAM is a web service that helps you securely control access to AWS services and resources for your users. With IAM, you can create and manage AWS users and groups and use permissions to allow or deny their access to AWS resources.

**Key Components of AWS IAM**

1. **Users:**
   * An IAM user is an entity that you create in AWS to represent a person or service that interacts with AWS.
   * Users have unique credentials and can have individual permissions.
   * Each user has a name and credentials (passwords, access keys, etc.) and can be assigned specific permissions via policies.
2. **User Groups:**

* You can use user groups to specify permissions for a collection of users, which can make it easier to manage permissions for multiple users.Users in a group inherit the permissions assigned to the group.

1. **Policies:**
   * Policies are objects in AWS that, when associated with identities or resources, define their permissions.
   * Policies are written in JSON and specify what actions are allowed or denied.
   * There are different types of policies:
     + **Managed Policies:** Standalone policies that you can attach to multiple users, groups, and roles.
     + **Inline Policies:** Policies that you create and manage within a single IAM identity (user, group, or role).
   * AWS provides many pre-built managed policies for common use cases, but you can also create your own custom policies.
2. **Roles:**
   * An IAM role is an IAM identity that you can create in your account that has specific permissions.
   * Roles are similar to users but are meant to be assumed by authorized entities, such as IAM users, applications, or AWS services.
   * Roles allow you to delegate access with temporary security credentials.
   * Example use cases include granting AWS services permissions to interact with your resources, allowing users from another AWS account to access your resources, or enabling users to switch roles within your account.

**How IAM Components Work Together:**

1. **Creating Users and Groups:**
   * You create IAM users for individual people or services that need access to your AWS resources.
   * You can then place these users into groups based on their job functions or access needs.
2. **Assigning Policies:**
   * You attach policies to users, groups, or roles to define their permissions.
   * For example, you might attach a policy to a group that allows read-only access to S3 buckets, and all users in that group will inherit those permissions.
3. **Using Roles:**
   * You create roles to delegate permissions without sharing long-term credentials.
   * For example, you can create a role that allows EC2 instances to read data from S3 buckets, and then configure your EC2 instances to assume that role.

**Best Practices for Using IAM:**

1. **Principle of Least Privilege:**
   * Grant only the permissions necessary for users to perform their tasks.
2. **Use Groups for Permissions:**
   * Manage permissions by creating groups and assigning policies to these groups.
3. **Enable Multi-Factor Authentication (MFA):**
   * Enhance security by requiring MFA for users with privileged access.
4. **Regularly Review Permissions:**
   * Periodically review and audit IAM policies and access to ensure they meet current needs and best practices.
5. **Use Roles for Applications and Services:**
   * Assign roles to AWS services and applications instead of using IAM user access keys to enhance security.

By properly using IAM, you can securely manage access to your AWS resources and ensure that users and services have the appropriate permissions needed to perform their tasks.

**2.** **AWS EC2(Elastic compute cloud):**

Amazon Elastic Compute Cloud (EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers by providing secure, resizable compute capacity.

**Key Concepts and Components of AWS EC2**

1. **Instances:**
   * An EC2 instance is a virtual server in Amazon's Elastic Compute Cloud (EC2) for running applications on the Amazon Web Services (AWS) infrastructure.
   * You can launch different types of instances based on the required computing power and memory.
2. **Instance Types:**
   * EC2 provides a wide selection of instance types optimized for different use cases such as compute, memory, storage, and GPU-optimized instances.
   * Instance types vary by CPU, memory, storage, and network performance capabilities.
3. **Amazon Machine Images (AMIs):**
   * An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch an instance.
   * You can use pre-configured AMIs provided by AWS, the community, or the AWS Marketplace, or create your own custom AMIs.
4. **Key Pairs:**
   * Key pairs are used to securely connect to your EC2 instances.
   * AWS stores the public key, and you store the private key, which you use to access your instances securely.
5. **Elastic Block Store (EBS):**
   * Amazon EBS provides persistent block storage volumes for use with EC2 instances.
   * Each EBS volume is automatically replicated within its Availability Zone to protect you from component failure, offering high availability and durability.
6. **Security Groups:**
   * Security groups act as virtual firewalls to control the inbound and outbound traffic for your instances.
   * You can configure rules that allow or deny specific traffic based on IP address, port, and protocol.
7. **Elastic IP Addresses:**
   * An Elastic IP address is a static, public IPv4 address designed for dynamic cloud computing.
   * You can associate an Elastic IP address with any instance or network interface for use in your AWS account.
8. **Auto Scaling:**
   * Auto Scaling allows you to automatically scale your Amazon EC2 capacity up or down according to conditions you define.
   * With Auto Scaling, you can ensure that you have the correct number of EC2 instances available to handle the load for your application.
9. **Load Balancing:**
   * AWS provides Elastic Load Balancing (ELB) to automatically distribute incoming application traffic across multiple instances for better fault tolerance.
   * ELB supports different load balancers, including Application Load Balancer, Network Load Balancer, and Classic Load Balancer.

**How EC2 Components Work Together**

1. **Launching Instances:**
   * You launch an instance from an AMI, specifying the instance type, key pair, security groups, and EBS volumes.
   * The instance starts running in your specified Availability Zone.
2. **Managing Instances:**
   * You can connect to your instance using SSH (for Linux) or RDP (for Windows) using your private key.
   * You can monitor the instance's performance and health, and modify instance attributes as needed.
3. **Storage Management:**
   * Attach EBS volumes to your instances for persistent storage.
   * You can create snapshots of your EBS volumes for backup or replication purposes.
4. **Scaling and Load Balancing:**
   * Use Auto Scaling groups to dynamically adjust the number of instances based on demand.
   * Distribute traffic across multiple instances using Elastic Load Balancing to ensure high availability and fault tolerance.

**Best Practices for Using EC2**

1. **Choose the Right Instance Type:**
   * Select an instance type that matches your workload requirements to optimize cost and performance.
2. **Use Security Groups Effectively:**
   * Define inbound and outbound rules carefully to restrict access to your instances.
3. **Regularly Update AMIs:**
   * Keep your AMIs up to date with the latest patches and software updates.
4. **Backup Data with EBS Snapshots:**
   * Regularly create snapshots of your EBS volumes to ensure data durability and availability.
5. **Monitor and Optimize Usage:**
   * Use AWS CloudWatch to monitor instance performance and set alarms for critical metrics.
   * Review and optimize your EC2 usage to minimize costs.

By following these guidelines and leveraging the powerful features of EC2, you can build scalable, reliable, and cost-effective applications on AWS.

**3.** **VPC**

Amazon Virtual Private Cloud (VPC) is a foundational AWS service that enables you to launch AWS resources in a logically isolated virtual network that you define. With a VPC, you have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways.

**Key Concepts**

1. **VPC Components**
   * **Subnets**: Subnet is a range of IP addresses in your VPC. You can place resources in different subnets based on security and operational requirements.
   * **Route Tables**: Route tables contain a set of rules, called routes, that determine where network traffic is directed.
   * **Internet Gateway**: A horizontally scaled, redundant, and highly available VPC component that allows communication between instances in your VPC and the internet.
   * **NAT Gateway/Bastion Host**: NAT Gateway allows instances in a private subnet to connect to the internet or other AWS services, but prevents the internet from initiating a connection with those instances. A Bastion Host is a special-purpose server used to manage instances in the private subnet.
   * **Endpoints**: Private connection between your VPC and another AWS service without requiring an Internet gateway, NAT device, VPN connection, or AWS Direct Connect connection.
2. **Networking**
   * **CIDR Blocks**: Classless Inter-Domain Routing blocks allow you to allocate an IP address range for your VPC. Example: 10.0.0.0/16.
   * **Subnets**: Each subnet must reside entirely within one Availability Zone and cannot span multiple zones.
   * **Route Tables**: Each subnet must be associated with a route table, which controls the routing for the subnet.
   * **Security Groups**: Act as a virtual firewall to control the traffic for one or more instances.
   * **Network ACLs (Access Control Lists)**: An additional layer of security that acts as a firewall for controlling traffic in and out of one or more subnets.
3. **Connectivity**
   * **Internet Gateway (IGW)**: A gateway that you attach to your VPC to enable communication between resources in your VPC and the internet.
   * **NAT Gateway/Instance**: Provides internet access to instances in a private subnet.
   * **VPC Peering**: Network connection between two VPCs that enables you to route traffic between them privately.
   * **VPN Connection**: Establish a secure connection between your VPC and your on-premises network.
   * **AWS Direct Connect**: Dedicated network connection from your premises to AWS.
4. **Security**
   * **Security Groups**: Virtual firewalls that control the traffic for one or more instances.
   * **Network Access Control Lists (NACLs)**: Control traffic to and from subnets, providing an optional layer of security.
   * **Flow Logs**: Capture information about the IP traffic going to and from network interfaces in your VPC.
5. **High Availability and Fault Tolerance**
   * Deploying instances across multiple Availability Zones to ensure high availability.
   * Using Elastic Load Balancing to distribute incoming application traffic across multiple targets in multiple Availability Zones.

**Setting Up a VPC**

1. **Create a VPC**
   * Define the IP address range (CIDR block) for the VPC, e.g., 10.0.0.0/16.
2. **Create Subnets**
   * Create one or more subnets within the VPC. Subnets are mapped to specific Availability Zones.
3. **Configure Route Tables**
   * Define routes in the route table to control the traffic flow within the VPC and to the internet or other VPCs.
4. **Set Up Security Groups and NACLs**
   * Create and associate security groups with instances and NACLs with subnets to control inbound and outbound traffic.
5. **Attach an Internet Gateway**
   * Attach an Internet Gateway to the VPC to enable internet access for instances in public subnets.
6. **Launch Instances**
   * Launch instances within the VPC subnets, associate security groups, and ensure they have the required permissions and connectivity.

**Best Practices**

1. **Plan Your IP Addressing**: Carefully plan your CIDR blocks and subnets to ensure efficient IP address utilization and avoid conflicts.
2. **Isolate Resources**: Use multiple VPCs to isolate different environments such as production, staging, and development.
3. **Leverage Security Groups and NACLs**: Use security groups for instance-level security and NACLs for subnet-level security.
4. **Monitor and Log**: Enable VPC Flow Logs to monitor traffic and troubleshoot issues.
5. **Use Automation**: Automate VPC creation and configuration using AWS CloudFormation or Terraform.

AWS VPC provides a flexible and secure environment for running your workloads in the cloud. By understanding and implementing its features effectively, you can build robust and scalable applications.

**4.** **VPC (Deep Dive notes)**

**Detailed Components of AWS VPC**

**1. Subnets**

* **Public Subnets**: Subnets that have a route to the Internet Gateway. Instances in public subnets can send outbound traffic directly to the internet.
* **Private Subnets**: Subnets that do not have a route to the Internet Gateway. Instances in private subnets cannot communicate directly with the internet.
* **Subnet Configuration**:
  + **CIDR Block**: Specify the range of IP addresses for the subnet.
  + **Availability Zone**: Each subnet resides within a single Availability Zone.
  + **Auto-assign Public IP**: Option to automatically assign a public IP address to instances launched in the subnet.

**2. Route Tables**

* **Main Route Table**: The default route table that automatically comes with your VPC.
* **Custom Route Tables**: Additional route tables that you can create to manage the routing of specific subnets.
* **Routes**: Entries that specify destination CIDR blocks and targets (e.g., Internet Gateway, NAT Gateway, VPC Peering, etc.).
* **Associations**: Subnets are explicitly associated with route tables.

**3. Internet Gateway (IGW)**

* **Attaching an IGW**: An IGW must be attached to the VPC to enable internet access.
* **Route Table Configuration**: Add a route to the route table with destination 0.0.0.0/0 pointing to the IGW.

**4. NAT Gateway/Instance**

* **NAT Gateway**: Managed service provided by AWS that is highly available and scalable. Used to provide outbound internet access to instances in private subnets.
* **NAT Instance**: An EC2 instance configured to act as a NAT device. Requires manual configuration and maintenance.
* **Route Table Configuration**: Add a route to the private subnet’s route table with destination 0.0.0.0/0 pointing to the NAT Gateway/Instance.

**5. Endpoints**

* **VPC Endpoints**: Allow you to privately connect your VPC to supported AWS services and VPC endpoint services powered by AWS PrivateLink.
* **Types of Endpoints**:
  + **Interface Endpoints**: Elastic Network Interface (ENI) with a private IP address that serves as an entry point for traffic destined to a supported service.
  + **Gateway Endpoints**: Target for a route in your route table used for traffic destined to a supported AWS service (e.g., S3, DynamoDB).

**6. VPC Peering**

* **VPC Peering Connections**: A networking connection between two VPCs that enables you to route traffic between them privately.
* **Peering Setup**:
  + **Request and Accept**: One VPC owner sends a peering request to another VPC owner who accepts it.
  + **Route Tables Update**: Update route tables to route traffic between peered VPCs.
  + **DNS Resolution**: Enable DNS resolution to resolve private DNS names to private IP addresses between peered VPCs.

**7. VPN Connection**

* **Customer Gateway**: Represents your on-premises gateway device.
* **Virtual Private Gateway**: AWS gateway to which you connect your VPN.
* **Site-to-Site VPN**: Establishes a secure and encrypted connection between your VPC and your on-premises network or another VPC.

**8. AWS Direct Connect**

* **Dedicated Connection**: Establish a dedicated network connection from your premises to AWS.
* **Virtual Interfaces**: Create virtual interfaces to connect to public AWS services or to connect to your VPC.
* **Link Aggregation Groups (LAG)**: Combine multiple connections at a single AWS Direct Connect endpoint into a single managed connection.

**Security Features**

**1. Security Groups**

* **Stateful Firewalls**: Security groups are stateful; if you allow an inbound request from an IP address, the response is automatically allowed regardless of outbound rules.
* **Inbound and Outbound Rules**: Define rules to allow specific types of traffic. By default, all inbound traffic is denied, and all outbound traffic is allowed.

**2. Network ACLs (NACLs)**

* **Stateless Firewalls**: NACLs are stateless; responses to allowed inbound traffic must be explicitly allowed by outbound rules.
* **Rule Number**: Each rule is associated with a number to determine its priority (lower numbers have higher priority).
* **Default NACL**: Automatically created with your VPC and allows all inbound and outbound traffic by default.
* **Custom NACLs**: Created and associated with subnets, providing granular control over traffic.

**High Availability and Fault Tolerance**

**1. Multi-AZ Deployments**

* Deploy instances across multiple Availability Zones to protect against failure in a single zone.
* Use Elastic Load Balancing (ELB) to distribute incoming traffic across instances in multiple Availability Zones.

**2. Elastic Load Balancing (ELB)**

* **Types of Load Balancers**:
  + **Application Load Balancer (ALB)**: Ideal for HTTP/HTTPS traffic and provides advanced routing features.
  + **Network Load Balancer (NLB)**: Capable of handling millions of requests per second while maintaining ultra-low latencies.
  + **Classic Load Balancer (CLB)**: Legacy option for basic load balancing across multiple EC2 instances.
* **Health Checks**: Regularly check the health of registered targets and route traffic only to healthy instances.

**Monitoring and Logging**

**1. VPC Flow Logs**

* **Capture Traffic**: Log all IP traffic going to and from network interfaces in your VPC.
* **Destination**: Send flow logs to Amazon CloudWatch Logs or Amazon S3 for storage and analysis.

**2. CloudWatch and CloudTrail**

* **Amazon CloudWatch**: Monitor your AWS resources and applications in real-time. Set up alarms and take automated actions.
* **AWS CloudTrail**: Record API calls made in your account and deliver log files to Amazon S3. Used for governance, compliance, and auditing.

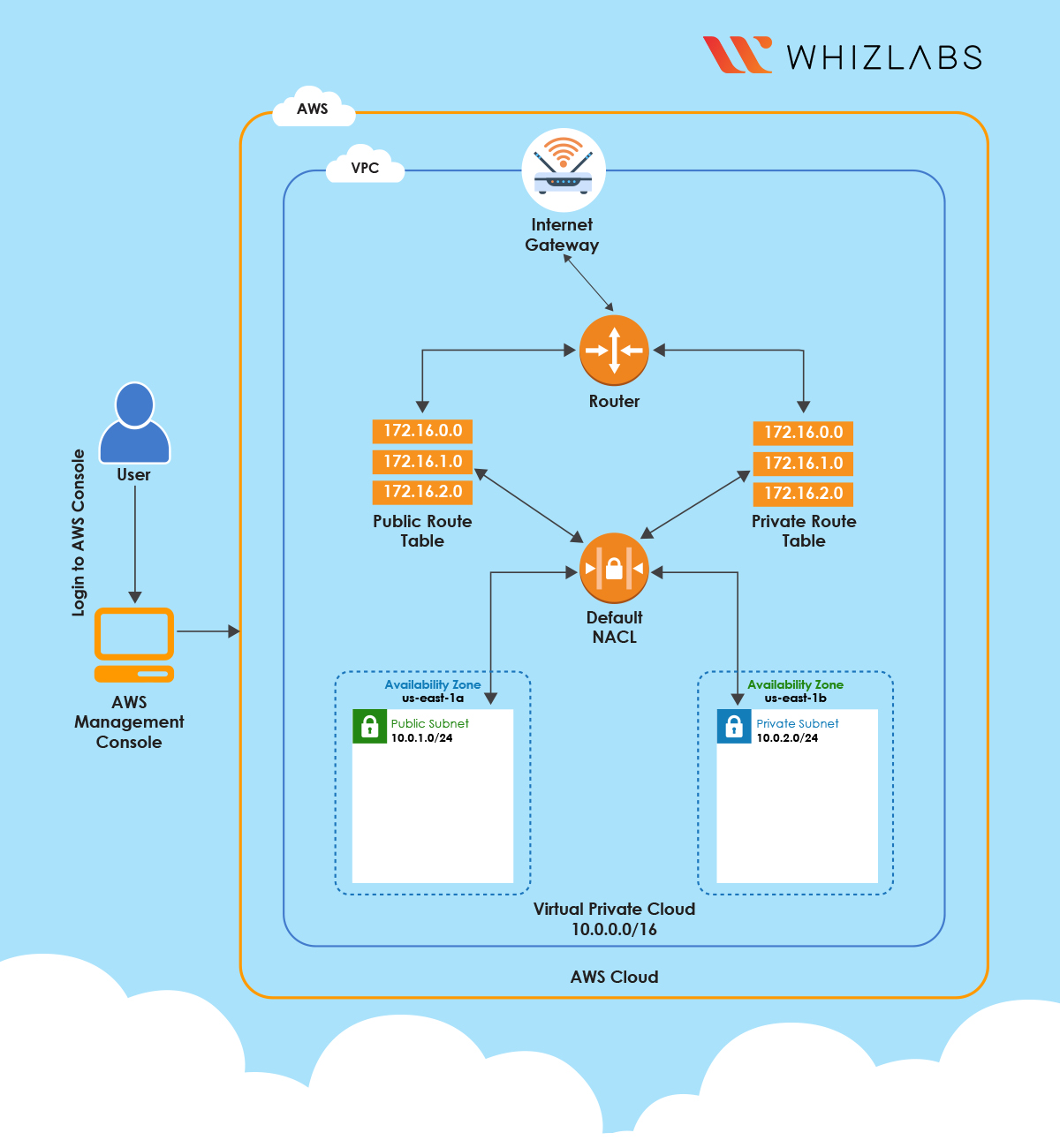
**Automation and Infrastructure as Code**

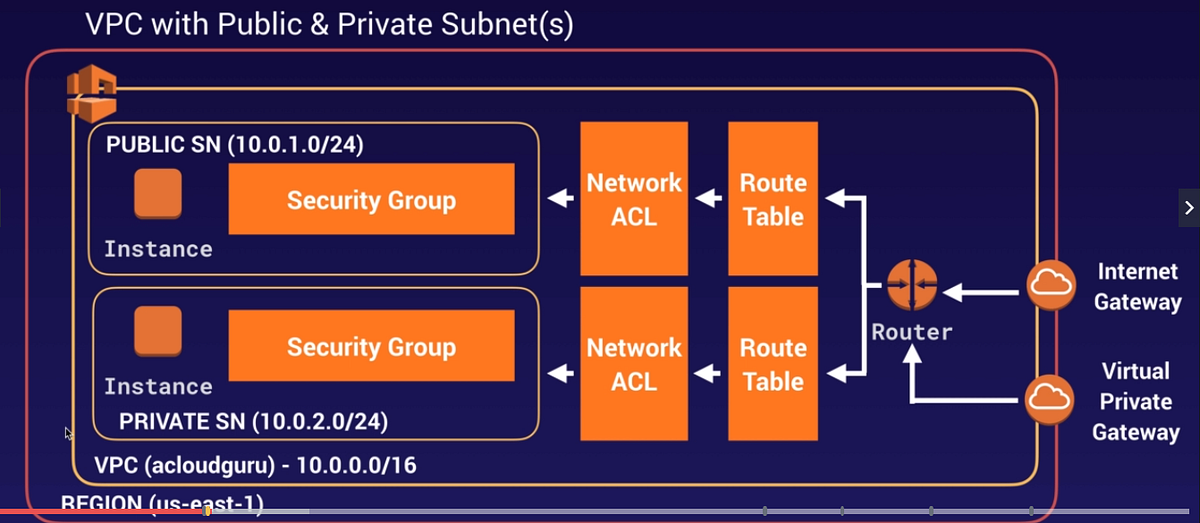
**1. AWS CloudFormation**

* Define and provision your AWS infrastructure using templates.
* Templates are written in JSON or YAML and describe the resources and their configurations.

**2. Terraform**

* Open-source infrastructure as code tool that allows you to define and provide data center infrastructure using a high-level configuration language.
* Supports AWS and other cloud providers for creating, updating, and versioning your infrastructure.





**5.** **AWS Security Groups & NACLs [Last point of security in vpc]**

**Security Groups:**

Security Groups are virtual firewalls for your instances to control inbound and outbound traffic. They operate at the instance level and are stateful.

1. **Stateful Nature**
   * If an incoming request is allowed, the response to that request is automatically allowed, regardless of outbound rules.
2. **Inbound and Outbound Rules**
   * **Inbound Rules**: Control the incoming traffic to your instances.
   * **Outbound Rules**: Control the outgoing traffic from your instances.
   * By default, all inbound traffic is denied, and all outbound traffic is allowed.
3. **Rule Specifications**
   * Each rule specifies a protocol (TCP, UDP, ICMP), port range, and source/destination IP ranges or security groups.
   * You can allow traffic from specific IP addresses or ranges (CIDR blocks), or from other security groups.
4. **Association with Instances**
   * When you launch an instance, you associate one or more security groups with the instance.
   * You can modify the associated security groups for an instance at any time.
5. **Default Security Group**
   * When you create a VPC, a default security group is automatically created.
   * The default security group allows all inbound traffic from instances assigned to the same security group and all outbound traffic.
6. **Use Cases**
   * Control access to instances based on roles or functions (e.g., web servers, database servers).
   * Allow SSH access only from specific IP addresses.
   * Allow HTTP/HTTPS traffic to web servers from the internet.

**Network Access Control Lists (NACLs):**

Network Access Control Lists (NACLs) are an additional layer of security that acts as a firewall for controlling traffic in and out of one or more subnets. They operate at the subnet level and are stateless.

1. **Stateless Nature**
   * NACLs are stateless, meaning responses to allowed inbound traffic must be explicitly allowed by outbound rules and vice versa.
2. **Inbound and Outbound Rules**
   * **Inbound Rules**: Control the incoming traffic to the subnet.
   * **Outbound Rules**: Control the outgoing traffic from the subnet.
   * By default, NACLs allow all inbound and outbound traffic.
3. **Rule Specifications**
   * Each rule is associated with a number to determine its priority (lower numbers have higher priority).
   * Rules can either allow or deny traffic based on protocol (TCP, UDP, ICMP), port range, and source/destination IP ranges (CIDR blocks).
4. **Association with Subnets**
   * Each subnet must be associated with a NACL.
   * A subnet can only be associated with one NACL at a time, but you can associate multiple subnets with the same NACL.
5. **Default NACL**
   * When you create a VPC, a default NACL is automatically created.
   * The default NACL allows all inbound and outbound traffic.
6. **Custom NACLs**
   * You can create custom NACLs to provide more granular control over traffic.
   * Custom NACLs start with no inbound or outbound rules, and you must add rules to allow or deny traffic as needed.
7. **Use Cases**
   * Add an additional layer of security to control traffic at the subnet level.
   * Define rules to deny traffic from specific IP addresses or ranges.
   * Implement subnet-level access control policies.

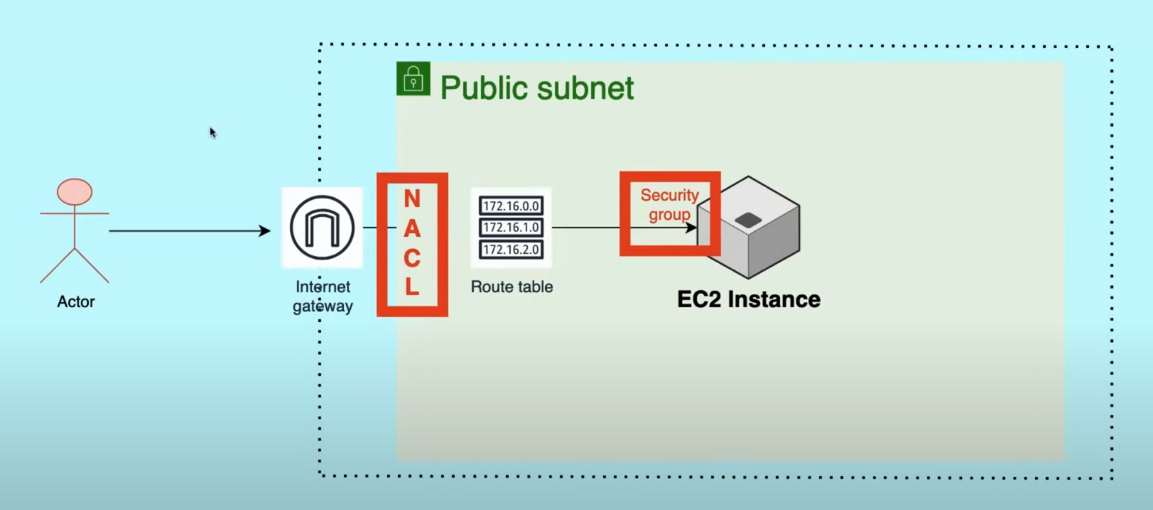
**Differences Between Security Groups and NACLs:**

1. **Level of Operation**
   * **Security Groups**: Operate at the instance level.
   * **NACLs**: Operate at the subnet level.
2. **Statefulness**
   * **Security Groups**: Stateful.
   * **NACLs**: Stateless.
3. **Default Behavior**
   * **Security Groups**: Deny all inbound traffic by default, allow all outbound traffic by default.
   * **NACLs**: Allow all inbound and outbound traffic by default.
4. **Rule Processing**
   * **Security Groups**: Evaluate all rules before deciding to allow traffic.
   * **NACLs**: Evaluate rules in order, starting with the lowest numbered rule, and stops when it finds a match.
5. **Use Cases**
   * **Security Groups**: Fine-grained control of access to individual instances.
   * **NACLs**: Broad control of access to and from subnets.

**Best Practices**

1. **Security Groups**
   * Use security groups to define instance-level security.
   * Group instances with similar security requirements into the same security group.
   * Regularly review and update security group rules to ensure they meet current security requirements.
2. **NACLs**
   * Use NACLs for an additional layer of security at the subnet level.
   * Implement NACL rules to block known malicious IP addresses or ranges.
   * Regularly review and update NACL rules to ensure they meet current security requirements.

By effectively using security groups and NACLs, you can achieve a robust security posture for your AWS resources. Would you like more detailed examples or further clarification on any specific aspect?



**6.** **Route53 [Domain Name System]:**

AWS Route 53 is a scalable and highly available Domain Name System (DNS) web service designed to provide reliable and cost-effective ways to route end-users to Internet applications. It integrates with other AWS services and offers advanced features to manage your domain names, route traffic, and monitor the health of your resources.

**Key Features of AWS Route 53:**

1. **Domain Registration**:
   * Register new domain names and manage DNS settings.
   * Supports common top-level domains (TLDs) like .com, .net, .org, etc.
2. **DNS Routing**:
   * Translates human-readable domain names into IP addresses.
   * Supports different types of routing policies: Simple, Weighted, Latency, Failover, Geolocation, Geoproximity, and Multi-Value Answer.
3. **Traffic Management**:
   * **Simple Routing**: Maps a domain name to a single resource.
   * **Weighted Routing**: Distributes traffic across multiple resources based on specified weights.
   * **Latency-Based Routing**: Routes traffic to the region with the lowest latency.
   * **Failover Routing**: Provides high availability by routing traffic to a standby resource if the primary one fails.
   * **Geolocation Routing**: Directs traffic based on the geographical location of the users.
   * **Geoproximity Routing**: Routes traffic based on the geographic location of resources and users, with optional bias to route more or less traffic to specific resources.
   * **Multi-Value Answer Routing**: Returns multiple IP addresses for DNS queries, improving redundancy and load balancing.
4. **Health Checks and Monitoring**:
   * Configure health checks to monitor the health and performance of your application endpoints.
   * Route 53 can automatically route traffic away from unhealthy endpoints to healthy ones.
   * Integrates with Amazon CloudWatch for detailed monitoring and alerting.
5. **DNS Failover**:
   * Automatically routes traffic to healthy endpoints if a primary endpoint becomes unavailable.
   * Configurable failover policies ensure high availability and fault tolerance.
6. **DNSSEC (Domain Name System Security Extensions)**:
   * Adds an additional layer of security to your domains by protecting the integrity and authenticity of DNS data.
   * Helps prevent DNS spoofing and man-in-the-middle attacks.
7. **Private DNS**:
   * Create private DNS namespaces for your Amazon VPC (Virtual Private Cloud).
   * Allows you to manage DNS resolution within your private network.
8. **Integration with Other AWS Services**:
   * Seamlessly integrates with AWS services like EC2, ELB, S3, CloudFront, and more.
   * Simplifies the management of DNS records and routing policies for AWS resources.

**How Route 53 Works:**

1. **Registering a Domain**:
   * Use Route 53 to register a new domain name.
   * Route 53 automatically configures DNS settings for the domain.
2. **Creating a Hosted Zone**:
   * A hosted zone is a container for DNS records for a specific domain.
   * Create a public or private hosted zone depending on your requirements.
3. **Adding DNS Records**:
   * DNS records define how you want to route traffic for your domain.
   * Common record types include A (Address), CNAME (Canonical Name), MX (Mail Exchange), TXT (Text), and more.
4. **Configuring Health Checks**:
   * Set up health checks to monitor the availability and performance of your endpoints.
   * Use health checks to create failover policies and route traffic to healthy resources.
5. **Applying Routing Policies**:
   * Choose the appropriate routing policy based on your use case (e.g., simple, weighted, latency-based).
   * Configure routing policies to control how Route 53 responds to DNS queries.

**Common Use Cases for Route 53:**

1. **Website Hosting**:
   * Use Route 53 to route traffic to web servers hosted on Amazon EC2, S3 static websites, or CloudFront distributions.
2. **Load Balancing**:
   * Distribute traffic across multiple EC2 instances, regions, or AWS services using weighted or latency-based routing.
3. **Failover**:
   * Ensure high availability by configuring failover routing policies to automatically route traffic to standby resources during outages.
4. **Geolocation-Based Routing**:
   * Direct users to the nearest or most appropriate endpoint based on their geographical location.
5. **Hybrid Cloud DNS**:
   * Manage DNS for both on-premises and cloud resources in a hybrid cloud environment.
6. **Multi-Region Deployment**:
   * Route traffic to multiple AWS regions to improve fault tolerance and reduce latency for global users.

**Role of a DevOps Engineer with Route 53:**

1. **DNS Management**:
   * Set up and manage DNS records for domains.
   * Ensure proper routing of traffic to application endpoints.
2. **Traffic Routing and Load Balancing**:
   * Configure routing policies to optimize traffic distribution and improve application performance.
   * Implement weighted, latency-based, or geolocation routing based on requirements.
3. **Health Monitoring and Failover**:
   * Set up health checks to monitor the health of application endpoints.
   * Configure failover policies to maintain high availability.
4. **Security**:
   * Implement DNSSEC to secure DNS data.
   * Manage access and permissions for DNS management.
5. **Integration with AWS Services**:
   * Integrate Route 53 with other AWS services for seamless DNS management.
   * Use Route 53 in conjunction with EC2, ELB, CloudFront, and other services.
6. **Automation**:
   * Automate DNS management using Infrastructure as Code (IaC) tools like AWS CloudFormation, Terraform, or AWS CLI.

**Example Scenario**

Imagine you have a web application hosted on multiple EC2 instances across different regions. You can use Route 53 to:

* Register your domain and create a hosted zone.
* Add A records pointing to the IP addresses of your EC2 instances.
* Set up latency-based routing to route users to the nearest region for lower latency.
* Configure health checks to monitor the availability of your EC2 instances.
* Implement failover routing to route traffic to healthy instances if some instances become unavailable.

By leveraging Route 53's capabilities, you ensure your web application is highly available, performs well for global users, and is resilient to failures.

**7.** **AWS S3 [Simple Storage Service]:**

**Overview**

* **Amazon S3** is a scalable, high-speed, low-cost web-based cloud storage service.
* It allows users to store and retrieve any amount of data at any time, from anywhere on the web.

**Key Features**

* **Scalability**: Automatically scales to meet your storage needs.
* **Durability and Availability**: Designed for 99.999999999% (11 9's) durability and 99.99% availability.
* **Security**: Supports data encryption, access control mechanisms, and integrates with AWS Identity and Access Management (IAM).
* **Data Management**: Features include versioning, lifecycle policies, and object tagging.
* **Data Transfer**: Supports multipart upload, Transfer Acceleration, and integration with AWS Snowball for large data transfers.
* **Storage Classes**: Offers various storage classes such as Standard, Intelligent-Tiering, Standard-IA (Infrequent Access), One Zone-IA, Glacier, and Glacier Deep Archive, each optimized for different use cases.

**Common Operations**

1. **Creating a Bucket**
   * A bucket is a container for storing objects (files).
   * Bucket names must be globally unique and follow certain naming conventions.
2. **Uploading Objects**
   * Objects can be uploaded via the AWS Management Console, AWS CLI, SDKs, or REST API.
   * Supports multipart uploads for large objects.
3. **Access Control**
   * Permissions can be managed using bucket policies, IAM policies, and Access Control Lists (ACLs).
   * Supports fine-grained access control.
4. **Versioning**
   * Allows you to keep multiple versions of an object to recover from unintended user actions and application failures.
5. **Lifecycle Management**
   * Automates the transition of objects between different storage classes and can delete objects after a specified period.
6. **Replication**
   * Cross-Region Replication (CRR) and Same-Region Replication (SRR) for automatic, asynchronous copying of objects across buckets.

**Example Bucket Policies**

**1. Allow Read-Only Access to a Specific Bucket**

This policy grants read-only access to all objects in the specified bucket.

{

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

"Statement": [

{

"Effect": "Allow",

"Principal": "\*",

"Action": "s3:GetObject",

"Resource": "arn:aws:s3:::example-bucket/\*"

}

]

}

**2. Restrict Access to a Specific IP Range**

This policy restricts access to the bucket to requests originating from a specific IP range.

{

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

"Statement": [

{

"Effect": "Deny",

"Principal": "\*",

"Action": "s3:\*",

"Resource": "arn:aws:s3:::example-bucket/\*",

"Condition": {

"NotIpAddress": {

"aws:SourceIp": "192.0.2.0/24"

}

}

}

]

}

**3. Allow Cross-Account Access to a Specific Bucket**

This policy grants another AWS account permission to access the bucket.

{

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

"Statement": [

{

"Effect": "Allow",

"Principal": {

"AWS": "arn:aws:iam::123456789012:root"

},

"Action": "s3:\*",

"Resource": [

"arn:aws:s3:::example-bucket",

"arn:aws:s3:::example-bucket/\*"

]

}

]

}

**4. Deny Delete Operations**

This policy denies all users from performing delete operations on the objects in the bucket.

{

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

"Statement": [

{

"Effect": "Deny",

"Principal": "\*",

"Action": [

"s3:DeleteObject",

"s3:DeleteObjectVersion"

],

"Resource": "arn:aws:s3:::example-bucket/\*"

}

]

}

**Best Practices**

* **Enable Versioning**: Protects against accidental deletions and overwrites.
* **Use Lifecycle Policies**: Automate data management and reduce costs.
* **Enable Logging**: Track access and usage for security and compliance.
* **Encrypt Data**: Use server-side encryption (SSE) or client-side encryption for data protection.
* **Implement IAM Policies**: Use the principle of least privilege to restrict access.

**8.** **AWS CLI [Command Line Interface]**

**Overview**

* **AWS CLI** is a unified tool to manage your AWS services from the command line.
* It allows you to control multiple AWS services and automate tasks through scripts.

**Installation**

* **Windows**: Use the MSI installer.
* **macOS**: Use brew install awscli or the bundled installer.
* **Linux**: Use package managers like apt, yum, or the bundled installer.

**Configuration**

After installing, configure the CLI with your AWS credentials.

***aws configure***

You will be prompted to enter:

* **AWS Access Key ID**
* **AWS Secret Access Key**
* **Default region name** (e.g., us-west-2)
* **Default output format** (e.g., json, yaml, text, table)

Configuration files are stored in ~/.aws/ directory:

* **config**: Contains configuration settings like region and output format.
* **credentials**: Stores your AWS access keys.

**Common Commands**

**General Syntax**

***aws [service] [operation] [options]***

**Examples**

1. **S3 Operations**
   * **List Buckets**

***aws s3 ls***

* + **Create a Bucket**

***aws s3 mb s3://my-bucket-name***

* + **Copy a File to S3**

***aws s3 cp file.txt s3://my-bucket-name/***

* + **Synchronize a Directory with S3**

***aws s3 sync my-directory/ s3://my-bucket-name/***

1. **EC2 Operations**
   * **Describe Instances**

***aws ec2 describe-instances***

* + **Start an Instance**

***aws ec2 start-instances --instance-ids i-1234567890abcdef0***

* + **Stop an Instance**

***aws ec2 stop-instances --instance-ids i-1234567890abcdef0***

1. **IAM Operations**
   * **Create a User**

***awsiam create-user --user-name new-user***

* + **List Users**

***awsiam list-users***

* + **Attach a Policy to a User**

***awsiam attach-user-policy --user-name new-user --policy-arnarn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess***

1. **CloudFormation Operations**
   * **Create a Stack**

***aws cloudformation create-stack --stack-name my-stack --template-body file://template.json***

* + **Update a Stack**

***aws cloudformation update-stack --stack-name my-stack --template-body file://template.json***

* + **Delete a Stack**

***aws cloudformation delete-stack --stack-name my-stack***

**Advanced Features**

1. **Profiles**
   * You can create multiple profiles to manage different AWS accounts or configurations.
   * Define profiles in the ~/.aws/config file:

ini

Copy code

***[profile personal]***

***aws\_access\_key\_id = AKIAIOSFODNN7EXAMPLE***

***aws\_secret\_access\_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY***

***region = us-west-2***

* + Use a specific profile:

***aws s3 ls --profile personal***

1. **Output Formats**
   * The default output format can be overridden using the --output option:

***aws ec2 describe-instances --output table***

1. **Filtering Output**
   * Use --query to filter and format the output:

***aws ec2 describe-instances --query 'Reservations[\*].Instances[\*].[InstanceId,State.Name]' --output table***

1. **AWS CLI v2 Features**
   * Interactive aws configure with guided prompts.
   * awssso support for single sign-on.
   * awseks for improved Amazon EKS management.
   * Integrated AWS CloudShell.

**Best Practices**

* **Script Automation**: Use AWS CLI in shell scripts to automate routine tasks.
* **IAM Roles and Policies**: Ensure the IAM roles and policies used with AWS CLI have the least privilege necessary.
* **Security**: Regularly rotate your access keys and avoid hardcoding credentials in scripts.
* **Configuration Management**: Use named profiles to manage different environments (e.g., development, testing, production).

**9.** **AWS CFT [CloudFormation Template]**

AWS CloudFormation is a service that provides a common language for you to model and provision AWS and third-party application resources in your cloud environment. CloudFormation allows you to use a simple text file to model and provision all the resources needed for your applications across all regions and accounts in an automated and secure manner.

**Key Concepts**

1. **Template:** A JSON or YAML formatted text file that describes the AWS resources to create, including their properties. Templates can be used repeatedly to create identical copies of the same stack or to update existing stacks.
2. **Stack:** A collection of AWS resources that you create and manage as a single unit. You create, update, and delete a collection of resources by creating, updating, and deleting stacks.
3. **Change Sets:** Enable you to preview the changes AWS CloudFormation will make to your stack. This way, you can avoid unexpected changes.
4. **Resources:** The AWS components that make up your stack, such as EC2 instances, S3 buckets, and RDS instances.
5. **Parameters:** Input values that you can pass to your template when you create or update a stack to customize how resources are configured.
6. **Mappings:** Static variables within the template. They can be used to specify conditional parameter values, among other things.
7. **Conditions:** Define conditions that control whether certain resources are created or whether certain properties are assigned a value during stack creation or update.
8. **Outputs:** Output values that you can import into other stacks or display as information to the user.

**Basic Structure of a Template**

A CloudFormation template is composed of several sections:

* **AWSTemplateFormatVersion:** (Optional) The version of the AWS CloudFormation template format.
* **Description:** (Optional) A description of what this template does.
* **Metadata:** (Optional) JSON or YAML objects that provide additional information about the template.
* **Parameters:** (Optional) Values that are passed in by the user.
* **Mappings:** (Optional) A way to specify conditional values.
* **Conditions:** (Optional) Define conditions to control resource creation.
* **Resources:** (Required) The AWS resources to be created.
* **Outputs:** (Optional) Values to be returned.

**Example Template (YAML)**

Here’s a simple CloudFormation template in YAML that creates an S3 bucket and an EC2 instance:

***AWSTemplateFormatVersion: '2010-09-09'***

***Description: Simple CloudFormation template to create an S3 bucket and an EC2 instance***

***Parameters:***

***KeyName:***

***Description: Name of an existing EC2 KeyPair to enable SSH access to the instance***

***Type: String***

***Resources:***

***MyBucket:***

***Type: AWS::S3::Bucket***

***Properties:***

***BucketName: !Sub '${AWS::StackName}-bucket'***

***MyInstance:***

***Type: AWS::EC2::Instance***

***Properties:***

***InstanceType: t2.micro***

***KeyName: !RefKeyName***

***ImageId: ami-0c55b159cbfafe1f0***

***Tags:***

***- Key: Name***

***Value: MyInstance***

***Outputs:***

***BucketName:***

***Description: Name of the S3 bucket***

***Value: !RefMyBucket***

***InstanceId:***

***Description: ID of the EC2 instance***

***Value: !RefMyInstance***

**Key Features**

1. **Infrastructure as Code:** CloudFormation treats infrastructure as code, enabling you to define and provision AWS infrastructure through text-based templates.
2. **Automated Deployment:** Automates the deployment and management of resources, reducing manual intervention and increasing repeatability.
3. **Version Control:** Templates can be stored in version control systems, allowing for versioning and tracking changes.
4. **Rollbacks:** If stack creation or updates fail, CloudFormation can automatically roll back to the previous state, ensuring stability.
5. **Cross-Stack References:** You can export values from one stack and use them in another, enabling modular and reusable architecture.
6. **StackSets:** CloudFormation StackSets extend the functionality of stacks by enabling you to create, update, or delete stacks across multiple AWS accounts and regions with a single operation.

**Best Practices**

1. **Modular Templates:** Break down large templates into smaller, reusable modules to manage complexity.
2. **Parameters and Mappings:** Use parameters and mappings to create flexible and reusable templates.
3. **Use IAM Roles:** Ensure CloudFormation has the necessary permissions to create and manage resources by using appropriate IAM roles.
4. **Change Sets:** Always review change sets before applying updates to avoid unexpected modifications.
5. **Outputs for Cross-Stack References:** Use outputs to export values that other stacks can reference, promoting reuse and modular design.
6. **Template Validation:** Use the aws cloudformation validate-template command to validate templates before deployment.

**Managing CloudFormation Stacks**

**Creating a Stack**

***aws cloudformation create-stack --stack-name my-stack --template-body file://template.yaml --parameters ParameterKey=KeyName,ParameterValue=my-key-pair***

**Updating a Stack**

***aws cloudformation update-stack --stack-name my-stack --template-body file://template.yaml --parameters ParameterKey=KeyName,ParameterValue=my-key-pair***

**Deleting a Stack**

***aws cloudformation delete-stack --stack-name my-stack***

**10.** **AWS CodeCommit [Similar to Git]**

AWS CodeCommit is a fully managed source control service that makes it easy for teams to host secure and scalable Git repositories. It eliminates the need to operate your own source control system or worry about scaling its infrastructure.

**Key Features**

1. **Fully Managed:** AWS CodeCommit is a fully managed service, meaning that you don’t need to worry about maintaining the underlying infrastructure.
2. **Secure:** CodeCommit repositories are encrypted at rest using AWS Key Management Service (KMS) and in transit using SSL.
3. **High Availability:** CodeCommit repositories are stored redundantly across multiple AWS Availability Zones to ensure high availability and durability.
4. **Scalable:** CodeCommit can handle repositories of any size, with any number of files and branches.
5. **Integration:** Easily integrates with other AWS services such as AWS CodePipeline, AWS CodeBuild, AWS Lambda, and more.
6. **Standard Git:** CodeCommit supports standard Git commands, making it easy to migrate from other Git-based repositories.

**Basic Concepts**

1. **Repository:** A storage location for your code, where you can commit, branch, and merge your code changes.
2. **Branch:** A pointer to a specific commit in a repository. Branches are used to develop features or fixes in isolation from the main codebase.
3. **Commit:** A snapshot of your files in the repository at a specific point in time.
4. **Merge:** The process of combining changes from different branches into a single branch.
5. **Pull Request:** A way to propose changes to the repository. Pull requests let others review your changes before they are merged into the main branch.

**Getting Started with AWS CodeCommit**

**Setting Up**

1. **Create a Repository:**
   * Go to the AWS Management Console.
   * Navigate to AWS CodeCommit.
   * Click on "Create repository."
   * Enter a repository name and description.
   * Click on "Create."
2. **Configure IAM Permissions:**
   * Ensure you have the necessary IAM permissions to access and interact with CodeCommit repositories.
   * You can attach the AWS managed policy AWSCodeCommitFullAccess to your IAM user or role for full access.
3. **Configure Git for AWS CodeCommit:**
   * Install Git if it is not already installed.
   * Configure your Git client to use HTTPS or SSH for connecting to CodeCommit.

**Using HTTPS**

* Generate HTTPS credentials in the AWS Management Console under "My Security Credentials."
* Configure your Git client with these credentials:

***git config --global credential.helper '!awscodecommit credential-helper $@'***

***git config --global credential.UseHttpPath true***

**Using SSH**

* Generate an SSH key pair if you don’t have one.
* Upload your SSH public key to AWS under "My Security Credentials."
* Configure your SSH client to use this key for CodeCommit:

***ssh-keygen***

# Follow the prompts to generate a key pair

**11.** **AWS Code Pipeline[Similar to Jenkins]**

AWS CodePipeline is a continuous integration and continuous delivery (CI/CD) service for fast and reliable application and infrastructure updates. CodePipeline automates the build, test, and deploy phases of your release process every time there is a code change, based on the release model you define.

**Key Features:**

1. **Automation:** Automates the steps required to release your software changes continuously.
2. **Integration:** Integrates with other AWS services like AWS CodeBuild, AWS CodeDeploy, AWS Lambda, and third-party tools.
3. **Customizable Workflow:** Define the stages and actions of your release process.
4. **Scalability:** Automatically scales with the size of your codebase and the number of changes being made.
5. **Security:** Integrates with AWS Identity and Access Management (IAM) to control who can access your pipeline and what actions they can perform.
6. **Notifications:** Integrates with Amazon CloudWatch Events and Amazon SNS to provide notifications about your pipeline status.

**Basic Concepts:**

1. **Pipeline:** A workflow construct that defines how software changes go through the release process.
2. **Stage:** A logical unit of the pipeline, consisting of one or more actions. Each stage is connected sequentially.
3. **Action:** A single operation performed on the artifacts in your pipeline, such as a build or a deploy.
4. **Artifact:** The files worked on by the actions in the pipeline, such as source code, built packages, or configuration files.

**Workflow:**

1. **Source Stage:** The pipeline is triggered by a change in the source code repository (e.g., AWS CodeCommit, GitHub, Bitbucket).
2. **Build Stage:** Compiles the source code and runs tests using a build service like AWS CodeBuild.
3. **Test Stage:** (Optional) Runs additional tests or quality checks.
4. **Deploy Stage:** Deploys the built artifacts to environments such as development, staging, or production using AWS CodeDeploy or other deployment services.

**Getting Started:**

1. **Create a Pipeline:**
   * Go to the AWS Management Console.
   * Navigate to AWS CodePipeline.
   * Click on "Create pipeline."
   * Enter a pipeline name and select a service role.
   * Define the source, build, and deploy stages.
2. **Configure IAM Permissions:**
   * Ensure you have the necessary IAM permissions to create and manage pipelines.
   * Attach the AWSCodePipelineFullAccess policy to your IAM user or role for full access.
3. **Integrate with Source Control:**
   * AWS CodeCommit: Select the repository and branch.
   * GitHub: Connect to your GitHub account and select the repository and branch.
   * Bitbucket: Connect to your Bitbucket account and select the repository and branch.
4. **Configure Build and Deploy:**
   * AWS CodeBuild: Specify the build project and environment.
   * AWS CodeDeploy: Specify the deployment group and configuration.
   * Other options: Integrate with third-party build and deploy tools.

**Example Pipeline Stages:**

**Source Stage:**

***Source:***

***Name: Source***

***Actions:***

***- Name: SourceAction***

***ActionTypeId:***

***Category: Source***

***Owner: AWS***

***Provider: CodeCommit***

***Version: 1***

***OutputArtifacts:***

***- Name: SourceArtifact***

***Configuration:***

***RepositoryName: MyRepository***

***BranchName: master***

**Build Stage:**

***Build:***

***Name: Build***

***Actions:***

***- Name: BuildAction***

***ActionTypeId:***

***Category: Build***

***Owner: AWS***

***Provider: CodeBuild***

***Version: 1***

***InputArtifacts:***

***- Name: SourceArtifact***

***OutputArtifacts:***

***- Name: BuildArtifact***

***Configuration:***

***ProjectName: MyBuildProject***

**Deploy Stage:**

***Deploy:***

***Name: Deploy***

***Actions:***

***- Name: DeployAction***

***ActionTypeId:***

***Category: Deploy***

***Owner: AWS***

***Provider: CodeDeploy***

***Version: 1***

***InputArtifacts:***

***- Name: BuildArtifact***

***Configuration:***

***ApplicationName: MyApplication***

***DeploymentGroupName: MyDeploymentGroup***

**Best Practices:**

1. **Modular Pipelines:**
   * Break down complex pipelines into modular stages and actions for better manageability.
2. **Version Control:**
   * Use version control for your pipeline configuration to track changes and collaborate with your team.
3. **Security:**
   * Apply the principle of least privilege to IAM roles and users.
   * Use encryption for sensitive data and artifacts.
4. **Testing:**
   * Include multiple test stages in your pipeline to ensure code quality and reliability.
5. **Notifications:**
   * Set up notifications to stay informed about the status of your pipelines.

**Integration with Other AWS Services:**

1. **AWS CodeBuild:** Automatically build and test your code.
2. **AWS CodeDeploy:** Deploy your applications to Amazon EC2 instances, AWS Lambda, or on-premises servers.
3. **AWS Lambda:** Use Lambda functions as actions in your pipeline for custom workflows.
4. **Amazon ECS/EKS:** Deploy containerized applications to Amazon Elastic Container Service (ECS) or Elastic Kubernetes Service (EKS).

**Monitoring and Logging:**

1. **Amazon CloudWatch:**
   * Monitor pipeline metrics and set up alarms.
   * Log pipeline actions for auditing and troubleshooting.
2. **AWS CloudTrail:**
   * Track API calls to CodePipeline for auditing and compliance.

**12.** **AWS CloudWatch**

Amazon CloudWatch is a monitoring and observability service designed to provide data and actionable insights for AWS, on-premises, and hybrid applications. It collects and tracks metrics, collects and monitors log files, sets alarms, and automatically reacts to changes in AWS resources.

**Key Features**

1. **Metrics Monitoring:** Collect and track standard and custom metrics for AWS resources.
2. **Alarms:** Set alarms to automatically perform actions such as sending notifications or executing Auto Scaling policies.
3. **Logs Monitoring:** Collect, monitor, and analyze log files from AWS and on-premises resources.
4. **Dashboards:** Create visualizations to monitor and troubleshoot AWS resources in real-time.
5. **Events:** Respond to changes in your environment by triggering workflows using Amazon EventBridge (formerly CloudWatch Events).
6. **Insights:** Utilize CloudWatch Logs Insights to search and analyze log data interactively.

**Metrics**

1. **Default Metrics:** Automatically collected for AWS services like EC2, RDS, S3, Lambda, etc.
2. **Custom Metrics:** User-defined metrics sent to CloudWatch using the API or AWS SDK.
3. **High-Resolution Metrics:** Support for metrics with a resolution down to one second.

**Logs**

1. **Log Streams:** Sequences of log events from the same source.
2. **Log Groups:** Containers for log streams with the same monitoring and retention settings.
3. **Log Retention:** Configurable retention periods to automatically delete old logs.

**Alarms**

1. **Standard Alarms:** Monitor metrics and perform actions based on thresholds.
2. **Composite Alarms:** Combine multiple alarms to reduce noise and focus on critical issues.
3. **Alarm Actions:** Trigger notifications, Auto Scaling actions, or run Lambda functions.

**Dashboards**

1. **Widgets:** Various types of widgets to display metrics and logs, including line graphs, stacked area charts, number widgets, and text.
2. **Customization:** Fully customizable to display metrics from multiple regions and accounts.
3. **Sharing:** Share dashboards with team members for collaborative monitoring.

**Events**

1. **Event Rules:** Define patterns to match events and specify targets for these events.
2. **Targets:** AWS services or custom endpoints that respond to event rules (e.g., Lambda, SNS, SQS).
3. **Event Bus:** A pipeline that routes events between AWS services, SaaS applications, and custom applications.

**Logs Insights**

1. **Query Language:** A powerful SQL-like query language for searching and analyzing log data.
2. **Visualization:** Visualize query results with bar and line charts.
3. **Interactive:** Perform interactive queries on logs to troubleshoot and analyze data.

**Monitoring and Management**

1. **Agent:** CloudWatch Agent can be installed on EC2 instances or on-premises servers to collect additional system-level metrics and logs.
2. **Service Integrations:** Integrates with AWS services like Lambda, ECS, EKS, and on-premises environments.
3. **Auto Scaling:** Automatically scale resources based on CloudWatch metrics.

**Pricing**

1. **Metrics:** Charges are based on the number of metrics, the resolution, and the number of API requests.
2. **Logs:** Charges are based on the volume of logs ingested, stored, and retrieved.
3. **Alarms and Dashboards:** Charges are based on the number of alarms and dashboards.

**Best Practices**

1. **Granular Metrics:** Use high-resolution metrics for detailed monitoring.
2. **Efficient Logging:** Set appropriate retention policies to manage log storage costs.
3. **Effective Alarms:** Use composite alarms to minimize alarm fatigue and focus on critical alerts.
4. **Custom Dashboards:** Create dashboards tailored to specific operational and business needs.
5. **Automation:** Use EventBridge to automate responses to operational changes and incidents.

**Common Use Cases**

1. **Resource Utilization Monitoring:** Monitor CPU, memory, disk usage, and network activity.
2. **Application Performance Monitoring:** Track application-level metrics and user activity.
3. **Infrastructure Monitoring:** Monitor the health and performance of AWS infrastructure.
4. **Operational Monitoring:** Set up alarms and notifications for operational thresholds and incidents.
5. **Security Monitoring:** Analyze logs for security events and compliance purposes.

**Integration with Other AWS Services**

1. **AWS Lambda:** Trigger functions in response to CloudWatch Alarms and Events.
2. **Amazon SNS:** Send notifications based on CloudWatch Alarms.
3. **Amazon S3:** Store CloudWatch Logs in S3 for long-term storage and analysis.
4. **AWS Systems Manager:** Automate operational tasks in response to CloudWatch Events.

**13.** **AWS LAMBDA**

AWS Lambda is a serverless compute service that allows you to run code without provisioning or managing servers. You pay only for the compute time you consume—there is no charge when your code is not running.

**Key Features:**

1. **Serverless:** No need to manage infrastructure.
2. **Event-driven:** Automatically runs code in response to events such as changes to data in an Amazon S3 bucket or an update to a DynamoDB table.
3. **Scalability:** Automatically scales by running code in response to each trigger.
4. **Integrated with AWS Services:** Easily integrates with other AWS services like S3, DynamoDB, SNS, etc.
5. **Supports multiple languages:** Java, Go, PowerShell, Node.js, C#, Python, and Ruby.

**How it Works:**

1. **Create a Lambda Function:** Write the code for your function and specify the runtime environment.
2. **Trigger the Function:** Set up triggers from AWS services like S3, DynamoDB, API Gateway, etc.
3. **Run and Monitor:** AWS Lambda runs the function and scales automatically. You can monitor the function's performance and errors using AWS CloudWatch.

**Comparing AWS Lambda and EC2:**

| **Feature** | **AWS Lambda** | **EC2** |
| --- | --- | --- |
| **Management** | Fully managed, no server management | Full control over the server environment |
| **Scalability** | Automatic scaling | Manual scaling (Auto Scaling available) |
| **Billing** | Pay per invocation and duration | Pay for uptime and instance type |
| **Use Cases** | Event-driven, short-duration tasks | Long-running applications |
| **Startup Time** | Milliseconds | Minutes |
| **Customizability** | Limited by runtime environment constraints | Highly customizable with full OS access |

**Example Use Cases for DevOps with AWS Lambda**

1. **Automated Resource Tagging:**
   * Use a Lambda function triggered by CloudTrail to automatically tag new resources with metadata such as the owner or project name.
2. **Log Aggregation:**
   * Trigger Lambda functions to aggregate and process logs from various sources and store them in a centralized location like S3 or Elasticsearch.
3. **Dynamic Infrastructure Scaling:**
   * Automatically scale infrastructure components based on demand. For example, add instances to an Auto Scaling group based on metrics.
4. **Deployment Automation:**
   * Trigger Lambda functions from a CodePipeline action to automate application deployments to environments like development, testing, and production.
5. **Configuration Management:**
   * Ensure compliance with configuration policies by using Lambda functions to check and enforce configurations on resources periodically.

**14.** **AWS LAMBDA – CLOUD COST OPTIMIZATION DEMO**

**Step-by-Step Procedure for Automating EBS Snapshot Cleanup with AWS Lambda**

**Objective:** Create a Lambda function that fetches all EBS snapshots owned by the same account, retrieves a list of active EC2 instances, checks for stale snapshots, and deletes them to optimize storage costs.

**Permissions Needed (Least Privilege):**

1. ec2:DescribeSnapshots
2. ec2:DescribeInstances
3. ec2:DeleteSnapshot
4. logs:CreateLogGroup
5. logs:CreateLogStream
6. logs:PutLogEvents

**Step-by-Step Implementation:**

**Step 1: Create IAM Role for Lambda**

1. Go to the IAM console.
2. Click on **Roles** and then **Create role**.
3. Select **AWS service** and choose **Lambda**.
4. Click **Next: Permissions**.
5. Attach the following policy (create a new policy if necessary): (Or else adjust the permissions for the role which will create automatically while creating the lambda function)

json

Copy code

***{***

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

***"Statement": [***

***{***

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

***"Action": [***

***"ec2:DescribeSnapshots",***

***"ec2:DescribeInstances",***

***"ec2:DeleteSnapshot",***

***"logs:CreateLogGroup",***

***"logs:CreateLogStream",***

***"logs:PutLogEvents"***

***],***

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

***}***

***]***

***}***

1. Click **Next: Tags**, add tags if needed, then click **Next: Review**.
2. Give the role a name (e.g., LambdaEBSCleanupRole) and click **Create role**.

**Step 2: Create the Lambda Function**

1. Go to the AWS Lambda console.
2. Click on **Create function**.
3. Choose **Author from scratch**.
4. Provide a function name (e.g., EBS\_Snapshot\_Cleanup).
5. Select **Python 3.x** as the runtime.
6. Under **Permissions**, choose the existing role created in Step 1.
7. Click **Create function**.
8. Default function execution time is 3 sec. If need we can change the time(Ex: 10 sec). But make sure that the pricing is also depends on execution time in AWS Lambda service.

**Step 3: Add Code to Lambda Function**

1. In the Lambda function's code editor, replace the default code with the following:

***import boto3***

***import logging***

***# Set up logging***

***logger = logging.getLogger()***

***logger.setLevel(logging.INFO)***

***ec2 = boto3.client('ec2')***

***def lambda\_handler(event, context):***

***# Fetch all EBS snapshots owned by the account***

***snapshots = ec2.describe\_snapshots(OwnerIds=['self'])['Snapshots']***

***# Fetch all running and stopped instances***

***instances = ec2.describe\_instances(***

***Filters=[***

***{'Name': 'instance-state-name', 'Values': ['running', 'stopped']}***

***]***

***)['Reservations']***

***active\_volumes = set()***

***# Get all volumes attached to running and stopped instances***

***for reservation in instances:***

***for instance in reservation['Instances']:***

***for block\_device in instance.get('BlockDeviceMappings', []):***

***active\_volumes.add(block\_device['Ebs']['VolumeId'])***

***stale\_snapshots = []***

***# Check each snapshot to see if its volume is associated with an active instance***

***for snapshot in snapshots:***

***volume\_id = snapshot.get('VolumeId')***

***if volume\_id and volume\_id not in active\_volumes:***

***stale\_snapshots.append(snapshot['SnapshotId'])***

***# Delete stale snapshots***

***for snapshot\_id in stale\_snapshots:***

***try:***

***ec2.delete\_snapshot(SnapshotId=snapshot\_id)***

***logger.info(f"Deleted stale snapshot: {snapshot\_id}")***

***except Exception as e:***

***logger.error(f"Error deleting snapshot {snapshot\_id}: {str(e)}")***

1. Click **Deploy** to save the changes.

**Step 4: Create CloudWatch Event to Trigger the Lambda Function**

1. Go to the Amazon CloudWatch console.
2. Click on **Rules** under **Events** in the left navigation pane.
3. Click **Create rule**.
4. In the **Event Source** section, select **Event Source** and then **Create a new rule**.
5. Choose **Schedule**.
6. Set the schedule (e.g., rate(1 day) for daily execution).
7. In the **Targets** section, click **Add target** and select **Lambda function**.
8. Choose the Lambda function created in Step 2 (EBS\_Snapshot\_Cleanup).
9. Click **Configure details**.
10. Provide a name and description for the rule, then click **Create rule**.

**Step 5: Testing**

1. Manually invoke the Lambda function from the Lambda console to ensure it works as expected.
2. Verify that logs are being created in CloudWatch Logs for debugging and monitoring.

**Summary**

This setup creates a Lambda function that periodically checks for stale EBS snapshots and deletes them, optimizing storage costs. The function is triggered automatically via a CloudWatch event, ensuring regular cleanup without manual intervention.

**15. AWS LAMBDA DEVOPS COMMON REAL-TIME ACTIVITIES WITH EXAMPLES**

**1. Automated Resource Tagging:**

**Use Case:** Automatically tag newly created EC2 instances with specific tags.

**Implementation Steps:**

1. **Create an IAM Role:**
   * Create a role with permissions to describe and tag EC2 instances.
   * Attach this role to the Lambda function.
2. **Create a Lambda Function:**

***import boto3***

***def lambda\_handler(event, context):***

***ec2 = boto3.client('ec2')***

***instances = event['detail']['responseElements']['instancesSet']['items']***

***for instance in instances:***

***instance\_id = instance['instanceId']***

***ec2.create\_tags(***

***Resources=[instance\_id],***

***Tags=[***

***{'Key': 'Environment', 'Value': 'Dev'},***

***{'Key': 'Owner', 'Value': 'DevOpsTeam'}***

***]***

***)***

1. **Set Up CloudTrail and CloudWatch Events:**
   * Enable CloudTrail to log API calls.
   * Create a CloudWatch Event rule to trigger the Lambda function when an EC2 instance is launched.
2. **Test the Setup:**
   * Launch a new EC2 instance and check if the tags are applied automatically.

**2. Log Aggregation and Processing**

**Use Case:** Aggregate and process logs from multiple S3 buckets and store them in Elasticsearch.

**Implementation Steps:**

1. **Create an Elasticsearch Domain:**
   * Set up an Elasticsearch domain in AWS.
2. **Create an IAM Role:**
   * Create a role with permissions to read from S3 and write to Elasticsearch.
   * Attach this role to the Lambda function.
3. **Create a Lambda Function:**

***import boto3***

***from elasticsearch import Elasticsearch, RequestsHttpConnection***

***from requests\_aws4auth import AWS4Auth***

***s3 = boto3.client('s3')***

***host = 'your-elasticsearch-domain' # For example, search-your-domain.region.es.amazonaws.com***

***def lambda\_handler(event, context):***

***awsauth = AWS4Auth('your-access-key', 'your-secret-key', 'region', 'es')***

***es = Elasticsearch(***

***hosts=[{'host': host, 'port': 443}],***

***http\_auth=awsauth,***

***use\_ssl=True,***

***verify\_certs=True,***

***connection\_class=RequestsHttpConnection***

***)***

***for record in event['Records']:***

***bucket = record['s3']['bucket']['name']***

***key = record['s3']['object']['key']***

***response = s3.get\_object(Bucket=bucket, Key=key)***

***content = response['Body'].read().decode('utf-8')***

***# Assuming each line in the log file is a JSON object***

***for line in content.splitlines():***

***es.index(index='logs', doc\_type='\_doc', body=line)***

1. **Set Up S3 Event Notification:**
   * Configure S3 to trigger the Lambda function on object creation.
2. **Test the Setup:**
   * Upload a log file to the S3 bucket and verify that the logs are indexed in Elasticsearch.

**3. Deployment Automation**

**Use Case:** Automate deployment of a web application using CodePipeline and Lambda.

**Implementation Steps:**

1. **Set Up CodePipeline:**
   * Create a pipeline with source, build, and deploy stages.
   * Use CodeCommit or GitHub as the source.
   * Use CodeBuild for building the application.
2. **Create a Lambda Function for Custom Deployment:**

***import boto3***

***def lambda\_handler(event, context):***

***# Code to deploy the application***

***ecs = boto3.client('ecs')***

***ecs.update\_service(***

***cluster='your-cluster-name',***

***service='your-service-name',***

***taskDefinition='your-task-definition'***

***)***

1. **Add Lambda Function to CodePipeline:**
   * Add a custom action in CodePipeline to trigger the Lambda function after the build stage.
2. **Test the Pipeline:**
   * Push a change to the source repository and verify that the application is built and deployed automatically.

**4. Dynamic Infrastructure Scaling**

**Use Case:** Automatically add instances to an Auto Scaling group based on CloudWatch metrics.

**Implementation Steps:**

1. **Set Up CloudWatch Alarms:**
   * Create CloudWatch alarms for metrics like CPU utilization.
2. **Create a Lambda Function for Scaling:**

***import boto3***

***def lambda\_handler(event, context):***

***asg = boto3.client('autoscaling')***

***asg.set\_desired\_capacity(***

***AutoScalingGroupName='your-asg-name',***

***DesiredCapacity=event['desired\_capacity']***

***)***

1. **Set Up CloudWatch Event Rules:**
   * Create rules to trigger the Lambda function based on CloudWatch alarms.
2. **Test the Setup:**
   * Simulate a high CPU load and verify that the Auto Scaling group scales up.

**5. Configuration Compliance**

**Use Case:** Ensure compliance by checking and enforcing configurations on resources periodically.

**Implementation Steps:**

1. **Create a Lambda Function:**

***import boto3***

***def lambda\_handler(event, context):***

***ec2 = boto3.client('ec2')***

***instances = ec2.describe\_instances(***

***Filters=[{'Name': 'tag:Environment', 'Values': ['Prod']}]***

***)***

***for reservation in instances['Reservations']:***

***for instance in reservation['Instances']:***

***if instance['InstanceType'] != 't2.micro':***

***ec2.modify\_instance\_attribute(***

***InstanceId=instance['InstanceId'],***

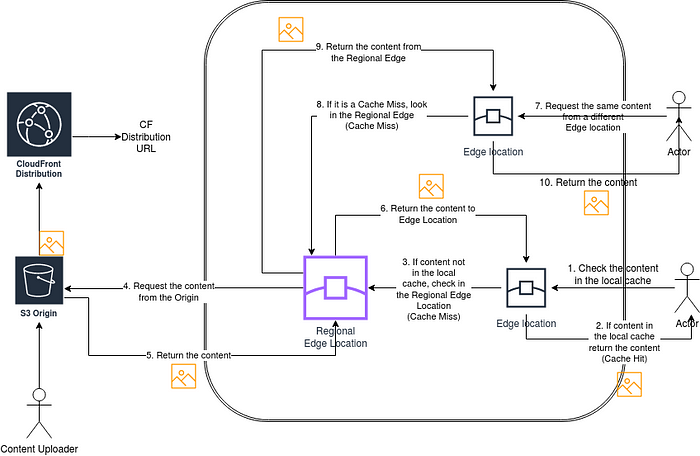
***InstanceType={'Value': 't2.micro'}***

***)***

1. **Set Up CloudWatch Events:**
   * Create a rule to trigger the Lambda function at regular intervals.
2. **Test the Setup:**
   * Ensure that non-compliant instances are reconfigured automatically.

**16.** **AWS CloudFront**

AWS CloudFront is a **content delivery network** (CDN) service provided by Amazon Web Services. It delivers content with low latency and high transfer speeds by caching copies of your content at edge locations around the world. CloudFront can serve dynamic, static, streaming, and interactive content.



**Key Features:**

1. **Global Edge Network:**
   * CloudFront has a network of edge locations worldwide, ensuring your content is delivered from the nearest server to the user, reducing latency.
2. **Content Caching:**
   * Content is cached at edge locations, which helps in speeding up the delivery of content by serving cached copies instead of fetching it from the origin server every time.
3. **Security:**
   * **AWS Shield:** Provides DDoS protection to safeguard your applications.
   * **AWS WAF:** Web application firewall to protect against common web exploits.
   * **HTTPS Support:** Enforce HTTPS to secure data in transit.
4. As end users will access to content through Edge network url, We can disable the public access of S3 buckets, So that no public can access the S3 bucket directly**Programmability:**
   * **Lambda@Edge:** Allows you to run custom code closer to your users to customize the content delivery.
   * **Edge Functions:** Perform actions such as A/B testing, user authentication, and more.
5. **Cost Efficiency:**
   * **Pay-as-you-go:** Pay only for what you use with no upfront fees.
   * **Reduced Data Transfer Costs:** Data transfer from CloudFront is often cheaper than from the origin server directly.

**Why Use CloudFront with S3-hosted Static Website?**

Even though you can host a static website directly from an S3 bucket, using CloudFront offers several benefits:

**Advantages:**

1. **Performance Improvement:**
   * **Low Latency:** Delivers content from the nearest edge location to the user.
   * **Caching:** Reduces the load on the origin S3 bucket by serving cached content.
2. **Scalability:**
   * **Handling High Traffic:** Automatically scales to handle sudden traffic spikes without any intervention.
3. **Enhanced Security:**
   * **DDoS Protection:** Built-in protection against DDoS attacks.
   * **HTTPS:** Secure data transmission with SSL/TLS.
   * **Access Control:** Restrict access to content using signed URLs and cookies.
4. **Cost Savings:**
   * **Cheaper Data Transfer:** Data transfer out from CloudFront is often less expensive than directly from S3, especially for large amounts of data.
5. **Customization:**
   * **Lambda@Edge:** Execute custom code at edge locations to modify responses, perform redirects, and more.

**Disadvantages:**

1. **Additional Cost:**
   * **Pricing:** Introduces extra costs for data transfer and requests, which may not be economical for low-traffic websites.
2. **Complexity:**
   * **Setup and Management:** Adds complexity to the setup and ongoing management compared to directly serving from S3.
   * **Cache Invalidation:** Changes to content may not be immediately visible; managing cache invalidation can add complexity and cost.

**How to Use CloudFront with S3**

1. **Set Up an S3 Bucket:**
   * Create an S3 bucket and upload your static website content.
2. **Create a CloudFront Distribution:**
   * Go to the CloudFront console and create a new distribution.
   * Specify the S3 bucket as the origin.
   * Configure cache behaviors, security settings, and other preferences.
3. **Configure Domain Name:**
   * Use the CloudFront domain name provided or set up a custom domain with Route 53.
   * Update DNS records to point to the CloudFront distribution.
4. **Enable HTTPS:**
   * Configure SSL/TLS certificates in the CloudFront distribution settings to enable HTTPS for secure content delivery.
5. **Test and Deploy:**
   * Test the CloudFront distribution to ensure content is delivered correctly.
   * Deploy your website by updating the DNS settings to point to the CloudFront distribution.

**Summary**

AWS CloudFront enhances the performance, scalability, security, and cost-efficiency of your S3-hosted static website. By caching content at edge locations, CloudFront ensures faster delivery to users worldwide. Despite some additional costs and complexity, the benefits of improved performance, security, and scalability make it a valuable addition to your web hosting infrastructure.

**17.** **AWS ECR [Elastic Container Registry]**

AWS Elastic Container Registry (ECR) is a fully managed container image registry service provided by Amazon Web Services (AWS). It enables you to store, manage, and deploy container images (Docker images) securely, making it an essential component of your containerized application development workflow. ECR integrates seamlessly with other AWS services like Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS).

## Key Benefits of ECR

* **Security**: ECR offers encryption at rest, and images are stored in private repositories by default, ensuring the security of your container images.
* **Integration**: ECR integrates smoothly with AWS services like ECS and EKS, simplifying the deployment process.
* **Scalability**: As a managed service, ECR automatically scales to meet the demands of your container image storage.
* **Availability**: ECR guarantees high availability, reducing the risk of image unavailability during critical times.
* **Lifecycle Policies**: You can define lifecycle policies to automate the cleanup of unused or old container images, helping you save on storage costs.

## Getting Started with AWS ECR

### Creating an ECR Repository

1. Go to the AWS Management Console and navigate to the Amazon ECR service.
2. Click on "Create repository" to create a new repository.
3. Enter a unique name for your repository and click "Create repository."

### Installing AWS CLI

To interact with ECR from your local machine, you'll need to have the AWS Command Line Interface (CLI) installed. Follow the instructions in the [AWS CLI User Guide](https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-quickstart.html) to install it.

### Configuring AWS CLI

After installing the AWS CLI, open a terminal and run the following command to configure your CLI with your AWS credentials:

aws configure

Enter your AWS Access Key ID, Secret Access Key, default region, and preferred output format when prompted.

## Pushing Docker Images to ECR

Now that you have your ECR repository set up and the AWS CLI configured, let's push a Docker image to ECR.

1. Build your Docker image locally using the docker build command:

***docker build -t <your-image-name><path-to-dockerfile>***

1. Tag the image with your ECR repository URI:

***docker tag <your-image-name>:<tag><your-aws-account-id>.dkr.ecr.<your-region>.amazonaws.com/<your-repository-name>:<tag>***

1. Log in to your ECR registry using the AWS CLI:

***awsecr get-login-password --region <your-region> | docker login --username AWS --password-stdin <your-aws-account-id>.dkr.ecr.<your-region>.amazonaws.com***

1. Push the Docker image to ECR:

***docker push <your-aws-account-id>.dkr.ecr.<your-region>.amazonaws.com/<your-repository-name>:<tag>***

## Pulling Docker Images from ECR

To pull and use the Docker images from ECR on another system or AWS service, follow these steps:

1. Log in to ECR using the AWS CLI as shown in Step 3 of the previous section.
2. Pull the Docker image from ECR:

***docker pull <your-aws-account-id>.dkr.ecr.<your-region>.amazonaws.com/<your-repository-name>:<tag>***

**18.** **AWS ECS [Elastic Container Service]**

AWS Elastic Container Service (ECS) is a fully managed container orchestration service provided by Amazon Web Services (AWS). It allows you to easily run, stop, and manage Docker containers on a cluster of Amazon EC2 instances or with AWS Fargate, which is a serverless compute engine for containers.

### Key Features of AWS ECS:

1. **Fully Managed:**
   * ECS handles the orchestration and management of containers, simplifying the deployment and scaling of applications.
2. **Integration with AWS:**
   * Deep integration with AWS services such as IAM, VPC, CloudWatch, ELB, and more.
3. **Flexibility:**
   * Supports both EC2 (infrastructure management) and Fargate (serverless) launch types.
4. **High Availability:**
   * Ensures high availability with multiple Availability Zones and automatic recovery of failed instances.
5. **Scalability:**
   * Automatically scales your containerized applications to meet demand using AWS Auto Scaling and Elastic Load Balancing.
6. **Security:**
   * Provides fine-grained access control through IAM roles for tasks, ensuring secure access to resources.

### Basic Concepts:

1. **Cluster:**
   * A cluster is a logical grouping of tasks or services. You can run one or more clusters in your account.
2. **Task Definition:**
   * A JSON template that describes one or more containers that form your application. It specifies various parameters for your containers, such as Docker image, CPU, memory, ports, and environment variables.
3. **Service:**
   * A service defines how many tasks should be running and maintains the desired number of tasks, ensuring that the specified number of tasks are running at all times.
4. **Task:**
   * A single running copy of any containers defined in a task definition. A task is the smallest deployable unit in ECS.
5. **Container Instance:**
   * An EC2 instance that is running the Amazon ECS container agent and has been registered into an ECS cluster.

### Working with AWS ECS:

**Prerequisites:**

* AWS CLI installed and configured with appropriate IAM permissions.
* Docker installed on your local machine.

### Step-by-Step Procedure through AWS CLI:

#### 1. Create a Cluster:

***awsecs create-cluster --cluster-name my-cluster***

#### 2. Register a Task Definition:

***{***

***"family":"sample-task",***

***"containerDefinitions":[***

***{***

***"name":"sample-app",***

***"image":"nginx:latest",***

***"cpu":256,***

***"memory":512,***

***"essential":true,***

***"portMappings":[***

***{***

***"containerPort":80,***

***"hostPort":80***

***}***

***]***

***}***

***]***

***}***

Save the above JSON to a file (e.g., task-def.json), and register the task definition:

***awsecs register-task-definition --cli-input-json file://task-def.json***

#### 3. Create a Service:

***awsecs create-service --cluster my-cluster --service-name my-service --task-definition sample-task --desired-count 2***

#### 4. List Running Tasks:

***awsecs list-tasks --cluster my-cluster***

#### 5. Scale the Service:

***awsecs update-service --cluster my-cluster --service my-service --desired-count 4***

#### 6. Delete the Service:

***awsecs delete-service --cluster my-cluster --service my-service***

#### 7. Delete the Cluster:

***awsecs delete-cluster --cluster my-cluster***

### Required IAM Permissions:

Ensure your IAM user or role has the following permissions to work with ECS:

***{***

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

***"Statement":[***

***{***

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

***"Action":[***

***"ecs:CreateCluster",***

***"ecs:DeleteCluster",***

***"ecs:RegisterTaskDefinition",***

***"ecs:DeregisterTaskDefinition",***

***"ecs:DescribeClusters",***

***"ecs:DescribeTasks",***

***"ecs:ListTasks",***

***"ecs:StartTask",***

***"ecs:StopTask",***

***"ecs:CreateService",***

***"ecs:UpdateService",***

***"ecs:DeleteService",***

***"iam:PassRole",***

***"ec2:DescribeInstances",***

***"elasticloadbalancing:\*",***

***"cloudwatch:\*"***

***],***

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

***}***

***]***

***}***

### Step-by-Step Procedure through AWS Console:

**1. Create cluster:** Name the cluster🡺Choose the infrastructure as AWS Fargate or Amazon EC2🡺create

**2. Create Task definition(Which is similar to pod.yml):** Give a name🡺Choose the infrastructure as AWS Fargate or Amazon EC2🡺choose task size(CPU & memory )🡺choose task role and task execution role🡺Add container details(container name, image url[docker image registry url], container port🡺create

**3. Run the Task(Which is similar to pod):** Once the task definition was created, we can run the task.

### Summary:

AWS ECS is a powerful service for running and managing Docker containers at scale. It provides high availability, flexibility, and deep integration with the AWS ecosystem, making it an excellent choice for deploying containerized applications. Whether using EC2 or Fargate launch types, ECS simplifies container orchestration, allowing you to focus on building your applications.

### ECS Vs EKS Vs KUBERNETES

1. **AWS Elastic Container Service (ECS):**
   * Fully managed container orchestration service by AWS.
   * Supports both EC2 and AWS Fargate launch types.
   * AWS-specific, tightly integrated with other AWS services.
2. **Amazon Elastic Kubernetes Service (EKS):**
   * Managed Kubernetes service by AWS.
   * Provides a fully managed Kubernetes control plane.
   * Allows for running Kubernetes workloads on AWS.
3. **Kubernetes:**
   * Open-source container orchestration platform.
   * Widely used and supported by many cloud providers and on-premises environments.
   * Provides a consistent API and control plane for managing containerized applications.

### Key Comparisons:

| **Feature/Aspect** | **ECS** | **EKS** | **Kubernetes** |
| --- | --- | --- | --- |
| **Management** | Fully managed by AWS | Managed Kubernetes by AWS | Self-managed (or managed by third-party providers) |
| **Integration** | Tight integration with AWS services | Integrates well with AWS services | Can integrate with various services from multiple providers |
| **Setup Complexity** | Simple setup with AWS management | More complex due to Kubernetes setup | Complex setup and management |
| **Launch Types** | EC2 and Fargate | EC2 and Fargate | Varies based on provider and environment |
| **Flexibility** | Less flexible, AWS-specific features | Highly flexible with Kubernetes ecosystem | Highly flexible, vast ecosystem of tools |
| **Scalability** | Auto Scaling with ECS | Kubernetes-native scaling and AWS Auto Scaling | Kubernetes-native scaling |
| **Customizability** | Limited customizability | High customizability with Kubernetes | Highly customizable |
| **Community Support** | Smaller AWS community | Large Kubernetes community + AWS support | Largest community and support ecosystem |
| **Learning Curve** | Lower | Moderate | High |
| **Cost Management** | Pay for what you use (integrated with AWS pricing) | Pay for managed control plane and resources | Self-managed costs can vary |

**19. AWS EKS [Elastic Kubernetes Service]**

**20.** **AWS Secrets Management Techniques**

Managing secrets, such as database credentials, API keys, and other sensitive information, is crucial for maintaining the security of your applications. AWS offers several services to manage secrets securely and efficiently. The primary services for secret management in AWS are AWS Secrets Manager and AWS Systems Manager Parameter Store.

#### AWS Secrets Manager

**Overview**: AWS Secrets Manager helps you protect access to your applications, services, and IT resources without the upfront cost and operational overhead of managing your own hardware security module (HSM) infrastructure.

**Key Features**:

1. **Automatic Rotation**: Secrets Manager allows you to rotate secrets automatically according to a schedule that you specify.
2. **Fine-Grained Access Control**: You can control access to secrets using AWS Identity and Access Management (IAM) policies.
3. **Secrets Encryption**: Secrets are encrypted at rest using encryption keys that you manage through AWS Key Management Service (KMS).
4. **Audit and Monitor**: You can track and monitor secrets using AWS CloudTrail and Amazon CloudWatch.

**Basic Operations**:

1. **Create a Secret**:

***awssecretsmanager create-secret --name MySecret --secret-string '{"username":"admin","password":"password"}'***

1. **Retrieve a Secret**:

***awssecretsmanager get-secret-value --secret-id MySecret***

1. **Update a Secret**:

***awssecretsmanager update-secret --secret-id MySecret --secret-string '{"username":"admin","password":"newpassword"}'***

1. **Delete a Secret**:

***awssecretsmanager delete-secret --secret-id MySecret***

1. **Enable Automatic Rotation**:

***awssecretsmanager rotate-secret --secret-id MySecret --rotation-lambda-arnarn:aws:lambda:region:account-id:function:rotation-function-name --rotation-interval 30***

**Integration Example with AWS Lambda**:

***import boto3***

***importjson***

***deflambda\_handler(event, context):***

***secrets\_client = boto3.client('secretsmanager')***

***secret\_value = secrets\_client.get\_secret\_value(SecretId='MySecret')***

***secret = json.loads(secret\_value['SecretString'])***

***username = secret['username']***

***password = secret['password']***

***# Use the username and password for your application logic***

#### AWS Systems Manager Parameter Store

**Overview**: AWS Systems Manager Parameter Store provides secure, hierarchical storage for configuration data management and secrets management. You can store data such as passwords, database strings, and license codes as parameter values.

**Key Features**:

1. **Secure String Parameters**: Store sensitive data encrypted with AWS KMS.
2. **Parameter Policies**: Implement policies for parameter expiration and enforced value patterns.
3. **Hierarchy**: Organize parameters into hierarchies to manage resources efficiently.

**Basic Operations**:

1. **Create a Parameter**:

***awsssm put-parameter --name "/myapp/dbpassword" --value "password" --type"SecureString" --key-id "alias/aws/ssm"***

1. **Retrieve a Parameter**:

***awsssm get-parameter --name "/myapp/dbpassword" --with-decryption***

1. **Update a Parameter**:

***awsssm put-parameter --name "/myapp/dbpassword" --value "newpassword" --type"SecureString" --overwrite***

1. **Delete a Parameter**:

***awsssm delete-parameter --name "/myapp/dbpassword"***

**Integration Example with AWS Lambda**:

***import boto3***

***deflambda\_handler(event, context):***

***ssm\_client = boto3.client('ssm')***

***parameter = ssm\_client.get\_parameter(Name='/myapp/dbpassword', WithDecryption=True)***

***db\_password = parameter['Parameter']['Value']***

***# Use the db\_password for your application logic***

#### Best Practices for Secret Management

1. **Least Privilege**: Grant the minimum necessary permissions to users and applications to access secrets.
2. **Automatic Rotation**: Regularly rotate secrets to reduce the risk of exposure.
3. **Audit and Monitor**: Enable logging and monitoring to track access and usage of secrets.
4. **Encrypt Secrets**: Always encrypt secrets at rest and in transit.
5. **Environment Variables**: Avoid hardcoding secrets in your application code. Use environment variables or secret management services.

#### Conclusion

AWS Secrets Manager and AWS Systems Manager Parameter Store provide robust solutions for managing secrets in your AWS environment. They offer features like automatic rotation, encryption, fine-grained access control, and integration with other AWS services, ensuring that your secrets are managed securely and efficiently.

**21.** **AWS Config**

AWS Config is a service that enables you to assess, audit, and evaluate the configurations of your AWS resources. It helps you to monitor and record resource configurations and automate the evaluation of recorded configurations against desired configurations. With AWS Config, you can track changes to the configurations of your AWS resources and ensure compliance with policies.

#### Key Features:

1. **Configuration Recorder:** Records configurations of supported resources.
2. **Configuration History:** Maintains a history of configurations for resources.
3. **Configuration Snapshot:** Captures the latest configurations of resources.
4. **Configuration Stream:** Delivers configuration changes in real-time.
5. **Rules:** Evaluates resource configurations against desired settings.
6. **Conformance Packs:** Collections of AWS Config rules and remediation actions.
7. **Compliance:** Tracks the compliance state of your resources against the rules.

#### Common Use Cases:

* **Compliance Auditing:** Ensure resources comply with regulatory standards.
* **Security Analysis:** Identify non-compliant resources.
* **Resource Monitoring:** Track configuration changes and relationships.

#### Example Scenario

We'll create an AWS Config rule to detect compliant and non-compliant EC2 instances based on whether they have monitoring enabled.

#### Step-by-Step Guide

1. **Set Up AWS Config:**
   * Navigate to the AWS Config service in the AWS Management Console.
   * Set up the AWS Config recorder to start recording configurations for your resources.
   * Ensure that EC2 instances are included in the resources to be recorded.
2. **Create a Custom AWS Config Rule:**
   * You can create a custom rule using AWS Lambda to evaluate whether EC2 instances have monitoring enabled.
3. **Define the Rule:**
   * The rule will check the Monitoring attribute of EC2 instances to determine compliance.
4. **Lambda Function for Custom Rule:**
   * Create a Lambda function to evaluate the EC2 instances. Below is a sample Lambda function code:

***import boto3***

***defevaluate\_compliance(configuration\_item):***

***ec2 = boto3.client('ec2')***

***instance\_id = configuration\_item['resourceId']***

***# Describe the instance to get monitoring status***

***response = ec2.describe\_instances(InstanceIds=[instance\_id])***

***monitoring\_state = response['Reservations'][0]['Instances'][0]['Monitoring']['State']***

***# Check if monitoring is enabled***

***ifmonitoring\_state == 'enabled':***

***return'COMPLIANT'***

***else:***

***return'NON\_COMPLIANT'***

***deflambda\_handler(event, context):***

***configuration\_item = event['configurationItem']***

***compliance\_type = evaluate\_compliance(configuration\_item)***

***evaluation = {***

***'ComplianceResourceType': configuration\_item['resourceType'],***

***'ComplianceResourceId': configuration\_item['resourceId'],***

***'ComplianceType': compliance\_type,***

***'OrderingTimestamp': configuration\_item['configurationItemCaptureTime']***

***}***

***config = boto3.client('config')***

***config.put\_evaluations(***

***Evaluations=[evaluation],***

***ResultToken=event['resultToken']***

***)***

1. **Deploy the Lambda Function:**
   * Deploy the Lambda function and ensure it has the necessary permissions to describe EC2 instances and put evaluations in AWS Config.
2. **Create the AWS Config Rule:**
   * In the AWS Config console, create a new rule and select "Custom Lambda Rule."
   * Provide the ARN of the Lambda function created earlier.
   * Specify the trigger types and resources (EC2 instances) to be evaluated.
3. **Compliance Evaluation:**
   * AWS Config will now use the custom rule to evaluate the compliance of your EC2 instances based on whether monitoring is enabled or not.
   * You can view the compliance results in the AWS Config console.

#### Conclusion

Using AWS Config with a custom Lambda function, you can detect compliant and non-compliant EC2 instances based on specific configuration attributes, such as whether monitoring is enabled. This allows you to maintain compliance and ensure your resources are configured according to your organization's policies.

**22.** **AWS Load Balancers**

AWS provides several types of load balancers that distribute incoming traffic across multiple targets, such as EC2 instances, containers, and IP addresses, in one or more Availability Zones. Load balancers improve the availability and fault tolerance of your applications.

#### Types of AWS Load Balancers

1. **Elastic Load Balancing (ELB):**
   * **Classic Load Balancer (CLB):** This is the original load balancer provided by AWS. It operates at both the request and connection level. It’s suitable for applications that were built within the EC2-Classic network.
   * **Application Load Balancer (ALB):** Operates at the application layer (Layer 7) and is ideal for HTTP and HTTPS traffic. It offers advanced routing, SSL termination, and visibility features targeted at modern application architectures, including microservices and container-based applications.
   * **Network Load Balancer (NLB):** Operates at the transport layer (Layer 4). It is designed to handle millions of requests per second while maintaining ultra-low latencies. Ideal for applications that require extreme performance.
   * **Gateway Load Balancer (GWLB):** Operates at the network layer and is designed to deploy, scale, and manage virtual appliances such as firewalls, intrusion detection and prevention systems, and deep packet inspection systems.

#### Key Features

1. **High Availability:** Automatically distributes incoming application traffic across multiple targets in multiple Availability Zones, ensuring high availability.
2. **Automatic Scaling:** Scale your load balancers as your traffic demands grow and shrink.
3. **Health Checks:** Continuously monitor the health of registered targets and route traffic only to healthy targets.
4. **Security:** Integrate with AWS Certificate Manager (ACM) for SSL/TLS certificates and use security groups and AWS WAF (Web Application Firewall) for enhanced security.
5. **Routing Mechanisms:** Different types of load balancers support various routing mechanisms:
   * ALB supports host-based, path-based, and HTTP header-based routing.
   * NLB supports IP-based and connection-based routing.
6. **Monitoring and Logging:** Amazon CloudWatch metrics, AWS CloudTrail logs, and access logs for monitoring and analysis.

#### Use Cases

1. **Application Load Balancer (ALB):**
   * Suitable for web applications and microservices.
   * Advanced request routing based on URL, hostname, headers, and more.
   * Supports WebSocket and HTTP/2.
2. **Network Load Balancer (NLB):**
   * Ideal for applications needing high performance and low latency, such as gaming or financial applications.
   * Can handle sudden and volatile traffic patterns.
   * Supports static IP addresses and elastic IP.
3. **Classic Load Balancer (CLB):**
   * Legacy applications that need basic load balancing.
   * Simple and easy to configure.
4. **Gateway Load Balancer (GWLB):**
   * Deploy, scale, and manage virtual network appliances.
   * Ideal for security applications like firewalls and intrusion detection systems.

#### Conclusion

AWS Load Balancers are crucial components for ensuring high availability, scalability, and reliability of your applications. Depending on the specific needs of your application, AWS provides different types of load balancers that can help you distribute traffic efficiently and maintain the performance of your services.

**23.** **Migrating applications to AWS Cloud**

Migrating applications to the cloud involves a series of strategic decisions and technical steps to ensure a smooth transition. Here are detailed notes on the process:

### 1. ****Understanding Cloud Migration****

**Definition:** Cloud migration is the process of moving data, applications, or other business elements from an organization's on-premises infrastructure to a cloud computing environment.

**Benefits:**

* **Scalability:** Easily scale resources up or down based on demand.
* **Cost Efficiency:** Pay only for the resources you use.
* **Performance:** Access to high-performance computing resources.
* **Disaster Recovery:** Enhanced backup and recovery options.
* **Accessibility:** Access applications and data from anywhere.

### 2. ****Migration Strategies (The 6 Rs)****

1. **Rehost (Lift and Shift):**
   * Moving applications to the cloud without making significant changes.
   * Quick and cost-effective for immediate benefits.
   * Tools: AWS VM Import/Export, Azure Site Recovery.
2. **Refactor (Re-architect):**
   * Modifying the application architecture to take full advantage of cloud-native features.
   * Improves scalability, performance, and maintainability.
   * Requires significant effort and expertise.
3. **Revise (Re-platform):**
   * Making a few cloud optimizations without changing the core architecture.
   * For example, moving a database to a managed cloud database service.
4. **Rebuild:**
   * Rebuilding the application from scratch using cloud-native technologies.
   * Suitable for applications that need to leverage advanced cloud features.
5. **Replace:**
   * Replacing the application with a cloud-based SaaS solution.
   * Cost-effective and reduces maintenance overhead.
6. **Retain:**
   * Keeping some applications on-premises if they are not suitable for migration.
   * This could be due to compliance, security, or performance reasons.

### 3. ****Pre-Migration Planning****

1. **Assessment:**
   * Evaluate the current application portfolio.
   * Identify dependencies and interdependencies.
   * Determine the readiness of applications for the cloud.
2. **Choosing the Cloud Provider:**
   * Compare services, pricing, and support offered by providers like AWS, Azure, GCP.
   * Consider the provider's compliance and security measures.
3. **Defining Objectives:**
   * Clear understanding of what you want to achieve (e.g., cost savings, performance improvement, scalability).
4. **Developing a Migration Plan:**
   * Detailed roadmap with timelines and milestones.
   * Risk assessment and mitigation strategies.

### 4. ****Migration Process****

1. **Pilot Migration:**
   * Start with a small, non-critical application to test the migration process.
   * Use feedback to refine the strategy.
2. **Data Migration:**
   * Use tools and services provided by cloud providers (e.g., AWS Data Migration Service, Azure Database Migration Service).
   * Ensure data integrity and security during the transfer.
3. **Application Migration:**
   * Migrate the application components following the chosen strategy.
   * Test the application thoroughly in the cloud environment.
4. **Optimization:**
   * Fine-tune the application to optimize performance and cost.
   * Implement auto-scaling and load balancing.

### 5. ****Post-Migration Activities****

1. **Validation and Testing:**
   * Conduct thorough testing to ensure applications are working as expected.
   * Validate data integrity and performance.
2. **Monitoring and Optimization:**
   * Use monitoring tools to track performance, usage, and cost.
   * Continuously optimize for better performance and cost-efficiency.
3. **Security and Compliance:**
   * Implement cloud security best practices (e.g., encryption, IAM policies).
   * Ensure compliance with relevant regulations and standards.

### 6. ****Common Challenges and Solutions****

1. **Data Transfer Issues:**
   * Use high-speed data transfer tools.
   * Plan for data transfer windows to minimize downtime.
2. **Application Downtime:**
   * Use blue-green deployment or canary releases to minimize downtime.
3. **Skill Gaps:**
   * Provide training for the team on cloud technologies.
   * Leverage cloud provider support and resources.
4. **Cost Management:**
   * Use cost management tools to monitor and control cloud spending.
   * Implement resource tagging for better cost tracking.

### 7. ****Tools and Services****

* **AWS:** AWS Migration Hub, AWS Database Migration Service, AWS Snowball.
* **Azure:** Azure Migrate, Azure Site Recovery, Azure Database Migration Service.
* **GCP:** Google Cloud Migrate, Velostrata, Google Cloud Transfer Service.

**24.** **AWS 3-Tire Architecture**

Visit[Building a Resilient Three-Tier Architecture on AWS with Deploying MERN Stack Application | Showwcase](https://www.showwcase.com/article/35459/building-a-resilient-three-tier-architecture-on-aws-with-deploying-mern-stack-application)

**Note**: Now a day, most of the organizations are prefering kubernetes architecture rather than 3-tire architecture.

In interview point of view, we have to say that I worked on 3-tire in my previous project, and in current project worked on kubernetes architecture.

**1. Creating VMs/EC2s in Cloud(ex: AWS)**

a. We can use AWS CLI, AWS CDK, AWS CFT (CloudFormation Templates, AWS API(REST API) if the organization is only using the AWS cloud. Because these are more efficient for AWS and will get any quick support from AWS.

b. Now a Days, Many organizations are using hybrid model in cloud. Some services are in GCP(ex: ML, Kubernetes…) and some services in AWS, In those cases, Terraform whould be the preferred on to use.

Note: It is not always recommended to use terraform because of its popularity.

**2. How to connect AWS CLI with AWS?**

Download and install CLI from google. Run **aws –version** in cmd prompt to verify the aws installation. Once done,

Open AWS console🡺profile🡺security credentials🡺create access keys Now open cmd prompt and run **aws configure** and enter the required details. That’s it.

Note: There are number of docs regarding the commands to manage the AWS services from AWS CLI(Like creating EC2s, S3s…e

**Kubernetes**

**1. Why Kubernetes?**

Ans: While dealing with Docker, it has some drawbacks on below filelds.

A. Single Host (By default it does’t have any clusters)

B. Auto-Healing

C. Auto-Scaling

D. Enterprise level solutions(Load balancers, API Gateways, Firewalls…etc)

To overcome these, Have to adopt to kubernetes and it has a huge contributors community which is making it a good solution for all the drawbacks.

**2.** **Kubernetes architecture?**

Data Plane (Workload execution)

Control Plane (Cluster management)

1. **Control manager**: This basically controls the master cluster state.

2. **API Server**: This helps to communicate with the worker nodes. It will handle requests and cluster state changes

3. **Scheduler**: It will assign pods to nodes

4. **etcd**: Key-value store for cluster data

5. **Cloud Controller manager**: Which basically handles the cloud integrations(Not applicable if the kubernetes is in on-premises)

1. **Kubelet**: Manages the pods and containers on each node

2. **Kube-proxy**: Handles networking for pods

3. **Container runtime**: Runs the actual container on nodes.

1. Kubelet: Manages the pods and containers on each node

2. Kube-proxy: Handles networking for pods

3. Container runtime: Runs the actual container on nodes.

User

In Kubernetes, the architecture is divided into two main planes: the control plane and the data plane. Each plane has specific roles and components that are essential for the operation and management of the Kubernetes cluster.

**### Control Plane**

The control plane is responsible for managing the state of the Kubernetes cluster. It makes decisions about the cluster (e.g., scheduling, scaling) and detects and responds to cluster events (e.g., when nodes go down).

\*\*Key Components of the Control Plane:\*\*

1. kube-apiserver:

-Role: Serves as the front-end for the Kubernetes control plane. It exposes the Kubernetes API.

-Function: Handles RESTful requests, validates them, and processes the changes to the cluster state. Acts as a gatekeeper for authentication and authorization.

2. etcd:

- Role: A distributed key-value store used to store all cluster data.

- Function: Provides a consistent and highly-available source of truth for the cluster state, storing configuration data, metadata, and the state of the cluster.

3. kube-scheduler

- RoleAssigns newly created pods to nodes.

- Function: Watches for unassigned pods and assigns them to nodes based on resource availability and other constraints (affinity, taints, etc.).

4. kube-controller-manager:

-Role: Runs controller processes.

- Function: Manages various controllers that handle routine tasks such as node management, replication, and endpoint management. Examples include the Node Controller, Replication Controller, and Endpoint Controller.

5. cloud-controller-manager:

- Role: Interacts with the underlying cloud provider (if applicable).

- Function: Manages cloud-specific control logic, such as managing load balancers, volume provisioning, and node management in a cloud environment.

**### Data Plane**

The data plane is responsible for running user workloads (containers) and ensuring the actual implementation of the desired state set by the control plane.

**Key Components of the Data Plane:**

1. kubelet:

-Role: An agent that runs on each node in the cluster.

- Function: Ensures that containers are running in a pod. It receives pod definitions from the kube-apiserver and manages the pods on its node.

2. kube-proxy:kube-proxy can operate in three different modes: **UserSpace Mode, IPtable Mode and IPVS Mode**

-Role: Network proxy that runs on each node.

-Function: Maintains network rules on nodes, allowing network communication to pods. It forwards traffic to the appropriate pod across nodes within the cluster.

3. Container Runtime:

-Role: Software responsible for running containers.

-Function: Runs and manages containers on a node. Examples include Docker, containerd, and CRI-O.

**3.** **Popular Kubernetes Distributions**

1. Google Kubernetes Engine (GKE)
2. Amazon Elastic Kubernetes Service (EKS)
3. Azure Kubernetes Service (AKS)
4. Red Hat OpenShift (IBM)
5. Rancher
6. VMware Tanzu
7. Canonical Kubernetes (Charmed Kubernetes)
8. Kubernetes on Bare Metal (kops, kubeadm, etc.)**Currently using in our organization**
9. IBM Cloud Kubernetes Service (IKS)
10. DigitalOcean Kubernetes (DOKS)
11. Alibaba Cloud Container Service for Kubernetes (ACK)
12. Oracle Container Engine for Kubernetes (OKE)

**4.** **Organizations preferences on using kubernetes and kubernetes distribution.**

1. Some organizations will use kubernates in staging and pre-prod environments, So that N number of developers can make use of the platform for their testing. But When comes to production, organizations, will adpt to any popular distribusions(OpenShift, GKE, EKS…etc) for the quick support if goes into any issue.
2. Still many organizations are using kubernetes in production also, if they don’t have any urgency in the support. (Support from kubernetes opensouce support)

**5.** **How and what the DevOps engineers do with kubernetes in production in organization(Life cycle)**

**Using KOPS(Only if the organization using kubernetes, not the distribusions)**

* Installing
* Upgrading
* Modifiying the clusters
* Deleting the clusters
* Which are all done using KOPS

Note: If the organizations are using distribusions, Then the installation and configuration will be differe.

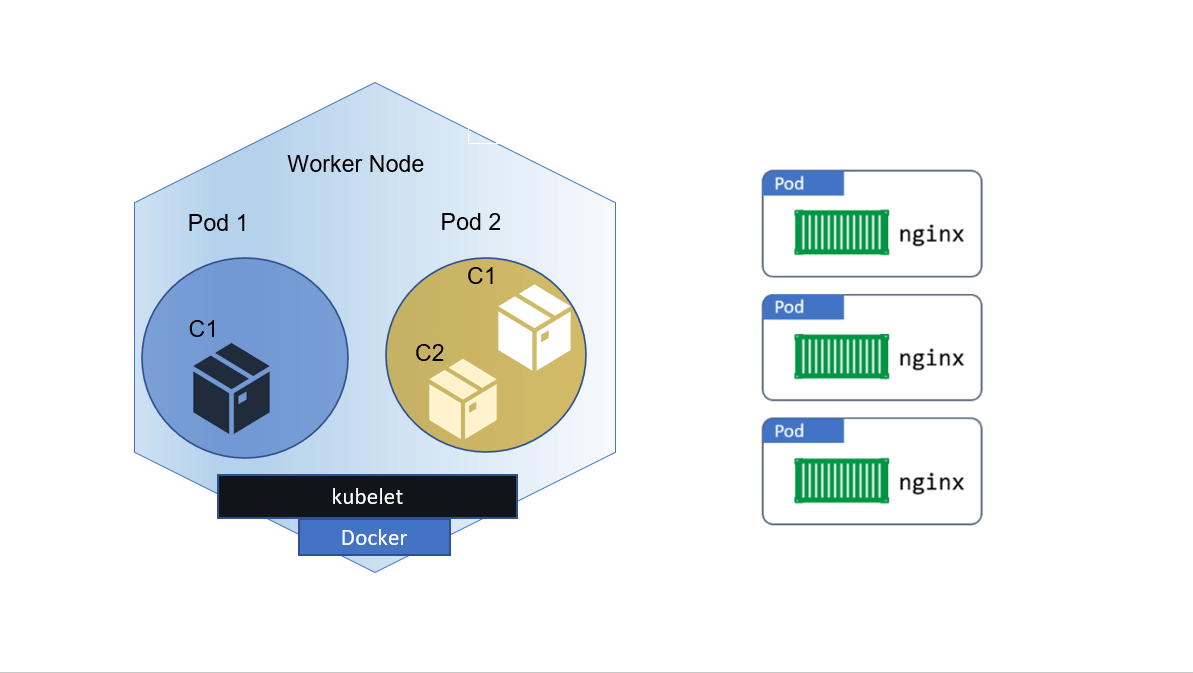
**Interview Tip: We can tell the interviewer that in our organization, we are installing kubernetes using KOPS.**

**6.** **Kubernetes Installation Using KOPS on EC2**

<https://github.com/GuruSarathKumar-Kota/Kubernetes>

**7.** **What is Pod?**

Ans: It is a smallest deployment in the kubernetes. In side the pod, the container will run (Either single container or multicontainer). A pod in kubernetes is a runtime specification of a container in docker. A pod provides more declarative way of defining using YAML and you can run more than one container in a pod.



kubectl create -f pod.yaml

kubectl apply -f pod.yaml

kubectl get pods

kubectl delete pod <pod name>

kubectl get pods -o wide

kubectl describe pod <pod name>

kubectl get all (To get all pods, deploys, nodes)

kubectl get all –A (To get all namespaces, clusters…)

kubectl get pods –v=<number(max:9> this will give the information regarding api calls happening behind the command

Simple Comparision with Docker:

Docker==Kubernetes

Compose file==pod.yaml file

docker run==kubectl apply/kubectl create

Docker ps==kubectl get pods/nodes

**Sample pod.yaml file: apiVersion: v1**

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

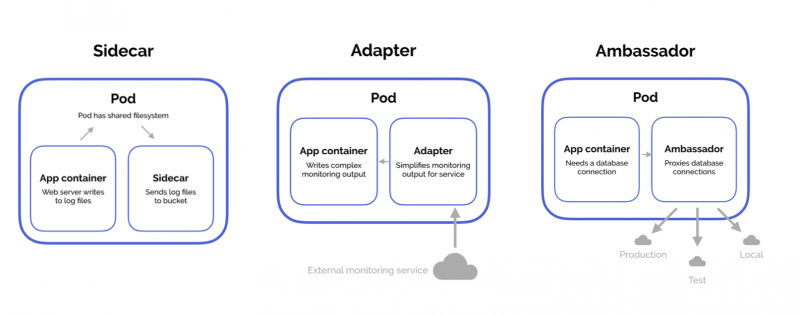
image: nginx:1.14.2

ports:

- containerPort: 80

**8.** **Multi-Container Pod Design Patterns in Kubernetes**

Ans: There are three common design patterns and use-cases for combining multiple containers into a single pod

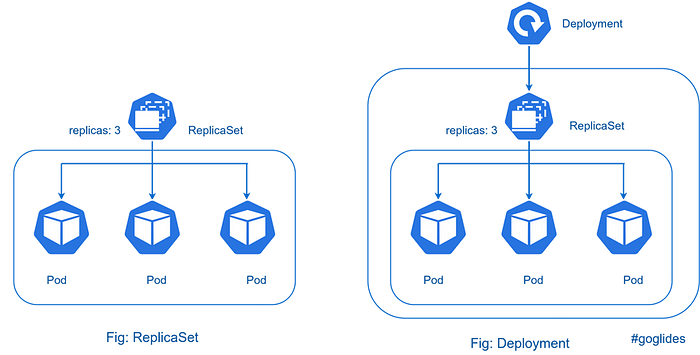


**9.What is Deployment? And****Deployment vs ReplicaSet vs Pod**

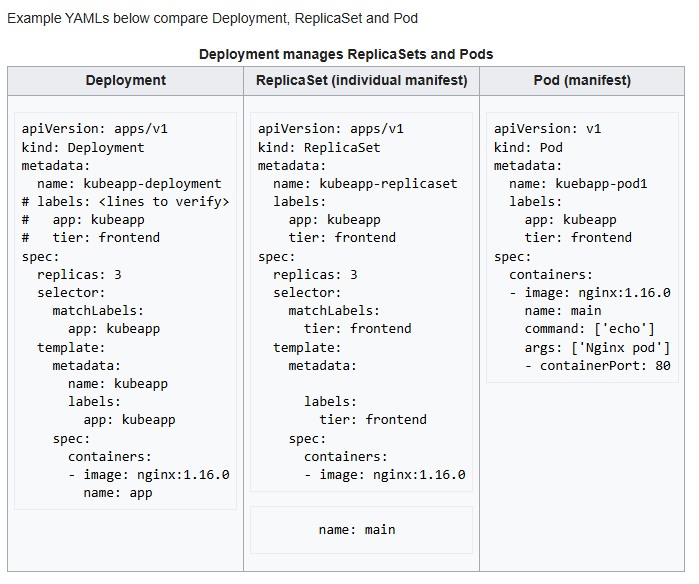
Ans: A **Deployment** in Kubernetes is a resource object that manages the deployment and **scaling** of a set of identical pods, ensuring that the specified number of pod **replicas are running** at any given time. It allows for declarative updates to applications, including **rolling updates** and **rollbacks**, ensuring the application runs smoothly and can be easily updated without downtime.

**Summary of Differences**

| **Feature** | **Pod** | **ReplicaSet** | **Deployment** |
| --- | --- | --- | --- |
| **Purpose** | Single instance of an app | Ensure a fixed number of pods | Manage ReplicaSets and updates |
| **Self-healing** | No | Yes | Yes |
| **Updates** | Manual (recreate pods) | Manual (update replicas) | Automated rolling updates |
| **Rolling Updates** | No | No | Yes |
| **Rollback** | No | No | Yes |
| **Declarative Config** | Basic | Intermediate | Advanced |
| **Use Case** | Basic unit of deployment | Maintain desired pod count | Full application lifecycle mgmt |



1. Replica set acts like a controller which controls the pod replicas(Auto-Healing)



**10.** **What is service? And problems without service and Solutions with service?.**

Ans: A Service in Kubernetes is an abstraction that defines a logical set of Pods and a policy to access them, providing a stable endpoint for communication. It enables load balancing and ensures reliable access to the Pods, even as they are scaled up or down.

**Problem 1**: Kubernetes Deployments can implement the auto-healing capability for pod that goes down using Replica Sets but whenever the new pod comes up, It comes up with a new IP address. So how the customer or user know about this new IP address ?

**Solution**: SVC Load balancing

Users will connect to Service load balancer instead of connecting to pod IPs. So that even there is change in the pod IPs, users doesn’t need to bother about that as the load balancer will tc of that.

**Problem 2**: Now, same as like users, how the load balancer knows the newly created IPs? If you application requires multiple replicas of a pod to serve multi concurrent users, each replica of a pod has unique IP address but customers or users would need a one common IP address or DNS. Just like we all access google.com(DNS) or 8.8.8.8(IP Address) to access google.

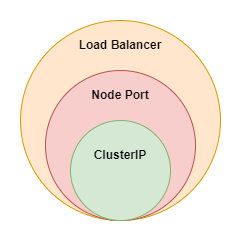
**Solution**: SVC Discovery (Using Lables and selectors)

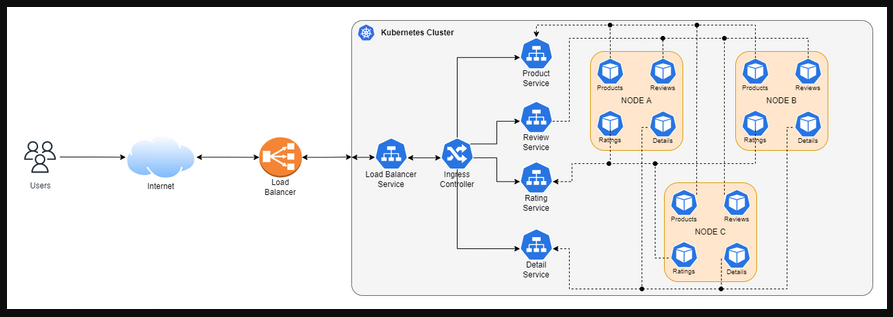
Now, We will assing the labels to the pods which are created under same replica sets. So that, even one pod go down and another pod created with new IP, the loadBalacer will connect to lable of that Pod instead of the IP, as the pods will create with the same lable only.

**Problem 3**: How are the applications deployed as Pods in Kubernetes are accessed by external or people from other teams in your organization who don't have access to the Kubernetes cluster ?

**Solution:**SVC expose to external world

1. ClusterIP: Exposes the Service on a cluster-internal IP. This type of Service makes the application accessible only within the Kubernetes cluster. It is the default Service type.
2. NodePort: Exposes the Service on each Node’s IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. This type of Service allows external access to the application using <NodeIP>:<NodePort>.
3. LoadBalancer: Exposes the Service externally using a cloud provider's load balancer. This type creates a ClusterIP and NodePort Service, and also provisions a load balancer which routes external traffic to the NodePort.
4. ExternalName: Maps the Service to the contents of the externalName field (e.g., foo.bar.example.com), returning a CNAME record with its value. This type of Service doesn't create any proxying or forwarding; it simply returns the CNAME record.





**11.** **Step by Step procedure on how the service(ClusterIP, NodePort, LoadBalancer) will be created?**

Once we have Docker file, Code, requirements.txt files in github**🡺**cloned to kubernetes cluster**🡺**Build the image using docker file🡺create deploy.yaml file using the created image🡺kubectl apply -f deploy.yaml🡺Now pods started running and we can ssh into the cluster and run "**curl http://<Ip of pod>:<port>"** can access application..

Now, without ssh to the cluster, if we try to access the pod/application using "**curl http://<Ip of pod>:<port>**" in the controlPlane(out side the cluster), we don't get anything. because by default the pods will create with control IP which is not possible to access from outside of the cluster.

**Now here, NodePort and LoadBalancer will come into picture**

Now create **service.yaml** file with contents like serviceName, type of service(Here it is NodePort) and importently name of the "**pod selector(Not the deploy lable/selector**", and the required port details and run **kubectl apply -f service.yaml**

By using "kubeclt get svc" we will get Cluster-IP and Ports details.

Now we cannect to the application using "**curl http://<nodeIP/EC2Ip if in cloud>:<NodePort which we mentioned in SVC.yml file>”.** We can access from the browser aswell. But remember this can be access only within the organization(i.e within the EC2 or cluster) as we still not yet configured LoadBalancer to access from outside of the org.

Now, To make the application accessable from outside world, Just change the type of mode from **"NodePort" to "LoadBalancer"** and run "kubectl apply -f service.yaml". EXTERNAL-IP will be created and can check using "kubectl get svc". Using the external IP, now everyone can access the application. Here, the external IP is controled by "clound controller manager if it is in cloud".

\*Pod selector name can be copied from template field in deploy.yaml file

\*using KubeShark, we can see/debug the traffic flow in k8s

**12.** **What is ingress? Services with and without ingress?**

Ans: Ingress in Kubernetes is an API object that manages external access to services within a cluster, typically via HTTP/HTTPS. It provides advanced routing capabilities, such as host- and path-based routing, SSL/TLS termination, and load balancing.

**Before Ingress**? Although k8s have services like 1. LoadBalancer 2. Expose to outside 3. Service Discoverability, It still have some drawbacks like below

* K8s SVC loadbalancer offers only round robin concept and it doesn't have capabilities of enterprise level loadbalancers like, traffic routing based on RoundRobin, Domine based, StickyMethod, Blacklisting IPs, Whilelisting IPs, Ratio Based....etc
* If we have 1000 services, cloud provider will charge for 1000 LoadBalancer IPs.

**After Ingress:**

This will provide different loadbalancing capabilities and expose a single IP to the outside, and route the traffic inside using route table/DNS

Use Cases:

1. When you need advanced routing capabilities, such as host-based or path-based routing.
2. When you want to manage SSL/TLS termination centrally.
3. When you need to handle more sophisticated traffic management, including load balancing, redirection, and URL rewrites.

**Different loadbalancer algorithms or strategies:**

1. **Round Robin**: Ideal for evenly distributing traffic when all servers are equally capable.
2. **Least Connections**: Best for handling highly variable workloads where servers may finish tasks at different rates.
3. **Weighted Round Robin/Least Connections**: Suitable when backend servers have varying capacities, ensuring more powerful servers receive proportionately more traffic.
4. **IP Hash**: Useful for session persistence without cookies, such as in stateful applications.
5. **Path-Based Routing**: Essential for microservices architectures where different services handle different parts of the application.
6. **Host-Based Routing**: Effective for multi-tenant environments or hosting multiple applications/domains on the same infrastructure.
7. **Random**: Simple but not commonly used due to lack of predictability.

**Different loadbalancer types:**

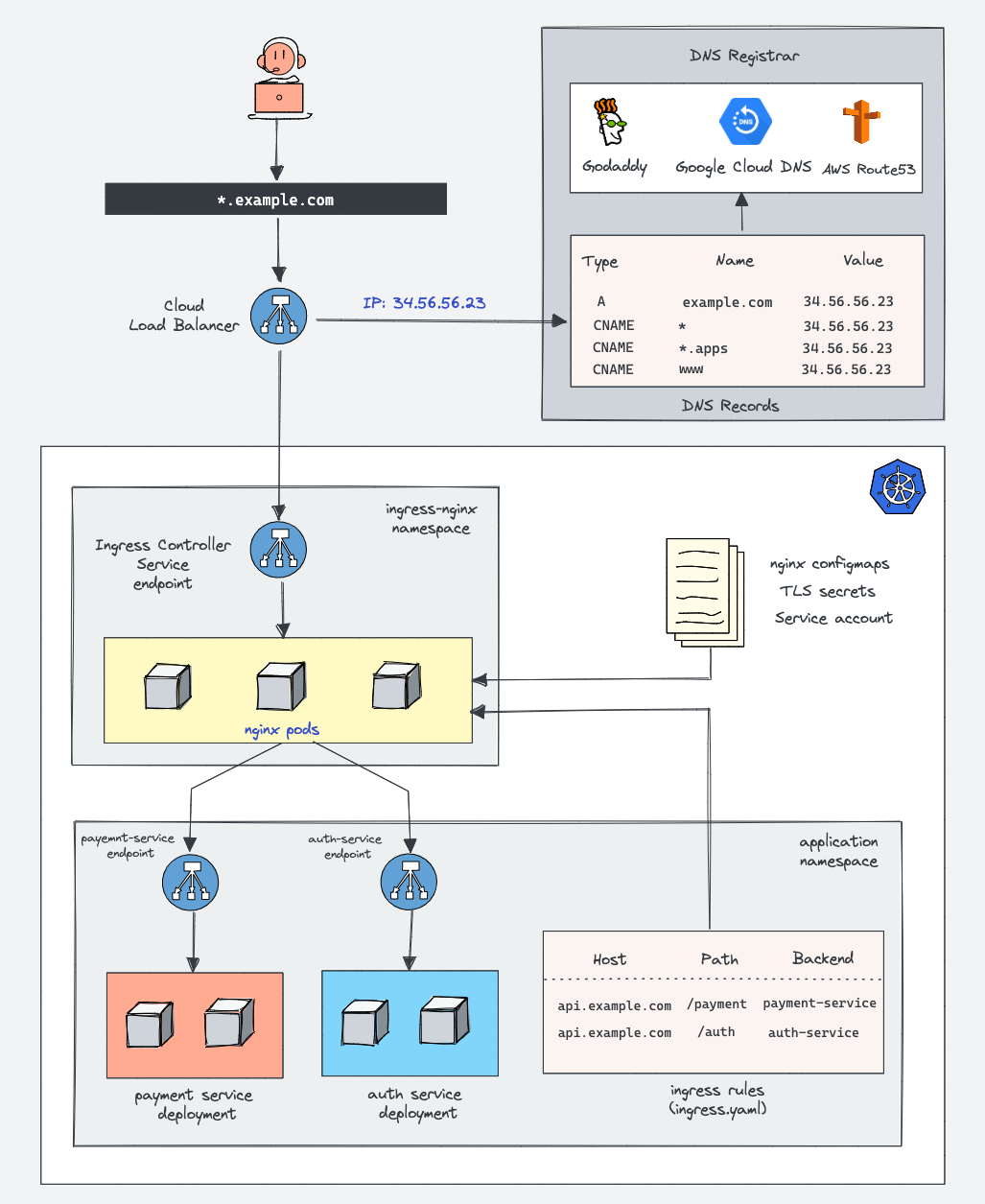
1. **Hardware Load Balancers**: Dedicated physical devices for high-performance environments.(F5 Big-IP, Citrix ADC (formerly NetScaler).
2. **Software Load Balancers**: Flexible, software-based solutions for various deployments (HAProxy, NGINX, Apache HTTP Server with mod\_proxy)
3. **Cloud Load Balancers**: Integrated cloud services offering seamless scalability and management (AWS Elastic Load Balancer (ELB), Google Cloud Load Balancer, Azure Load Balancer)
4. **Application Load Balancers (Layer 7):** Content-based routing for HTTP/S traffic (AWS Application Load Balancer (ALB), NGINX, Traefik)
5. **Network Load Balancers (Layer 4):** High-performance routing for TCP/UDP traffic (AWS Network Load Balancer (NLB), HAProxy in Layer 4 mode)
6. **Global Load Balancers**: Traffic distribution across multiple geographic locations (Google Cloud HTTP(S) Load Balancer, AWS Global Accelerator)
7. **Internal Load Balancers**: Traffic distribution within a private network (AWS Internal ELB, Google Cloud Internal Load Balancer)
8. **DNS Load Balancers**: Simple, DNS-based traffic distribution (AWS Route 53, Azure Traffic Manager, Google Cloud DNS)

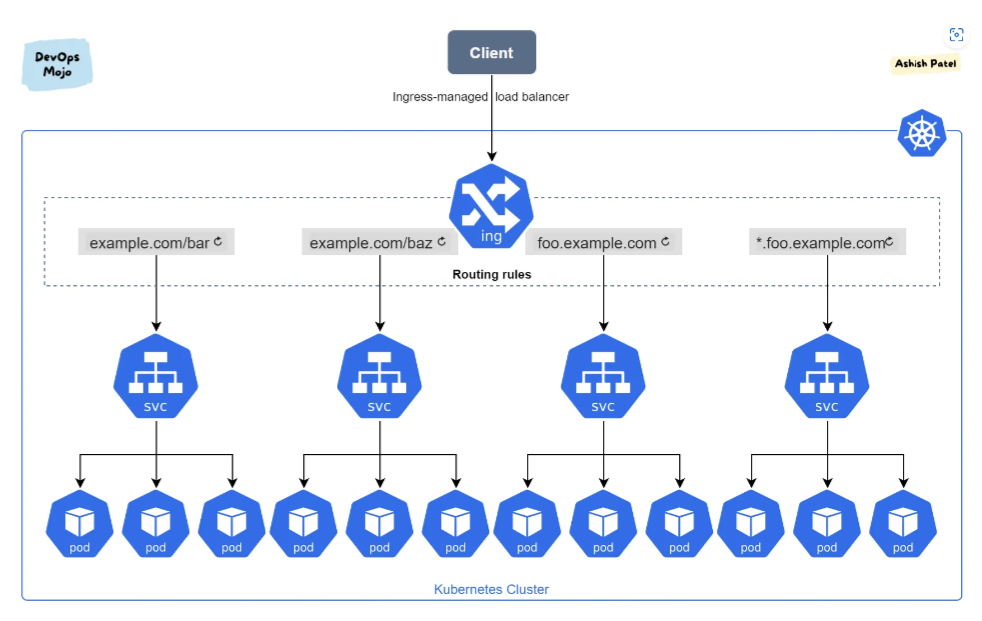
**Popular Load Balancer companies**

F5 BigIP, HAProxy, Citrix ADC, Nginx, AWS ELB, Azure LB, GCP LB

Points to Note:

* We need ingress controller to manage ingress in k8s
* Ingress controllers can sit inside the cluster as a pod, and can be placed outside the cluster as well when it is not possible to be containerised
* After creating the services using service.yml, we will get a service name. That service names will be configued in ingress.yml file and after run the ingress.yml using “kubectl apply –f ingress.yml” we will get the IP address to access the services from outside.
* To install a ingress controller, there are number of loadbalancers in the market like F5, HAProxy, AWS ELB….other than cloud LB, we can install others from their git repos.

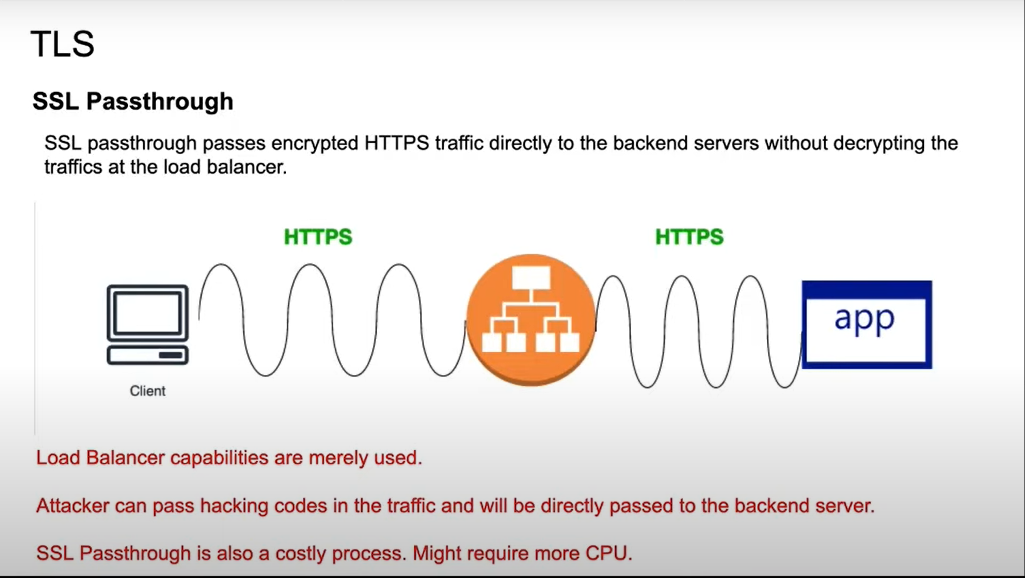




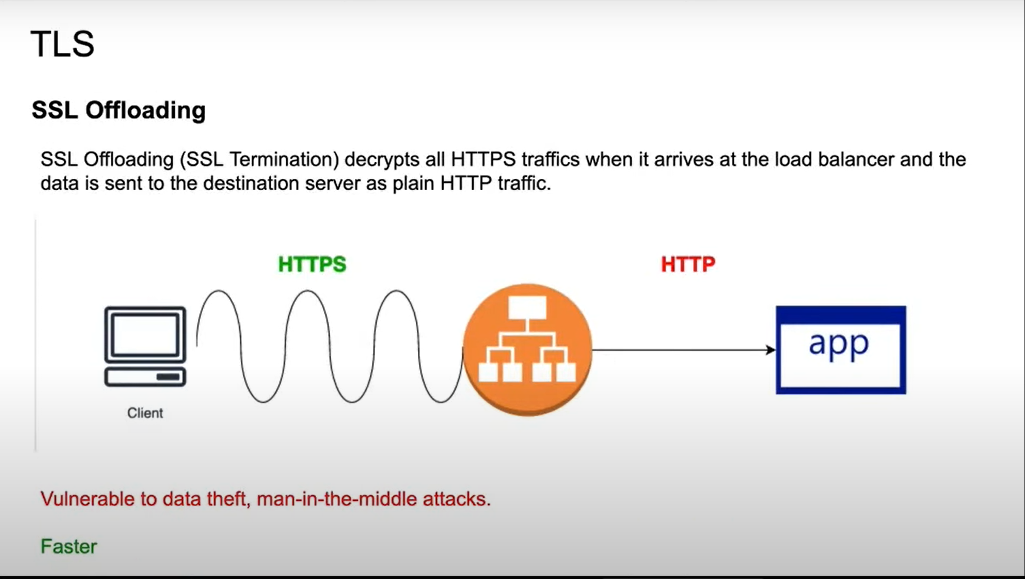
**13.INGRESSTLS methods(Secure ingress).**

Ans: 3 methods. SSL PASS THROUGH, SSL OFF-LOADING, SSL BRIDGING

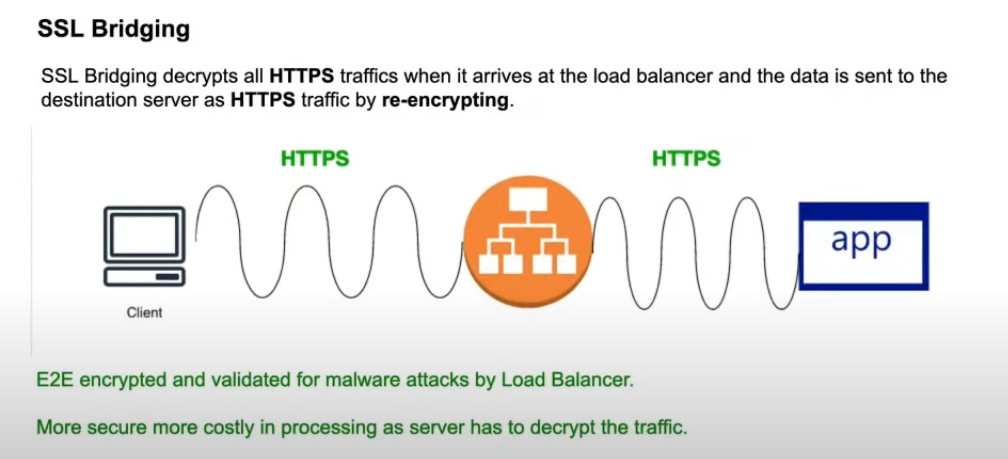
**SSL PASS THROUGH:**



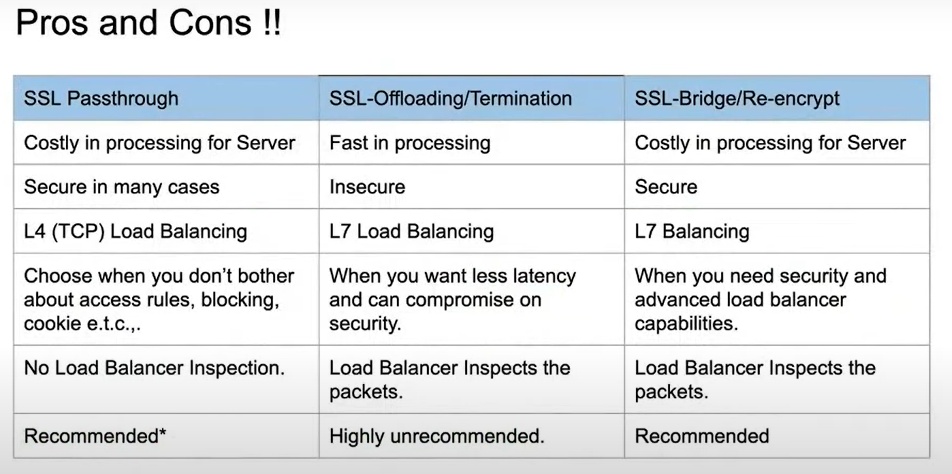
**SSL Offloading**



**SSL Bridging**



**SSL PASS THROUGH** Vs **SSL OFF-LOADING** Vs **SSL BRIDGING**

****

**14. RBAC(Role based access control):**

To manage clusters, pods and services in kubernetes, we need a user management for the security purpose.

* Kubernetes won’t manage roles, instead it will offload the roles to external identity providers(Similar to login with google, login with facebook….etc) like LDAP, KeyClock, OKTA, SSO, Github.
* For example, In case we are using kubernetes in AWS EKS, AWS can manage users via IAM.

Here we are learning 2 types of access managements. 1. Users/user groups 2. Service accounts for PODS

2. Service Accounts:

\*\*\*Here, Not like everywhere, service accounts are not meant for users. SA are meant for the access management to the services running in k8s. In k8s, when there is a pod created, by default service account will be created and it associated with respective pod. As part of the application the pod may read secrets, may access config maps.

\*\*If the pod is malicious pod, then it may delete something in the nodes(like API calls/services).

\*\*Using YAML, we have to create SA in k8s and we can manage the rules.

1. Users/User groups:

Users will be defines via other identity providers as said above. We can assign the roles like developer access, QA access based on the requirement.

**Role/Cluster Role---**&---**Role binding/Cluster Role binding:**

Users are created. Now we need to define some roles. So that whoever get assinged to the defined roles, they will get the appropriate/respective permissions.

Now, users&SAs are ready and roles are ready. To attach both, we need a bridge/mechanism and it is called role binding.

Users/Service Accounts

Roles/Cluster Roles

Role binding/Cluster Role binding

**15.** **Custom Resources | Custom Controllers | Custom resource Definition**

Custom Resources (CRs) are extensions of the Kubernetes API that allow you to introduce new resource types into the Kubernetes ecosystem. They enable you to define and manage application-specific or domain-specific objects that are not included in the default Kubernetes API. This is useful for managing the lifecycle of applications and components in a Kubernetes-native way.

Write a controller to watch for changes to instances of your custom resource and handle the business logic.

**Key Concepts of Custom Resources:**

1. **Custom Resource Definitions (CRDs):**

* CRDs are used to define new types of custom resources. They describe the schema and validation for your custom resource.
* When you create a CRD, Kubernetes' API server gains new endpoints for creating, reading, updating, and deleting instances of the custom resource type.
* Create a **YAML file to define the CRD**, specifying the name, version, and schema of the custom resource.
* Use kubectl to apply the CRD. (**kubectl apply -f myresource-crd.yaml**)

1. **Custom Controllers:**(https://github.com/kubernetes/sample-controller)

* Custom Controllers are often used alongside CRDs to handle the logic for managing the state of custom resources. They watch for changes to custom resources and react accordingly, usually by creating or managing other Kubernetes resources.
* These controllers follow the Kubernetes operator pattern, encapsulating operational knowledge in code.

1. **Operators:**

* An Operator is a Kubernetes pattern combining CRDs and custom controllers to manage complex applications. Operators leverage Kubernetes' reconciliation loop to ensure that the desired state of a custom resource matches the actual state.

🡺🡺🡺There are many external custom resources created and maintained by the Kubernetes community and various organizations.**Ex: Prometheus Operator, Argo CD, Istio**

Most operators can be installed using **Helm, OperatorHub, or custom installation scripts** provided by the maintainers.

**Finding and Using Operators:**

* **OperatorHub.io**: A central repository for finding Kubernetes Operators.
* **Helm Charts**: Many operators are available as Helm charts, simplifying installation and management.
* **GitHub**: Many open-source operators are hosted on GitHub, where you can find installation instructions and documentation.

**16.** **CONFIGMAPS & SECRETS:**

In Kubernetes, ConfigMaps and Secrets are used to manage configuration data and sensitive information, respectively. They allow you to decouple configuration artifacts from image content to keep containerized applications portable.

**ConfigMaps:**

ConfigMaps are used to store non-sensitive data in key-value pairs. They can be used to configure application settings and other non-sensitive information.

**Creating a ConfigMap:**

**1.** From Literal Values:

**kubectl create configmap my-config --from-literal=key1=value1 --from-literal=key2=value2**

**2.** From a File:

**kubectl create configmap my-config --from-file=config.properties**

**3.** From a Directory:

**kubectl create configmap my-config --from-file=path/to/directory/**

**Using ConfigMaps:**

**1.** As Environment Variables:

2. As Volume Mounts:

**Secrets:**

Secrets are used to store sensitive data, such as passwords, OAuth tokens, and SSH keys, in a base64-encoded format. Kubernetes manages Secrets securely and can provide them to containers in a controlled manner.

**Creating a Secret:**

**1.** From Literal Values:

**kubectl create secret generic my-secret --from-literal=username=myuser --from-literal=password=mypassword**

**2.** From a File:

**kubectl create secret generic my-secret --from-file=ssh-privatekey=/path/to/private.key**

**3.** From Environment Variables:

**kubectl create secret generic my-secret --from-env-file=path/to/envfile**

**Using Secrets**

1. As Environment Variables:

2. As Volume Mounts:

**Differences Between ConfigMaps and Secrets**

* **Purpose:** ConfigMaps are for non-sensitive data; Secrets are for sensitive data.
* **Encoding:** Secrets data is base64-encoded, while ConfigMaps store data in plain text.
* **Security:** Access to Secrets is controlled more tightly than ConfigMaps, and Secrets can be used with features like encryption at rest.

**Best Practices**

1. **Least Privilege:** Limit access to Secrets and ConfigMaps using RBAC policies.
2. **Encryption:** Use encryption at rest for Secrets.
3. **Avoid Plain Text:** Avoid storing sensitive information in ConfigMaps.
4. **Version Control:** Do not store Secrets in version control systems.

By using ConfigMaps and Secrets, you can manage configuration and sensitive information in a secure and efficient manner in your Kubernetes clusters.

**17.** **MONITORING USING PROMETHEUS & GRAFANA**

**It all starts with Monitoring**

Monitoring your Kubernetes cluster is essential for ensuring the health and performance of your applications and infrastructure. Here are some reasons why monitoring your Kubernetes cluster is important:

* Identify issues and troubleshoot: By monitoring your Kubernetes cluster, you can quickly identify issues such as application crashes, resource bottlenecks, and network problems. With real-time monitoring, you can troubleshoot issues before they escalate and impact your users.
* Optimize performance and capacity: Monitoring allows you to track the performance of your applications and infrastructure over time, and identify opportunities to optimize performance and capacity. By understanding usage patterns and resource consumption, you can make informed decisions about scaling your infrastructure and improving the efficiency of your applications.
* Ensure high availability: Kubernetes is designed to provide high availability for your applications, but this requires careful monitoring and management. By monitoring your cluster and setting up alerts, you can ensure that your applications remain available even in the event of failures or unexpected events.
* Security and compliance: Monitoring your Kubernetes cluster can help you identify potential security risks and ensure compliance with regulations and policies. By tracking access logs and other security-related metrics, you can quickly detect and respond to potential security threats.

**Using Prometheus for monitoring**

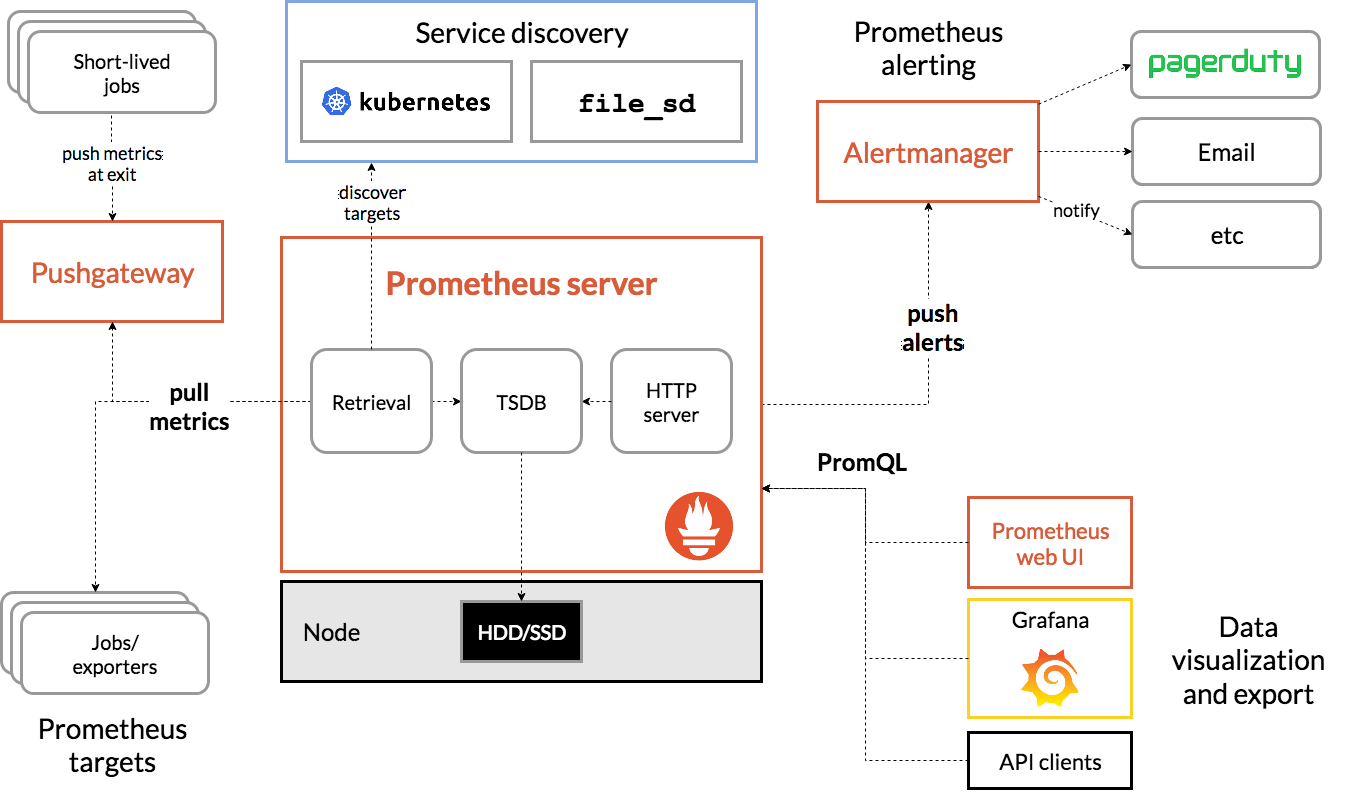
Prometheus is an open-source monitoring and alerting system that helps you collect and store metrics about your software systems and infrastructure, and analyze that data to gain insights into their health and performance. It provides a powerful query language, a flexible data model, and a range of integrations with other tools and systems. With Prometheus, you can easily monitor metrics such as CPU usage, memory usage, network traffic, and application-specific metrics, and use that data to troubleshoot issues, optimize performance, and create alerts to notify you when things go wrong.

**Why Prometheus over other monitoring tools ?**

Prometheus is a popular choice for Kubernetes monitoring for several reasons:

* Open-source: Prometheus is an open-source project that is free to use and has a large community of contributors. This means that you can benefit from ongoing development, bug fixes, and feature enhancements without paying for a commercial monitoring solution.
* Native Kubernetes support: Prometheus is designed to work seamlessly with Kubernetes, making it easy to deploy and integrate with your Kubernetes environment. It provides pre-configured Kubernetes dashboards and supports auto-discovery of Kubernetes services and pods.
* Powerful query language: Prometheus provides a powerful query language that allows you to easily retrieve and analyze metrics data. This allows you to create custom dashboards and alerts, and to troubleshoot issues more easily.
* Scalability: Prometheus is designed to be highly scalable, allowing you to monitor large and complex Kubernetes environments with ease. It supports multi-node architectures and can handle large volumes of data without significant performance degradation.
* Integrations: Prometheus integrates with a wide range of other tools and systems, including Grafana for visualization, Alertmanager for alerting, and Kubernetes API server for metadata discovery.

**Prometheus Architecture**



**What is Grafana ?**

Grafana is a popular open-source data visualization and analytics platform that allows you to create custom dashboards and visualizations based on a variety of data sources. Grafana is often used for monitoring and analyzing metrics and logs in real-time, making it an ideal tool for monitoring systems and applications, including Kubernetes environments.

Grafana supports a wide range of data sources, including databases, time-series databases, and other data storage systems. It provides a powerful query language that allows you to retrieve and analyze data from these sources, and to create custom dashboards and alerts based on that data.

In addition to its powerful data visualization and analysis capabilities, Grafana is also highly extensible. It supports a wide range of plugins and integrations, including integrations with popular monitoring and logging tools like Prometheus, Elasticsearch, and InfluxDB.

**18.** **STEP BY STEP SET-UP OF PROMETHEUS AND GRAFANA**

**Step 1: Access Grafana**

1. **Get the Grafana service details:**

kubectl get svc -n <grafana-namespace>

Find the Grafana service name (typically named grafana).

1. **Access Grafana:**
   * If Grafana is exposed via a LoadBalancer or NodePort, use the external IP or Node IP and port to access it.
   * If Grafana is exposed via an Ingress, use the Ingress hostname.

For example, if it's exposed via a LoadBalancer:

kubectl get svc -n <grafana-namespace>grafana

Then access Grafana at http://<external-ip>:<port>.

**Step 2: Log in to Grafana**

1. Open your web browser and go to the Grafana URL obtained in the previous step.
2. Log in with the default credentials (username: admin, password: admin). You might be prompted to change the password after the first login.

**Step 3: Add Prometheus as a Data Source**

1. In Grafana, click on the gear icon (⚙) in the left-hand menu to go to the **Configuration** section.
2. Click on **Data Sources**.
3. Click the **Add data source** button.
4. Select **Prometheus** from the list of available data sources.
5. Configure the Prometheus data source:
   * **Name**: Give your data source a name (e.g., Prometheus).
   * **URL**: Enter the Prometheus service URL. If Prometheus is running in the same Kubernetes cluster, you can use the service name (e.g., http://prometheus:9090).
   * **Access**: Select Server (default).
6. Click the **Save & Test** button to verify the configuration.

**Step 4: Create a Dashboard**

1. Click on the **Create** (plus icon) in the left-hand menu.
2. Select **Dashboard**.
3. Click **Add new panel** to create a new visualization.

**Step 5: Configure the Panel**

1. **Select the Prometheus data source** from the dropdown menu.
2. **Write a PromQL query** to retrieve the data you want to visualize. For example, to get the CPU usage, you might use:rate(node\_cpu\_seconds\_total{mode!="idle"}[5m])
3. **Customize the visualization:**
   * Choose the visualization type (Graph, Gauge, Table, etc.).
   * Configure the panel options (title, axes, legend, etc.).
4. Click **Apply** to save the panel.

**Step 6: Save the Dashboard**

1. Click the **Save dashboard** (disk icon) at the top.
2. Provide a name for your dashboard and click **Save**.

**Step 7: Access and Share the Dashboard**

1. Your dashboard is now ready and can be accessed from the **Dashboards** menu.
2. You can share the dashboard by clicking the share icon at the top of the dashboard view.

By following these steps, you will have a Grafana dashboard that visualizes data from Prometheus, allowing you to monitor and analyze your Kubernetes cluster metrics effectively.

**How to retrieve the Admin password of grafana**

To get the admin password for Grafana installed in a Kubernetes cluster, you typically need to check the Kubernetes secrets or configurations where Grafana's credentials are stored. Here are the steps to retrieve the Grafana admin password:

* **Identify the Namespace**

Determine the namespace where Grafana is installed. For example, it might be in a namespace called monitoring.

* **Get the Grafana Secret**

Grafana usually stores the admin credentials in a Kubernetes secret. You can list the secrets in the Grafana namespace: kubectl get secrets -n <grafana-namespace>

Look for a secret that likely contains the Grafana credentials. It might have a name like grafana, grafana-admin, or something similar.

* **Retrieve the Admin Password**

Once you identify the secret, you can retrieve the admin password. For example, if the secret name is grafana:

kubectl get secret grafana -n <grafana-namespace> -o jsonpath="{.data.admin-password}" | base64 --decode ; echo

This command will decode and print the admin password.

**Example**

If Grafana is installed in the namespace monitoring and the secret name is grafana:

kubectl get secret grafana -n monitoring -o jsonpath="{.data.admin-password}" | base64 --decode ; echo

**Additional Information**

If you used a Helm chart to install Grafana, the credentials might be set during the installation. You can check the Helm release values:

helm get values <release-name> -n <grafana-namespace>

Replace <release-name> with the name of your Helm release.

By following these steps, you should be able to retrieve the Grafana admin password and use it to log in to the Grafana web interface.

**19.** **Daemon and DaemonSet Concepts in Kubernetes**

In computing, a daemon is a background process that runs continuously and typically handles periodic service requests. The concept in Kubernetes revolves around ensuring that certain workloads run on all or specific nodes in a cluster.

### DaemonSet in Kubernetes

A DaemonSet is a Kubernetes resource that ensures that a copy of a pod is running on all (or some) nodes in the cluster. This is useful for tasks that need to be carried out on every node, such as logging, monitoring, or networking services.

#### Key Characteristics of DaemonSet

1. **One Pod Per Node**: Ensures that a single pod is running on each node.
2. **Automatic Updates**: When nodes are added to the cluster, pods are automatically added to the new nodes. Conversely, when nodes are removed, pods are garbage collected.
3. **Selective Nodes**: You can target specific nodes using node selectors, affinities, and tolerations.

#### Common Use Cases

* **Node Monitoring**: Deploying monitoring agents like Prometheus Node Exporter.
* **Log Collection**: Deploying log collection agents like Fluentd or Logstash.
* **Network Management**: Running networking components like CNI plugins or DNS.

### Creating a DaemonSet

Here’s an example of a basic DaemonSet YAML file that deploys a pod running a simple container on every node in the cluster:

yaml

Copy code

apiVersion:apps/v1

kind:DaemonSet

metadata:

name:my-daemonset

labels:

app:my-daemonset

spec:

selector:

matchLabels:

app:my-daemonset

template:

metadata:

labels:

app:my-daemonset

spec:

containers:

-name:my-daemonset-container

image:k8s.gcr.io/pause:3.1

### Detailed Concepts

1. **Node Affinity and Anti-Affinity**:
   * **Node Affinity** allows you to constrain which nodes your DaemonSet pods can be scheduled on, using labels on nodes.
   * **Node Anti-Affinity** allows you to prevent pods from being scheduled on specific nodes.
2. **Tolerations and Taints**:
   * **Tolerations** allow the DaemonSet pods to be scheduled on nodes with specific taints.
   * **Taints** can be applied to nodes to prevent certain pods from being scheduled on them unless the pod has a matching toleration.
3. **Rolling Updates**:
   * DaemonSets support rolling updates, allowing for updates to the Daemon Set’s pods without downtime.
4. **Update Strategy**:
   * You can define the update strategy for the DaemonSet, such as RollingUpdate or OnDelete.

### Example with Node Selector and Tolerations

Here’s an example of a DaemonSet that uses node selectors and tolerations:

*apiVersion:apps/v1*

*kind:DaemonSet*

*metadata:*

*name:my-daemonset*

*labels:*

*app:my-daemonset*

*spec:*

*selector:*

*matchLabels:*

*app:my-daemonset*

*template:*

*metadata:*

*labels:*

*app:my-daemonset*

*spec:*

*nodeSelector:*

*disktype:ssd*

*tolerations:*

*-key:"key1"*

*operator:"Equal"*

*value:"value1"*

*effect:"NoSchedule"*

*containers:*

*-name:my-daemonset-container*

*image:k8s.gcr.io/pause:3.1*

### Monitoring and Managing DaemonSets

* **kubectl get daemonset**: View DaemonSets in your cluster.
* **kubectl describe daemonset <name>**: Get detailed information about a DaemonSet.
* **kubectl delete daemonset <name>**: Delete a DaemonSet, which will also remove its pods.

By using DaemonSets, you ensure that a particular service runs on every node in your cluster, or a selected subset of nodes, providing consistent and reliable node-level services.

**Kubernetes Troubleshooting**

**1.** **Image pull backoff(**ErrImagePull**)**

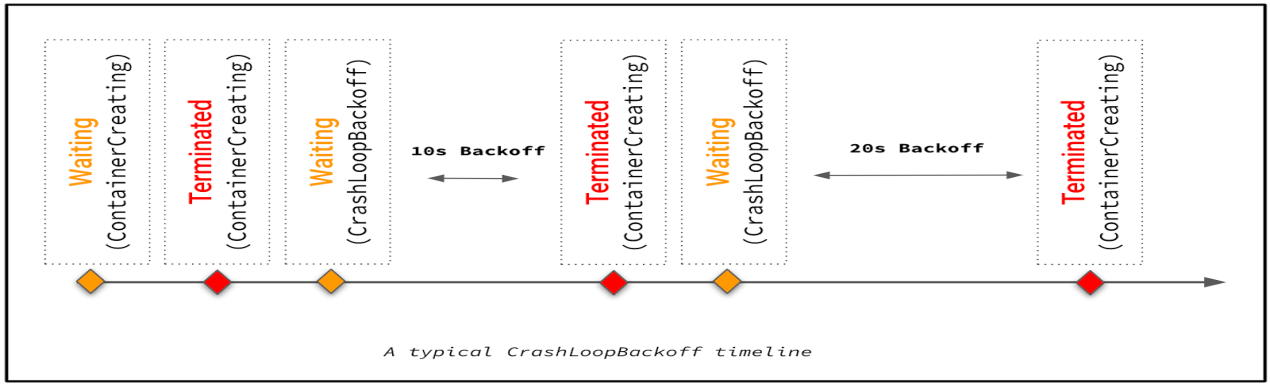
When a kubelet starts creating containers for a Pod using a container runtime, it might be possible the container is in Waiting state because of ImagePullBackOff.

The status ImagePullBackOff means that a container could not start because Kubernetes could not pull a container image for reasons such as

* **Invalid image name or**
* **Pulling from a private registry without imagePullSecret.**

The BackOff part indicates that Kubernetes will keep trying to pull the image, with an increasing back-off delay.

Kubernetes raises the delay between each attempt until it reaches a compiled-in limit, which is 300 seconds (5 minutes).



**2.** **CrashLoop BackOff**

**CrashLoopBackOff** is a common error state in Kubernetes (K8s) that indicates a pod is failing and restarting repeatedly. The term "CrashLoopBackOff" means that Kubernetes is backing off from restarting the pod due to the frequent crashes.

#### Common Causes

1. **Application Bugs**: Bugs in the application code, such as unhandled exceptions or segmentation faults, can cause the application to crash. For instance, if the application tries to access a null pointer or fails to catch and handle an exception correctly, it might terminate unexpectedly. Kubernetes, detecting the crash, will restart the container, but if the bug is triggered each time the application runs, this leads to a repetitive crash loop.
2. **Resource Constraints[OOMKILLED]**: If the memory limits set for a container are too low, the application might exceed this limit, especially under load, leading to the container being killed by Kubernetes. This can happen repeatedly if the workload does not decrease, causing a cycle of crashing and restarting. Kubernetes uses these limits to ensure that containers do not consume all available resources on a node, which can affect other containers.Here, CrashLoop Backoff is status and OOMKILLED is the error. When we use kubectl get pods, will see crashloop backoff status, and kubectl describe pod, then we will see OOMKILLED error
3. **Configuration Errors**: Misconfigurations can encompass a wide range of issues, from incorrect environment variables to improper setup of service ports or volumes. These misconfigurations can prevent the application from starting correctly, leading to crashes. For example, if an application expects a certain environment variable to connect to a database and that variable is not set or is incorrect, the application might crash as it cannot establish a database connection.
4. **Wrong Command Line Arguments:**Containers might be configured to start with specific command-line arguments. If these arguments are wrong or lead to the application exiting (for example, passing an invalid option to a command), the container will exit immediately. Kubernetes will then attempt to restart it, leading to the CrashLoopBackOff status. An example would be passing a configuration file path that does not exist or is inaccessible.
5. **Dependency Issues**: Missing or incorrectly configured dependencies such as databases, services, or secrets.
6. **Errors in the Liveness Probes**: Liveness probes in Kubernetes are used to check the health of a container. If a liveness probe is incorrectly configured, it might falsely report that the container is unhealthy, causing Kubernetes to kill and restart the container repeatedly. For example, if the liveness probe checks a URL or port that the application does not expose or checks too soon before the application is ready, the container will be repeatedly terminated and restarted.
7. **Network Issues**: Network failures or misconfigurations can prevent the application from communicating with required services.

#### Diagnosis

1. **Check Pod Logs**:
   * Use kubectl logs to view the logs of the failing container:

*kubectl logs <pod-name> -n <namespace>*

* + For multi-container pods, specify the container name:

*kubectl logs <pod-name> -n <namespace> -c <container-name>*

1. **Describe the Pod**:
   * Use kubectl describe pod to get detailed information about the pod’s state:

*kubectl describe pod <pod-name> -n <namespace>*

* + Look for events, warnings, and error messages that can provide clues about the cause of the crashes.

1. **Inspect Resource Usage**:
   * Check if the pod is getting enough resources (CPU, memory):

*kubectl top pod <pod-name> -n <namespace>*

1. **Check for Configuration Issues**:
   * Verify environment variables, config maps, secrets, and volume mounts.
   * Ensure all dependencies and services are correctly configured and accessible.

#### Solutions

1. **Fix Application Issues**:
   * Identify and fix bugs or exceptions in the application code.
   * Ensure proper handling of dependencies and resource limits within the application.
2. **Adjust Resource Limits**:
   * Increase CPU and memory limits/requests for the pod if it is running out of resources.
   * Example:

*resources:*

*limits:*

*memory:"512Mi"*

*cpu:"500m"*

*requests:*

*memory:"256Mi"*

*cpu:"250m"*

1. **Configuration Adjustments**:
   * Correct any misconfigurations in environment variables, config maps, secrets, and volume mounts.
   * Ensure all required services are reachable and correctly configured.
2. **Health Checks**:
   * Implement proper liveness and readiness probes to handle pod health and readiness:

*livenessProbe:*

*httpGet:*

*path:/healthz*

*port:8080*

*initialDelaySeconds:3*

*periodSeconds:3*

*readinessProbe:*

*httpGet:*

*path:/ready*

*port:8080*

*initialDelaySeconds:3*

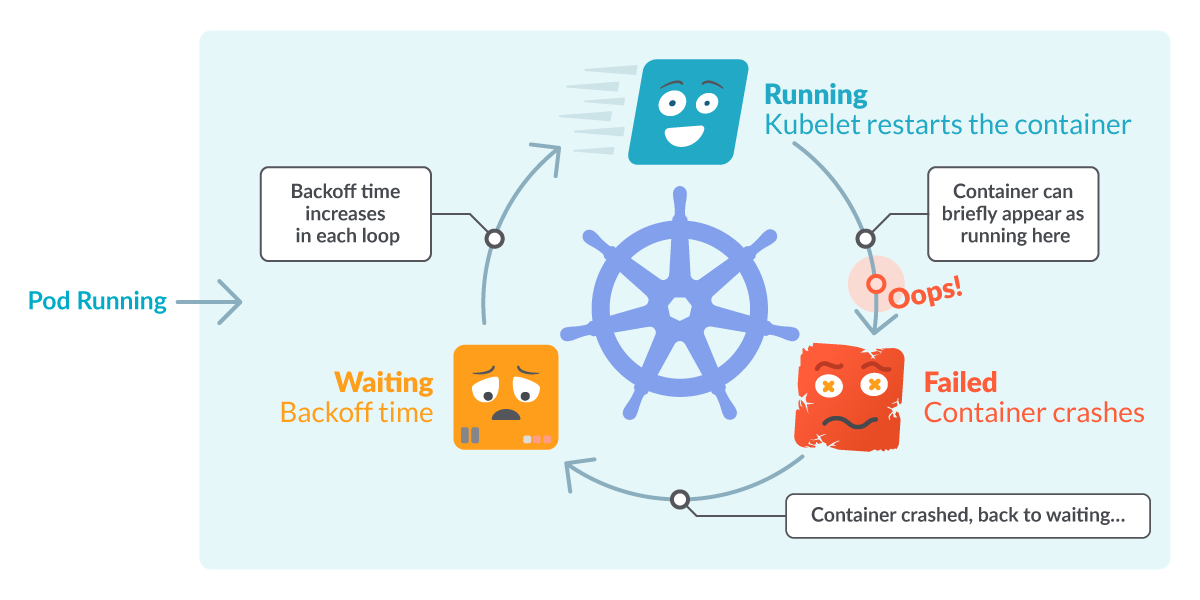
*periodSeconds:3*

1. **Retry Policy**:
   * Configure backoff and retry policies appropriately to handle transient errors without overwhelming the system:

*restartPolicy:Always*

#### Best Practices

1. **Monitoring and Logging**:
   * Implement robust monitoring and logging to detect and diagnose issues promptly.
   * Use tools like Prometheus, Grafana, and ELK stack for monitoring and logging.
2. **Graceful Shutdown**:
   * Ensure your application handles termination signals and shuts down gracefully to avoid data corruption and ensure a clean state on restart.
3. **Version Control and Rollbacks**:
   * Keep track of changes and be ready to roll back to a previous stable version if a new deployment causes crashes.



**3.** **Issues with Scheduler**

In Kubernetes, the scheduler is responsible for assigning pods to nodes in the cluster based on various criteria. Sometimes, you might encounter situations where pods are not being scheduled as expected. This can happen due to factors such as node constraints, pod requirements, or cluster configurations.

1. **Node Selector**
2. **Node Affinity**
3. **Taints**
4. **Tolerations**

### Node Selector

* **Definition**: A field in Kubernetes Pod specification to specify that a pod should only be scheduled on nodes with a specific label.
* **Use Case**: Ensuring certain applications run only on nodes with specific characteristics (e.g., ARM processor).
* **Example**:
  + Kubernetes deployment file with a node selector field:

*nodeSelector:*

*nodeName: arm-worker*

* + Without a matching node label, the pod remains unschedulable.

### Node Affinity

* **Difference from Node Selector**: Provides more flexibility with two options - required and preferred.
* **Required During Scheduling**: Similar to node selector, it enforces strict matching.
* **Preferred During Scheduling**: Allows for a preferred node match but will schedule on any available node if no match is found.
* **Example**:
  + Preferred scheduling:

*affinity:*

*nodeAffinity:*

*preferredDuringSchedulingIgnoredDuringExecution:*

*- weight: 1*

*preference:*

*matchExpressions:*

*- key: nodeName*

*operator: In*

*values:*

*- arm-worker*

* + Required scheduling:

*affinity:*

*nodeAffinity:*

*requiredDuringSchedulingIgnoredDuringExecution:*

*nodeSelectorTerms:*

*- matchExpressions:*

*- key: nodeName*

*operator: In*

*values:*

*- arm-worker*

### Taints

* **Definition**: Used to make a node unschedulable for certain pods.
* **Use Case**: Upgrading nodes, backups, handling nodes with issues.
* **Types**:
  + NoSchedule: Pods will not be scheduled on the node.
  + PreferNoSchedule: Scheduler avoids but does not strictly prevent scheduling.
  + NoExecute: Existing pods are evicted, and new pods are not scheduled.
* **Example**:

*kubectl taint nodes <node-name> key=value:NoSchedule*

### Practical Demonstration

1. **Creating a Kubernetes Cluster**:
   * Use kind or k3d for creating lightweight multi-node clusters.
   * Commands to create clusters:

*kind create cluster --name mycluster*

* + Check nodes:

*kubectl get nodes*

1. **Implementing Node Selector**:
   * Add a label to the node:

*kubectl label nodes <node-name> nodeName=arm-worker*

* + Deployment file with node selector:

*apiVersion: apps/v1*

*kind: Deployment*

*metadata:*

*name: nginx-deployment*

*spec:*

*replicas: 3*

*selector:*

*matchLabels:*

*app: nginx*

*template:*

*metadata:*

*labels:*

*app: nginx*

*spec:*

*containers:*

*- name: nginx*

*image: nginx*

*nodeSelector:*

*nodeName: arm-worker*

1. **Using Node Affinity**:
   * Preferred scheduling:

*affinity:*

*nodeAffinity:*

*preferredDuringSchedulingIgnoredDuringExecution:*

*- weight: 1*

*preference:*

*matchExpressions:*

*- key: nodeName*

*operator: In*

*values:*

*- arm-worker*

* + Required scheduling:

*affinity:*

*nodeAffinity:*

*requiredDuringSchedulingIgnoredDuringExecution:*

*nodeSelectorTerms:*

*- matchExpressions:*

*- key: nodeName*

*operator: In*

*values:*

*- arm-worker*

1. **Applying Taints**:
   * Apply taint to a node:

*kubectl taint nodes <node-name> key=value:NoSchedule*

* + Check the effect of taints using describe command:

*kubectl describe nodes <node-name>*

### Conclusion

* Node Selector, Node Affinity, Taints, and Tolerations are crucial concepts for effective Kubernetes cluster management.
* Proper implementation prevents unschedulable states and ensures optimal performance and maintenance.
* Recommended to use multi-node clusters for practice, utilizing tools like kind for ease and efficiency.

### What are Tolerations?

Tolerations allow the scheduler to place pods on nodes with matching taints. Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes.

### Adding Tolerations to a Pod

Tolerations are added to pod specifications to allow pods to be scheduled on nodes with specific taints. Here’s an example of how to add a toleration to a pod:

*apiVersion: v1*

*kind: Pod*

*metadata:*

*name: my-pod*

*spec:*

*containers:*

*- name: my-container*

*image: my-image*

*tolerations:*

*- key: "key1"*

*operator: "Equal"*

*value: "value1"*

*effect: "NoSchedule"*

### Troubleshooting with Tolerations

When troubleshooting, understanding how tolerations work with taints can help diagnose why certain pods are not scheduled on specific nodes or why pods are getting evicted.

#### Common Issues and Steps

1. **Pod Not Scheduled on Tainted Nodes**

**Symptom**: A pod is not being scheduled on a node that has a specific taint.

**Steps**:

* + **Check Node Taints**: Determine the taints applied to the node.

*kubectl describe node <node-name>*

Look under the Taints section.

* + **Check Pod Tolerations**: Ensure that the pod has the necessary tolerations to match the node’s taints.

*kubectl describe pod <pod-name>*

Look under the Tolerations section.

* + **Match Taints and Tolerations**: Verify that the taints and tolerations match. For example, if a node has a *taint:*

*key=value:NoSchedule*

Ensure the pod has a corresponding toleration:

*tolerations:*

*- key: "key"*

*operator: "Equal"*

*value: "value"*

*effect: "NoSchedule"*

1. **Pod Evicted Due to Taints**

**Symptom**: A pod gets evicted from a node with a NoExecute taint.

**Steps**:

* + **Check Node Taints**: Look for any NoExecute taints.

*kubectl describe node <node-name>*

* + **Check Pod Tolerations**: Ensure that the pod has the necessary tolerations to tolerate the NoExecute taint.

*kubectl describe pod <pod-name>*

* + **Example Toleration for NoExecute**:

*tolerations:*

*- key: "key"*

*operator: "Equal"*

*value: "value"*

*effect: "NoExecute"*

*tolerationSeconds: 3600*

The tolerationSeconds field specifies how long the pod can stay on the node with the NoExecute taint before being evicted.

### Real-Time Scenario: Pipeline Troubleshooting with Tolerations

#### Scenario:

You have a Jenkins pipeline deploying applications to a Kubernetes cluster. A recent change introduced node taints to prevent certain workloads from being scheduled on specific nodes. However, some critical pods are not being scheduled as expected.

1. **Investigate Deployment Failure**:
   * **Jenkins Logs**: Check the pipeline logs for errors related to pod scheduling.

*kubectl get pods -n <namespace>*

1. **Check Node Taints**:
   * **Identify Tainted Nodes**:

*kubectl get nodes -o json | jq '.items[] | select(.spec.taints) | {name: .metadata.name, taints: .spec.taints}'*

1. **Check Pod Specifications**:
   * **Verify Tolerations**:

*kubectl describe pod <pod-name>*

1. **Update Pod Tolerations**:
   * **Modify Pod Spec**: Add or update tolerations in the pod spec to match the node’s taints.
2. **Redeploy and Verify**:
   * **Redeploy Application**: Trigger the Jenkins pipeline to redeploy the application.
   * **Verify Scheduling**: Ensure that the pods are scheduled on the correct nodes.

### Interview Strategy: Explaining Tolerations and Troubleshooting

When explaining tolerations and troubleshooting in an interview:

* **Define Tolerations**: Explain what tolerations are and how they interact with taints.
* **Use Examples**: Provide concrete examples of issues you faced and how you resolved them.
* **Detail the Process**: Outline your troubleshooting steps clearly, emphasizing the use of kubectl commands.
* **Emphasize Real-Time Scenarios**: Highlight real-time scenarios where you successfully identified and resolved issues related to taints and tolerations.

By incorporating tolerations into your troubleshooting explanations, you demonstrate a thorough understanding of Kubernetes scheduling mechanics and your ability to diagnose and resolve complex issues.

**4.** **Realtime Issues with Stateful Sets and Persistent Volumes in Kubernetes**

* Explanation of common realtime issues DevOps Engineers face with Stateful Sets and Persistent Volumes.
* Concepts covered: Stateful Sets, Persistent Volumes, Persistent Volume Claims, Storage Classes, and CSI Driver.
* Focus on troubleshooting and fixing issues in realtime.

#### Issue Description

* A developer reported an issue via a Jira ticket:
  + Stateful application (e.g., database) works on AWS EKS but fails on other platforms like AKS, GKE, and local clusters (e.g., Minikube).
  + Issue requires DevOps expertise in Kubernetes.

#### Initial Setup

* A Minikube cluster is used for demonstration.
* Example stateful set YAML for an Nginx web application with three replicas.
* Persistent Volume Claim (PVC) template uses the storage class "EBS."

#### Troubleshooting Steps

1. **Deployment and Initial Check:**
   * Apply the stateful set YAML.
   * Check stateful set status: kubectl get statefulset.
   * Issue: Zero out of three pods are running.
2. **Pod Status Check:**
   * List pods: kubectl get pods.
   * Only one pod in pending status.
3. **Describe Pod:**
   * Describe pod to find detailed status: kubectl describe pod <pod-name>.
   * Warning: Pod has unbound immediate Persistent Volume Claims.

#### Understanding the Issue

* Stateful set is requesting a persistent volume that it cannot find.
* Unique feature of stateful sets: Each replica is created sequentially only if the previous one is successful.

#### Workflow Explanation

1. **Persistent Volume Claim (PVC) Workflow:**
   * Stateful set requests a PVC.
   * PVC has a storage class (e.g., EBS).
   * Storage class uses a provisioner to create a Persistent Volume (PV).
2. **AWS EKS Example:**
   * On AWS EKS, the stateful set works because EBS provisioner handles the request correctly.
   * On other platforms (e.g., Minikube, AKS, GKE), the storage class and provisioner might differ.

#### Fixing the Issue

1. **Identify Available Storage Classes:**
   * Check available storage classes: kubectl get storageclass.
   * On Minikube, the default storage class is "standard."
2. **Update PVC Template:**
   * Modify the stateful set YAML to use the "standard" storage class.
3. **Apply Changes:**
   * Delete the existing stateful set and PVCs: kubectl delete -f <statefulset-yaml>.
   * Apply the updated YAML: kubectl apply -f <statefulset-yaml>.
4. **Verify Pod Status:**
   * Watch pod status to ensure they transition from pending to running sequentially.

#### Advanced Concept: CSI Driver

* **External Storage Integration:**
  + For using non-native storage (e.g., NetApp) on platforms like AWS EKS.
  + Requires installation of a Container Storage Interface (CSI) driver.
* **Workflow with CSI Driver:**
  + Install the CSI driver from the storage provider.
  + The CSI driver acts as a provisioner or communicates with the provisioner to create PVs on external storage.

#### Conclusion

* Ensure correct storage class configuration when transitioning between different Kubernetes environments.
* For enterprise-level storage solutions, CSI drivers are essential.
* Upcoming video will demonstrate using CSI drivers with enterprise storage solutions like NetApp on EKS.

#### Summary

* The key to solving issues with stateful sets and persistent volumes often lies in correctly configuring storage classes and understanding the sequential nature of pod creation in stateful sets.
* Additional concepts like CSI drivers enable integration with external storage solutions, providing flexibility and performance benefits in enterprise environments.

**5.** **Real-time Security Access Issue in DevOps with Kubernetes Network Policies**

* Real-time security access issue faced by DevOps Engineers.
* Troubleshooting and fixing the issue using network policies in Kubernetes.
* Securing access to a database using network policies.

### Use Case

1. **Scenario:**
   * Kubernetes cluster with two namespaces: secure and sandbox.
   * Secure namespace contains a deployed database (e.g., Redis, MongoDB, MySQL).
   * The database should only be accessed by a specific application in the secure namespace.
2. **Security Requirement:**
   * Prevent access to the database from other namespaces (e.g., sandbox).
   * Security purpose: Prevent potential hacking or unauthorized access.

### Concept of Network Policy

* **Network Policy:**
  + Tool used to restrict access to pods within a Kubernetes cluster.
  + Divided into two categories: Ingress and Egress.
* **Ingress Policy:**
  + Restricts inward traffic to a pod.
  + Example: Blocking access to the database from pods in other namespaces.
* **Egress Policy:**
  + Restricts outward traffic from a pod.
  + Example: Blocking access to unsecure websites from pods.

### Practical Demonstration

1. **Setting Up Redis Database:**
   * Create a secure namespace.
   * Deploy Redis database in the secure namespace.
   * Verify Redis is running.
2. **Simulating Unauthorized Access:**
   * Create a pod (e.g., HTTPD pod) in the default namespace, representing a potential hacker.
   * Install Redis CLI in the hacker pod.
   * Demonstrate the hacker pod can connect to the Redis database in the secure namespace.
3. **Applying Network Policy:**
   * Create a network policy YAML file to secure the Redis pod.
   * Network policy configuration:
     + Apply to pods with the label app=redis.
     + Policy type: Ingress.
     + Only allow access from pods with the label role=non-redis-member.
   * Apply the network policy in the secure namespace.
   * Verify the hacker pod can no longer connect to the Redis database.

### Technical Details

1. **Network Policy YAML Configuration:**
   * Specifies the target pod using labels.
   * Defines policy type (Ingress) and allowed sources (pod selector, namespace selector, or IP blocks).

**6.** **Kubernetes Real-Time challenges in production:**

#### 1. Resource Management

**Resource Sharing**

One of the most common challenges DevOps engineers’ faces is resource sharing within a Kubernetes cluster. In a production environment, multiple development teams often deploy their services on the same Kubernetes cluster. For example, in an e-commerce application, you might have a web team handling frontend service, a payments team handling payment services, and a transactions or delivery team managing shipment services. All these teams share the same cluster resources, such as CPU and RAM.

When multiple teams deploy services on the same cluster without proper resource allocation, issues can arise. For instance, if one service starts leaking memory, it can consume more resources than intended, causing other services to crash due to insufficient resources. This situation can lead to an "Out of Memory" (OOM) killed error, where Kubernetes terminates pods to free up memory.

**Resource Quotas**

To prevent such issues, you can implement resource quotas. A resource quota is a limit set on a namespace to control the amount of CPU and RAM that services within that namespace can consume. For example, if a namespace has a quota of 15 GB RAM and 15 CPU, all services within that namespace cannot collectively exceed this limit. This approach helps ensure fair resource distribution among different teams and services.

However, setting resource quotas alone is not sufficient. While it restricts the impact of resource leaks to a particular namespace, it doesn't solve the issue of individual pods consuming excessive resources.

**Resource Limits**

To address this, you need to set resource limits on individual pods. Resource limits define the maximum amount of CPU and RAM that a single pod can consume. By setting these limits based on performance benchmarking, you can ensure that even if a pod leaks memory, it only affects itself and not the entire namespace or cluster. This step reduces the blast radius of any resource-related issues to the pod level.

For example, if a pod has a limit of 8 GB RAM and it leaks memory, Kubernetes will terminate the pod once it exceeds this limit. This approach ensures that other pods within the same namespace continue to function without interruption.

#### 2. Handling OOMKilled Errors

Even with resource quotas and limits in place, you may still encounter OOMKilled errors. This error occurs when a pod exceeds its allocated memory and is terminated by Kubernetes. To resolve this issue, follow these steps:

**Thread Dumps and Heap Dumps**

When you notice a pod going into OOMKilled status, you need to investigate the root cause. For Java applications, you can take thread dumps and heap dumps to analyze memory usage. Use commands like kill -3 or jstack to collect these dumps from the pod.

**Analysis and Resolution**

Share the collected dumps with the development team. They will analyze the dumps to identify memory leaks or other issues causing excessive memory usage. Once the root cause is identified, the development team will fix the issue and provide a new version of the application. You can then deploy this new version to the cluster.

By taking thread dumps and heap dumps, you provide crucial information to the development team, enabling them to resolve memory-related issues effectively.

#### 3. Upgrades

Upgrading a Kubernetes cluster is another common challenge that DevOps engineers face. Upgrades are necessary to leverage new features, security patches, and performance improvements. However, upgrading a production cluster requires careful planning and execution to avoid downtime and ensure a smooth transition.

**Manual and Detailed Steps**

To handle upgrades, create a detailed manual outlining the upgrade process. This manual should include steps for:

* **Taking Backups:** Ensure you have a backup of all resources before starting the upgrade.
* **Reading Release Notes:** Review the release notes of the new Kubernetes version to understand any breaking changes or new features that might impact your cluster.
* **Control Plane Components:** Upgrade control plane components like etcd, kube-apiserver, and kube-scheduler in the correct order.
* **Worker Nodes:** Upgrade worker nodes by draining them, making them unschedulable, performing the upgrade, and then rejoining them to the cluster.

**Drain and Upgrade Nodes**

When upgrading worker nodes, follow these steps:

1. **Drain the Node:** Move all pods from the node to other nodes.
2. **Make the Node Unschedulable:** Prevent new pods from being scheduled on the node.
3. **Upgrade the Node:** Perform the necessary upgrades on the node.
4. **Rejoin the Node:** Rejoin the upgraded node to the cluster and remove the unschedulable taint.

Repeat these steps for each node in the cluster to complete the upgrade process. This method ensures minimal disruption to your services during the upgrade.

#### Conclusion

By understanding and addressing these real-time challenges—resource management, handling OOMKilled errors, and performing upgrades—you can effectively manage and maintain a production Kubernetes cluster. Explaining these scenarios in interviews will demonstrate your practical experience and problem-solving skills in handling Kubernetes-related issues. Thank you for watching today's video. Please share your feedback in the comment section, and stay tuned for more Kubernetes-related content. See you in the next video!

**GitOps & Argo-CD**

**1. Introduction**

**Introduction to GitOps:**

* GitOps can be complex without understanding its fundamentals.
* Basic GitOps involves using a GitOps controller like Argo CD or Flux CD to deploy applications onto a Kubernetes cluster.
* GitOps is more than basic deployment; it includes managing both applications and infrastructure.

**Understanding GitOps:**

* **Definition**: Uses Git as a single source of truth for delivering applications and infrastructure.
* **Why GitOps**:
  + Without GitOps: No versioning or auditing for Kubernetes changes.
  + With GitOps: Provides versioning, tracking, and auditing similar to source code management.
* **How GitOps Works**:
  + Changes are made via pull requests in a Git repository.
  + GitOps tools (e.g., Argo CD) pick up changes and deploy them to Kubernetes.

**GitOps Workflow:**

* DevOps engineer updates Kubernetes YAML manifests.
* Pull request is reviewed and merged.
* GitOps controller detects the change and applies it to the Kubernetes cluster.
* This process standardizes deployment and ensures changes are tracked.

**Principles of GitOps:**

1. **Declarative**: Desired state must be expressed declaratively.
2. **Versioned and Immutable**: All changes are versioned and immutable.
3. **Pulled Automatically**: Changes are automatically detected and applied.
4. **Continuously Reconciled**: Ensures the Kubernetes state matches the desired state in Git.

**Advantages of GitOps:**

* **Security**: Prevents unauthorized changes; GitOps controllers enforce the desired state.
* **Versioning**: Maintains a history of changes, ensuring traceability.
* **Auto Upgrades**: Supports automatic updates using pull or push mechanisms.
* **Auto Healing**: Automatically corrects unintended changes, enhancing reliability.
* **Continuous Reconciliation**: Ensures the Kubernetes cluster state remains consistent with the Git repository.

**GitOps Beyond Kubernetes:**

* GitOps is not limited to Kubernetes; it can be applied to other infrastructures.
* Currently, popular GitOps tools like Argo CD and Flux primarily target Kubernetes.

**2. Architecture of Argo-CD**

**Popular GitOps Tools in the Market**

1. **Argo CD**:
   * Considered one of the best GitOps tools available.
   * Open-source project initially created by engineers at Atlassian.
   * Contributors include companies like Equity, BlackRock, Codefresh, Intuit, and Red Hat.
   * Part of CNCF, with over 13k stars on GitHub.
2. **Flux CD**:
   * Another CNCF GitOps tool.
   * Often compared with Argo CD.
3. **Jenkins X**:
   * Another tool in the GitOps space.
4. **Spinnaker**:
   * Primarily a deployment-oriented tool but can be used for GitOps.

**Understanding GitOps**

**GitOps Tools Functionality**:

* Maintain sync between version control systems (like Git) and Kubernetes.
* Use Git as the single source of truth.
* Continuously watch and reconcile the state between Git and Kubernetes.

**Architecture of GitOps Tools (Focusing on Argo CD)**

1. **Repo Server**:
   * Connects to Git to get the state of the repository.
2. **Application Controller**:
   * Connects to Kubernetes to get the state of the cluster.
   * Compares the state from Git with Kubernetes and syncs if necessary.
3. **API Server**:
   * Facilitates user interaction via UI or CLI.
   * Handles authentication and integrates with existing OIDC providers.
4. **Dex**:
   * Provides SSO capabilities, integrating with various authentication providers.
5. **Redis**:
   * Used for caching to maintain state information.

**Microservices Breakdown:**

* Repo Server: Talks to Git.
* Application Controller: Talks to Kubernetes.
* API Server: Handles user interactions.
* Dex: Manages SSO and authentication.
* Redis: Provides caching for stateful operations.

**Installation of Argo CD**

* **Methods**:
  + YAML manifest.
  + Helm.
  + Kubernetes operators.

**Github Actions**

* **GitHub Actions: Core Concepts**
* Foundation:
  + GitHub Actions is a feature tied to GitHub repositories, enabling automated workflows.
* Key Building Blocks:
  + Workflows:
    - The top-level structure.
    - Attached to GitHub repositories.
    - Contain one or more jobs.
    - Triggered by events (e.g., code pushes, manual activation).
  + Jobs:
    - Contained within workflows.
    - Define the execution environment (runner).
    - Contain one or more steps.
    - Runners can be GitHub-hosted (Linux, macOS, Windows) or self-hosted.
    - Jobs run in parallel by default, but can be configured to run sequentially.
    - Jobs can be conditional.
  + Steps:
    - Contained within jobs.
    - Define the actual tasks to be performed.
    - Can be shell scripts/command-line commands or "actions."
    - Actions are reusable, pre-defined scripts.
    - Steps within a job execute sequentially.
    - Steps can be conditional.
* Workflow Triggers/Events:
  + Determine when a workflow is executed.
  + Examples: manual activation, code pushes.
* Runners:
  + Execution environments (machines/operating systems).
  + GitHub-hosted or self-hosted.
* Actions:
  + Reusable, pre-defined scripts for specific tasks.
  + Can be custom-built or third-party.
  + A job must contain at least one step.

In essence:

1. You create a workflow within your GitHub repository.
2. The workflow is triggered by an event.
3. The workflow contains one or more jobs.
4. Each job runs on a runner.
5. Each job contains one or more steps that perform specific actions.
6. Steps can be command line instructions, or they can call prebuilt actions.

* **GitHub Actions: Creating a First Workflow**

1. Setup:

* GitHub Account:
  + Any GitHub account (free or paid) can be used.
* New Repository:
  + Create a new repository (e.g., gh-first-action).
  + Make it public.
  + Add a README.md file for initial content.
* Access Actions:
  + Navigate to the "Actions" tab in the repository.

2. Workflow Creation:

* Workflow vs. Action:
  + The "Actions" tab is used to create *workflows*.
  + "Actions" are reusable scripts within workflows.
* Template Selection:
  + Start with the "simple workflow" template ("Configure").
  + Other templates are available for more complex tasks.
* File Naming and Location:
  + Name the workflow file (e.g., first-action.yml).
  + Store it in .github/workflows/.
  + YAML format (.yml).
* YAML Editor:
  + GitHub provides an in-browser YAML editor.
  + YAML is indentation sensitive.

3. Workflow Structure (YAML):

* name:
  + Defines the workflow's name (e.g., First Workflow).
* on:
  + Specifies the trigger events.
  + workflow\_dispatch: for manual triggering.
  + Other events exist for automatic triggering (e.g., push).
* jobs:
  + Defines the jobs within the workflow.
  + Jobs run in parallel by default.
  + Indentation is critical in YAML.
    - Job names are user defined, all other keywords are reserved.
* Job Definition (e.g., first-job:):
  + runs-on:
    - Specifies the runner (execution environment).
    - Examples: ubuntu-latest, windows-latest, macos-latest.
    - To find other valid runners, search "Github Actions runners" in a search engine.
  + steps:
    - Defines the steps within the job.
    - Steps execute sequentially.
    - Steps are defined by a dash - then key value pairs.
    - name:
      * Gives the step a descriptive name (e.g., print greeting).
    - run:
      * Executes a command in the command line (shell script).
      * Example: run: echo "Hello World".
      * Steps can also use prebuilt "actions".

4. Example Workflow:

YAML

name: First Workflow

on:

workflow\_dispatch:

jobs:

first-job:

runs-on: ubuntu-latest

steps:

- name: print greeting

run: echo "Hello World"

- name: print goodbye

run: echo "done - bye"

Key Points:

* Workflows are YAML files in .github/workflows/.
* on: defines triggers.
* jobs: define the work to be done.
* runs-on: defines the execution environment.
* steps: define individual tasks.
* run: executes command-line commands.
* Indentation is crucial in YAML.
* **GitHub Actions: Running and Observing a Workflow**

1. Committing the Workflow:

* Saving the YAML:
  + After defining the workflow in the YAML editor, commit the changes.
  + This adds the workflow file to the repository's code.
* Workflow as Code:
  + GitHub Actions workflows are part of your repository's code.
  + They are defined in files within the repository.
* .github/workflows/:
  + Workflows are stored in the .github/workflows/ directory.

2. Observing Workflow Runs:

* Actions Tab Update:
  + After committing, the "Actions" tab updates.
  + It displays a list of identified workflows.
* Workflow Identification:
  + GitHub identifies workflows by searching for YAML files in .github/workflows/.
* Manual Triggering:
  + The workflow\_dispatch trigger enables manual workflow execution.
  + A "Run workflow" button appears.
* Workflow Execution:
  + Clicking "Run workflow" triggers the workflow.
  + Workflow runs are displayed in the "Actions" tab.
* Run Status:
  + Yellow dot: Workflow is running.
  + Green check mark: Workflow completed successfully.

3. Workflow Run Details:

* Workflow Run Page:
  + Clicking on a workflow run displays detailed information.
* Job Details:
  + The page shows the jobs executed in the workflow.
  + Clicking on a job displays its steps.
* Step Details:
  + Each step's details are available.
  + Includes:
    - Setup and cleanup steps (automatically added).
    - User-defined steps.
    - Executed commands.
    - Command results.
* Viewing Logs:
  + You can view the logs of each step, to see the output of the commands.

4. Re-running Workflows:

* Multiple Runs:
  + Workflows can be triggered multiple times.
  + The "Run workflow" button can be used repeatedly.
  + This allows for testing and repeated automation.

Key Takeaways:

* Workflows are saved as part of the repository's code.
* The "Actions" tab provides a central location for monitoring and managing workflows.
* GitHub provides detailed logs, that allows for debugging.
* Manual triggers allow for on demand execution of workflows.
* **GitHub Actions: Setting Up a React Project**

**1. Project Setup:**

* **Local Project:**
  + A ReactJS project is used for demonstration.
  + ReactJS knowledge is not required.
  + The project has:
    - Source code.
    - package.json (dependencies).
    - Automated tests.
* **Local Execution:**
  + Node.js must be installed.
  + npm install installs dependencies.
  + npm run dev starts a local development server.
  + npm test executes automated tests.
* **.gitignore:**
  + The .gitignore file is used to exclude the node\_modules folder from version control.

**2. Git Setup:**

* **Initialize Git:**
  + git init initializes a Git repository in the project directory.
* **Initial Commit:**
  + git add . stages all files.
  + git commit -m "Initial commit" creates the initial commit.
* **GitHub Repository:**
  + Create a new repository on GitHub.
  + Repository name: second-action-react-demo (or similar).
  + Can be private or public.
  + Do not add initial files (README, etc.) on GitHub.
* **Connect Local to Remote:**
  + Copy the repository URL from GitHub.
  + git remote add origin <repository\_url> adds the remote repository.
  + To make sure that you are prompted for your credentials, add your username to the url:
    - git remote add origin https://<your\_username>@github.com/<your\_username>/<repository\_name>.git
  + git push -u origin main pushes the local code to the remote repository.
* **Code on GitHub:**
  + After pushing, the project code is available on GitHub.

**3. Preparing for GitHub Actions:**

* **Purpose:**
  + Automate testing on GitHub.
  + Run tests whenever code is pushed.
* **Next Steps:**
  + Create a GitHub Actions workflow to run the tests.

**Key Points:**

* Using a real-world project provides a more practical example.
* Git setup is essential for integrating with GitHub Actions.
* The goal is to automate testing on GitHub.
* The .gitignore file is very important.

**1. Why the organizations are moving towards github actions from Jenkins**

Organizations are increasingly adopting GitHub Actions over Jenkins due to several key advantages in terms of integration, maintenance, scalability, cost, and ease of use. Below are the detailed reasons for this shift:

### ****1. Seamless GitHub Integration****

* GitHub Actions is **natively integrated** with GitHub, whereas Jenkins requires **separate configurations** (e.g., webhooks, API tokens) to work with GitHub repositories.
* With GitHub Actions, **workflow triggers** (on push, pull request, release, etc.) happen instantly within GitHub, reducing setup time.
* **Example**: You can trigger a CI/CD pipeline directly from a **GitHub event** (e.g., PR merge), without configuring external tools.

### ****2. Built-in CI/CD with No External Infrastructure****

* Jenkins requires **dedicated infrastructure** (self-hosted servers or cloud instances), whereas GitHub Actions runs **fully managed on GitHub-hosted runners**.
* GitHub provides **free CI/CD minutes** on GitHub-hosted runners, reducing operational overhead.
* With Jenkins, teams must **manage server updates, plugins, scaling, and maintenance**, whereas GitHub Actions is automatically updated.

### ****3. Simpler Configuration with YAML****

* GitHub Actions uses **YAML-based workflows**, which are easier to read and manage compared to Jenkins' **Groovy-based pipelines**.
* **Example:** A GitHub Actions workflow (.github/workflows/deploy.yml):

yaml

CopyEdit

name: Deploy to Production

on: push

jobs:

deploy:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Deploy Application

run: ./deploy.sh

* In Jenkins, pipelines require writing **Groovy scripts** (Jenkinsfile), which have a steeper learning curve.

### ****4. Security Advantages****

* **GitHub Actions** provides built-in **secrets management**, whereas Jenkins often requires **third-party plugins** (e.g., HashiCorp Vault, Jenkins Credentials Plugin).
* GitHub Actions limits access to secrets based on **repository permissions**, whereas Jenkins may expose secrets through logs if not configured properly.
* **Code scanning & security tools** like **GitHub Advanced Security (SAST, secret scanning)** integrate natively with GitHub Actions.

### ****5. Scalable and Cost-Effective****

* **GitHub-hosted runners** scale automatically, whereas Jenkins requires **manual provisioning and scaling** of worker nodes.
* **Reduced maintenance cost**: No need to maintain Jenkins infrastructure, update plugins, or manage scaling.
* Organizations using **GitHub Enterprise** benefit from additional security, compliance, and performance optimizations.

### ****6. Ecosystem and Marketplace****

* **GitHub Actions Marketplace** provides **pre-built actions** for cloud providers (AWS, Azure, GCP), security tools, testing frameworks, and deployments.
* Example:
  + actions/setup-node for Node.js
  + aws-actions/configure-aws-credentials for AWS
* In Jenkins, similar functionalities require **plugins** that need to be installed and managed manually.

### ****7. Faster Execution and Parallelism****

* **GitHub Actions** allows **parallel job execution** natively, whereas Jenkins requires additional configuration using **multi-branch pipelines**.
* **Matrix Builds**: Run jobs across multiple OS, environments, and dependencies simultaneously.

yaml

CopyEdit

strategy:

matrix:

os: [ubuntu-latest, windows-latest]

* Jenkins supports parallel execution but requires **complex scripting**.

### ****8. Easy Debugging and Monitoring****

* GitHub Actions provides a **clean, UI-driven workflow view** inside the GitHub repository.
* Real-time logs, re-run options, and failure notifications are integrated into GitHub’s interface.
* In Jenkins, log analysis often requires navigating the console logs or installing plugins like **Blue Ocean** for a visual pipeline view.

### ****9. Compliance and Audit Logs****

* GitHub provides **audit logs** for all workflow executions, useful for **compliance and security audits**.
* Jenkins requires external logging solutions like **ELK Stack (Elasticsearch, Logstash, Kibana)** for similar monitoring.

### ****10. Future-Proof and Industry Adoption****

* As organizations move towards **GitOps** and cloud-native CI/CD, GitHub Actions provides a **modern alternative** to Jenkins.
* Many companies migrating to **GitHub Enterprise Cloud** prefer to **centralize DevOps** within GitHub, reducing tool sprawl.
* **Example Organizations Adopting GitHub Actions:**
  + **Microsoft**: Migrated from Azure DevOps Pipelines
  + **Facebook**: Uses GitHub Actions for mobile app CI/CD
  + **Netflix**: Integrates GitHub Actions with cloud deployments

### ****When to Use Jenkins Instead of GitHub Actions?****

While GitHub Actions is gaining popularity, some organizations **still use Jenkins** in specific cases:  
✅ **Highly customized pipelines**: If an enterprise has thousands of existing Jenkins jobs, migration may not be immediate.  
✅ **On-premise deployment**: Jenkins is ideal when working in **air-gapped environments** or strict **on-premise infrastructures**.  
✅ **Large-scale CI/CD needs**: Organizations with massive workloads might require **self-hosted solutions** for cost efficiency.

### ****Conclusion****

Organizations are moving to **GitHub Actions from Jenkins** primarily for **ease of use, cloud integration, reduced maintenance, and better security**. However, Jenkins remains relevant for **legacy systems and highly customized pipelines**. The **decision depends on business needs**, but the industry trend is increasingly favoring **GitHub Actions** for its **developer-first experience and scalability**.

## ****2. Introduction to GitHub Actions****

GitHub Actions is a **CI/CD (Continuous Integration and Continuous Deployment) tool** built directly into GitHub. It allows developers to automate workflows, such as building, testing, and deploying code, within GitHub repositories.

### ****Key Features of GitHub Actions****

✔ **Event-driven automation** – Workflows are triggered by GitHub events (push, pull request, issue creation, etc.).  
✔ **YAML-based workflow configuration** – Uses .github/workflows/\*.yml files for defining CI/CD pipelines.  
✔ **Pre-built actions** – Provides a marketplace of reusable actions for cloud deployment, security scanning, and more.  
✔ **Integration with GitHub Ecosystem** – Directly integrates with repositories, GitHub Packages, and security features.  
✔ **Parallel and matrix builds** – Run jobs in parallel across different environments and configurations.

## ****GitHub-Hosted Runners vs. Self-Hosted Runners****

In GitHub Actions, **runners** are machines that execute workflows. GitHub provides two types of runners:

### ****1. GitHub-Hosted Runners****

These are managed by GitHub and automatically provisioned when a workflow runs.

#### ****Advantages of GitHub-Hosted Runners****

✅ **No Maintenance Required** – GitHub manages OS updates, security patches, and dependency installations.  
✅ **Pre-installed Software** – Comes with built-in tools like Node.js, Python, Docker, AWS CLI, and Terraform.  
✅ **Auto-Scaling** – GitHub automatically provisions and scales runners as needed.  
✅ **Security** – Workflows run in isolated environments, reducing security risks.  
✅ **Cross-Platform Support** – Available for Windows, Linux, and macOS.

#### ****Disadvantages of GitHub-Hosted Runners****

❌ **Limited Execution Time** – Free-tier runners have a time limit per job (2,000 minutes per month for private repositories).  
❌ **No Customization** – Users cannot modify or configure the underlying runner OS and environment.  
❌ **Cold Start Delay** – Each job starts a new virtual machine, which may take a few seconds to initialize.  
❌ **Cost Consideration** – Additional CI/CD minutes must be purchased if the free limit is exceeded.

### ****2. Self-Hosted Runners****

These are user-managed machines (on-premise or cloud) that execute GitHub Actions workflows.

#### ****Advantages of Self-Hosted Runners****

✅ **Custom Configuration** – Install specific dependencies, security tools, or use a custom OS.  
✅ **No Limits on Execution Time** – Unlike GitHub-hosted runners, there are no restrictions on job run time.  
✅ **Faster Execution** – No cold start delays; runners can stay active and ready for tasks.  
✅ **Cost Control** – More cost-effective for high-volume CI/CD pipelines, as you don't need to pay per-minute usage fees.  
✅ **Supports Private Networks** – Ideal for organizations running jobs in an internal network or behind firewalls.

#### ****Disadvantages of Self-Hosted Runners****

❌ **Infrastructure Management Required** – Users must manage OS updates, security patches, and software dependencies.  
❌ **Security Risks** – Since runners persist between jobs, they can be vulnerable to attacks if not properly secured.  
❌ **Scaling Complexity** – Manual effort required to scale up/down runners as workload increases.  
❌ **Additional Hardware/Cloud Costs** – Requires investment in infrastructure (servers or cloud instances).

## ****Comparison Table: GitHub-Hosted vs. Self-Hosted Runners****

| **Feature** | **GitHub-Hosted Runners 🖥️** | **Self-Hosted Runners ⚙️** |
| --- | --- | --- |
| **Maintenance** | Fully managed by GitHub | Managed by users |
| **Customization** | No, predefined images | Yes, install custom dependencies |
| **Execution Time** | Limited (based on plan) | Unlimited |
| **Scaling** | Automatic | Manual scaling required |
| **Cost** | Free-tier available, paid for extra minutes | May be cheaper for heavy usage |
| **Security** | Secure, isolated environments | Needs security hardening |
| **Performance** | Might have cold start delay | Faster execution |

### ****When to Use Which Runner?****

* ✅ **Use GitHub-Hosted Runners** if you want a **quick, maintenance-free** setup with minimal effort.
* ✅ **Use Self-Hosted Runners** if you need **customized environments**, faster builds, or need to work within a **private network**.

# ****Understanding CI/CD and Its Importance****

## ****1. Introduction to CI/CD****

* CI/CD (Continuous Integration and Continuous Deployment/Delivery) helps streamline the software development process.
* In a typical project, the source code is stored in a Git repository, which provides **version control**.
* GitHub is a **web-based platform** that enhances Git’s capabilities by acting as a centralized hub for hosting repositories.
* The **main or master branch** usually contains the production-ready code.

## ****2. Code Development Workflow****

### ****Feature Branch Workflow****

* Developers create a **feature branch** to introduce new features or modify existing ones.
* This branch is a **clone of the main codebase**, allowing safe collaboration without affecting production.
* Changes are committed to the feature branch, and a **pull request (PR)** is created to merge back into the main branch.
* Before merging, a **code review process** ensures quality and requires approval from team members.

### ****Deployment Process****

* After merging into the main branch, the updated code is deployed to the **Kubernetes production environment**, either **manually** or via automation.
* Without CI/CD, merging untested code directly into production poses a **significant risk** to application stability.

## ****3. Challenges Without CI/CD****

1. **Lack of Early Testing**
   * Without CI/CD, testing occurs **late** in the development cycle, often after multiple merges.
   * This delay makes it harder to catch and fix defects early.
2. **Inefficient Deployment Process**
   * Deployments rely on **manual procedures**, leading to inconsistencies and potential configuration errors.
3. **Quality Issues**
   * Without **automated testing**, quality depends on **manual testing**, which is prone to human errors and resource limitations.

## ****4. Continuous Integration (CI) Process****

### ****CI Pipeline Workflow****

1. A developer creates a **feature branch**, makes changes, and commits the code.
2. A **pull request (PR)** is generated for merging changes into the main branch.
3. Before merging, the **CI pipeline** is triggered, performing:
   * **Unit testing**
   * **Dependency scanning**
   * **Artifact building**
   * **Vulnerability scans**
4. If any test **fails**, the developer makes the required fixes and commits again, re-triggering the pipeline.
5. Once the **CI pipeline passes**, the PR is approved and merged.
6. The CI pipeline runs again on the merged code to ensure stability.

### ****Handling Multiple Feature Branches****

* Example:
  + **Developer A** works on Feature Branch A.
  + **Developer B** works on Feature Branch B.
  + Each feature branch undergoes **independent CI testing** before merging.
  + After merging, the **CI pipeline runs again** to verify both changes work together.
* This process ensures smooth **collaboration and integration** without breaking existing code.

## ****5. Continuous Deployment (CD) and Continuous Delivery****

### ****Continuous Deployment (CD)****

* After merging into the **main branch**, a **CD pipeline** automatically deploys the code to a **production environment**.
* Additional steps include:
  + **Deploying to staging** (a non-production environment for final testing).
  + Running a final **set of automated tests** to ensure quality.
  + If tests pass, the **CD pipeline deploys the code to production automatically**.

### ****Continuous Delivery (CD) with Manual Approval****

* In some cases, **manual approval** is required before deploying to production.
* This **adds a layer of control** to:
  + **Reduce risk**
  + **Ensure compliance**
  + **Allow human oversight**
* The process where **human intervention is needed** before deployment is called **Continuous Delivery**.

## ****6. Summary****

* **CI/CD automates** code integration, testing, and deployment, ensuring faster and safer software delivery.
* **Continuous Integration (CI)** ensures code is merged and tested frequently to prevent integration issues.
* **Continuous Deployment (CD)** automates deployment to production once CI is successful.
* **Continuous Delivery (CD)** introduces a **manual approval step** before production deployment.

This structured approach **improves code quality, reduces risks, and speeds up software releases**.

Here are the **notes** based on the transcript, with additional clarifications where needed:

# **GitHub Actions Overview**

GitHub Actions is an automation tool used for CI/CD. It enables automated workflows that can build, test, and deploy applications.

**Key Components of GitHub Actions**

GitHub Actions consists of three primary components:

1. **Workflow**
2. **Jobs**
3. **Steps**

**1. Workflows**

* A **workflow** is an automated process that can execute one or more jobs.
* Workflows are written in **YAML** files and stored in the .github/workflows/ directory of the repository.
* Each workflow runs in response to a specific **event** (e.g., push, pull\_request).
* Workflows can have an **optional name** using the name keyword, which is visible in the **Actions tab** in GitHub.

**Example: Workflow Triggered on Push**

name: Example Workflow

on: push

jobs:

example-job:

runs-on: ubuntu-latest

steps:

- name: Checkout Code

uses: actions/checkout@v3

* The workflow runs when a **push event** occurs.
* It contains a **job** named example-job that runs on an **Ubuntu** runner.
* The job has **steps**, such as checking out the repository.

**2. Jobs**

* **Jobs** are the **building blocks** of a workflow.
* A workflow can have **one or more jobs**.
* Each job runs on a **runner** (can be **GitHub-hosted** or **self-hosted**).
* The runs-on keyword specifies the runner.

**Example: Defining a Job in a Workflow**

jobs:

example-job:

runs-on: ubuntu-latest

* Here, the job runs on **Ubuntu's latest** version.

**3. Steps**

* **Steps** are individual **tasks** within a job.
* Steps are executed **sequentially** inside the job’s runner environment.
* Steps can:
  + Run **commands**
  + Use **prebuilt actions**
  + Execute **scripts**

**Example: Steps in a Job**

steps:

- name: Checkout Code

uses: actions/checkout@v3 # Prebuilt action to fetch the repo

- name: Install Node.js

uses: actions/setup-node@v3 # Sets up Node.js and NPM

- name: Install Dependencies

run: npm install # Runs a shell command

- name: Run Tests

run: npm test # Runs test scripts

* **First step:** Uses an action (checkout@v3) to pull the repo.
* **Second step:** Uses another action (setup-node@v3) to install Node.js.
* **Third step:** Runs a shell command (npm install).
* **Fourth step:** Runs another shell command (npm test).

**4. Actions**

* **Actions** are reusable automation components used within workflows.
* Actions can be:
  + **Prebuilt by GitHub**
  + **Created by the community**
  + **Custom actions** written by users
* Actions help integrate **third-party tools** into workflows.

**Examples of GitHub Actions**

| **Action** | **Purpose** |
| --- | --- |
| actions/checkout@v3 | Checks out the repository |
| actions/setup-node@v3 | Installs Node.js and NPM |
| docker/build-push-action | Builds and pushes Docker images |
| azure/k8s-create-cluster | Creates an AKS (Azure Kubernetes Service) cluster |
| hashicorp/vault-action | Manages secrets in HashiCorp Vault |

**Additional Points**

1. **Triggers & Events:**
   * Workflows are triggered by **events** (e.g., push, pull\_request, schedule, workflow\_dispatch).
   * Triggers define when a workflow runs.
2. **Runners:**
   * Jobs require a **runner** (a server that executes jobs).
   * Types of runners:
     + **GitHub-hosted** (e.g., ubuntu-latest, windows-latest, macos-latest)
     + **Self-hosted** (custom runner provided by the user)
3. **Sequential Execution:**
   * Steps within a job run sequentially, but jobs within a workflow can run **parallelly or sequentially**, depending on dependencies.
4. **Using Third-Party Actions:**
   * Workflows can use **community-built actions** from the GitHub Actions marketplace.

**Summary**

| **Component** | **Description** |
| --- | --- |
| **Workflow** | The automation process triggered by events (written in YAML) |
| **Jobs** | Building blocks of a workflow; each job runs on a runner |
| **Steps** | Individual tasks executed sequentially within a job |
| **Actions** | Prebuilt or custom automation components used inside steps |

# GitHub Actions: Setting Up and Running Your First Workflow

## 1. Creating a Repository

* You can create a new repository or use an existing one.
* Recommended to create a **public repository** for training purposes.
* Add a **README.md** file to initialize the repository.

## 2. Opening the Repository in VS Code

* Modify the repository URL from github.com to github.dev to open the **GitHub code editor**.
* Install the **GitHub Actions plugin** from the VS Code marketplace for better workflow management and validation.

## 3. Creating the GitHub Actions Workflow File

* Workflows must be placed inside .github/workflows/ directory.
* Example of creating a simple workflow:
* name: First Workflow
* on: push # Trigger workflow on every push
* jobs:
* first\_job:
* runs-on: ubuntu-latest # Define the runner (virtual machine)
* steps:
* - name: Print Welcome Message
* run: echo "Hello, GitHub Actions!"
* - name: List Repository Files
* run: ls -la
* - name: Read README File
* run: cat README.md

## 4. Understanding Workflow Execution

* When a file is committed and pushed, the workflow runs automatically.
* Navigate to the **Actions** tab in GitHub to monitor workflow execution.
* The workflow logs display the steps executed and highlight any errors.

## 5. Debugging Workflow Failures

* If the workflow fails, check the logs to identify the issue.
* Example failure:
  + cat README.md failed because the runner does not have access to the repository files.
* To resolve this, explicitly **check out** the repository using the actions/checkout action:
* steps:
* - name: Checkout Repository
* uses: actions/checkout@v4
* - name: List Repository Files
* run: ls -la

## 6. Understanding GitHub Actions Runners

* GitHub Actions uses **virtual machines (runners)** to execute workflows.
* By default, it runs on **GitHub-hosted runners** such as:
  + ubuntu-latest
  + windows-latest
  + macos-latest
* Runners come with **pre-installed software** (details available in GitHub documentation).

## 7. GitHub Token and Permissions

* A **GitHub Token** is automatically created when the action runs.
* Default permissions: **Read-only**.
* Modify permissions if the workflow needs to write, delete, or update resources.

## 8. Using Predefined Workflows

* GitHub provides **predefined workflows** based on project type (e.g., Node.js, Python, Docker deployments).
* These can be customized to suit project needs.

## 9. Next Steps

* Explore more GitHub Actions triggers (pull\_request, schedule, workflow\_dispatch).
* Learn about **job dependencies**, **artifacts**, and **secrets management**.
* Automate deployments using GitHub Actions and integrate with cloud providers (AWS, Azure, GCP).

By following these steps, you can successfully create, debug, and optimize GitHub Actions workflows for automation and CI/CD tasks!

## ****GitHub Actions Overview****

GitHub Actions are **prebuilt, reusable automation components** designed for specific tasks within GitHub workflows. They can be created by you or the community, enabling **easy sharing and reuse** of automation logic.

### ****Finding GitHub Actions****

* Actions can be found in the **GitHub Marketplace**.
* **Verified actions** (with a ✅ tick badge) are created by **GitHub or verified partners**.
* **Unverified actions** (without a tick badge) are created by **community members**.
* Always **review the source code** of community actions to ensure:
  + Secrets are not leaked.
  + It behaves as expected.
  + No unintended host interactions.

## ****Adding an Action to a Workflow****

1. **Locate the Action** in the GitHub Marketplace.
2. **Read the Action’s Documentation** to check:
   * Available **versions**.
   * Required **workflow syntax**.
   * Supported **input/output parameters**.
3. **Define the Action in the Workflow** using:
   * **Tags**
   * **Branches**
   * **SHA values**

## ****Specifying Action Versions****

GitHub Actions can be referenced in a workflow using:

1. **Tags** (@v3.6.0)
   * Used to **switch between major and minor versions**.
   * Example:
   * uses: actions/setup-node@v3.6.0
2. **Branches** (@main)
   * Always runs the **latest version on that branch**.
   * Risk: If an update includes **breaking changes**, it may affect your workflow.
   * Example:
   * uses: actions/setup-node@main
3. **SHA values** (@commit-sha)
   * **Most reliable method** as SHA values are **immutable**.
   * Recommended for ensuring **workflow stability**.
   * Example:
   * uses: actions/setup-node@f3bdf70e4f9a73c21a5dcd7e

✅ **Best Practice:** **Use SHA values** for production workflows to avoid unexpected changes.

## ****Additional Key Points****

* **Security Considerations:**
  + Always **review the source code** before using third-party actions.
  + Ensure **secrets** are handled securely.
  + Avoid exposing **repository contents** to unintended hosts.
* **Updating Actions Safely:**
  + Pin actions to a **specific version** (@vX.Y.Z) or **SHA hash** for **reproducibility**.
  + Avoid referencing the **main branch** unless necessary.
* **Community vs. Official Actions:**
  + Use **GitHub or verified actions** for critical workflows.
  + If using community actions, **review & test** before deployment.

Here are the summarized notes along with additional key points on **GitHub Actions - Checkout Action**:

## ****GitHub Actions - Checkout Action****

### ****Purpose of Checkout Action****

* The **checkout action** is used to **clone the GitHub repository** onto the runner before performing any operations.
* This ensures that workflows can access the repository's content.

## ****Finding and Using the Checkout Action****

1. **Go to GitHub Marketplace** → Click on **Actions** to explore various available actions.
2. **Search for "Checkout Action"** → Look for actions/checkout, which is:
   * Created by **GitHub**.
   * Marked with a ✅ **verified tick**.
3. **View Documentation** → The documentation provides:
   * The latest **version (e.g., v4.1.0)**.
   * The **syntax** for using the action.
   * Optional **parameters** for customization.
4. **View Action Code** → Hosted on GitHub under the **actions organization**.
   * It is written in **JavaScript**.
   * You can review the source code before using it.

## ****Adding the Checkout Action in a Workflow****

To use the checkout action, add the following step **before running any repository-related commands**:

jobs:

example-job:

runs-on: ubuntu-latest

steps:

- name: Checkout Repository

uses: actions/checkout@v4

### ****Breaking Down the Syntax****

* **name:** → Defines the step name.
* **uses:** → Specifies the action.
* **actions/checkout@v4** → Calls the checkout action with version **v4**.

✅ **Best Practice:** Always use the **latest stable release** (@v4 or @v4.1.0) for updates and security fixes.

## ****Executing the Workflow****

1. **Commit and Push Changes** → Workflow is triggered.
2. **Go to GitHub Actions Tab** → The new workflow run appears.
3. **Check the Execution Status**:
   * If **successful**, all steps will execute without errors.
   * If **failed**, check logs to identify issues.

## ****Understanding Workflow Execution****

1. **Downloading Actions to the Runner**
   * GitHub first **downloads the required actions** to the runner before execution.
   * Logs will show:
   * Downloading actions/checkout@v4
2. **Cloning the Repository**
   * The **entire repository** is cloned into the runner.
   * Running ls inside the workflow will list the repository files.
   * Hidden folders (.github/workflows) are not displayed by default but can be seen with:
   * ls -a
3. **Post-Checkout Actions**
   * Some actions **perform cleanup** after execution.
   * The logs may show:
   * Post checkout repository cleanup
   * This ensures unnecessary files are removed after job completion.

## ****Analyzing Logs in GitHub Actions****

* GitHub provides various ways to inspect logs:
  + **View Logs in GitHub UI**
  + **Download Log Archives**
  + **View Raw Logs**
  + **Check Execution Timestamps**
    - Displays **exact times** when each step was executed.

## ****Key Takeaways****

✅ **Always include the checkout action before working with repository files.**  
✅ **Use the latest stable version (v4 or v4.1.0) for security and performance improvements.**  
✅ **Check logs to troubleshoot errors and view execution details.**  
✅ **Post-checkout steps ensure cleanup and optimization.**

Here are the summarized notes along with additional key points on **Executing Multiple Commands in a Single Step & Running Third-Party Libraries in GitHub Actions**:

# ****Executing Multiple Commands & Third-Party Libraries in GitHub Actions****

## ****Executing Multiple Commands in a Single Step****

1. In GitHub Actions, multiple commands **can be executed in a single step** using a **pipe (|)** in YAML.
2. Example: Instead of writing multiple steps, we can merge them into one:
3. steps:
4. - name: List and Read Files
5. run: |
6. ls -lta # Lists all files including hidden ones
7. cat README.md # Displays the contents of README.md
8. **Why use a pipe (|)?**
   * In YAML, using a pipe allows commands to be executed **sequentially in the same step**.
   * This reduces redundancy and improves readability.
9. **Commenting out Steps:**
   * If previous steps are not needed, they can be commented out using # in YAML.

## ****Running a Third-Party Library in GitHub Actions****

1. **Example: Using cowsay (a third-party CLI tool)**
   * cowsay generates ASCII art with a message.
   * Example command:
   * cowsay -f dragon "Hello from GitHub Actions!"
   * This generates a **dragon ASCII art** with the message "Hello from GitHub Actions!".
2. **Adding a Step for ASCII Art in GitHub Actions**
3. steps:
4. - name: Generate ASCII Artwork
5. run: cowsay -f dragon "Hello from GitHub Actions!" > dragon.txt
6. **Why the Step Failed?**
   * The **cowsay library is not pre-installed** on the default GitHub Actions runner (ubuntu-latest).
   * Error: command not found
   * Solution: Install the missing package before using it.

## ****Key Takeaways****

✅ **Use | in YAML to execute multiple commands in a single step.**  
✅ **Comment out unnecessary steps using # to keep workflows clean.**  
✅ **Third-party libraries are not always pre-installed in GitHub Actions runners.**  
✅ **If a command fails due to a missing library, install it manually before using it.**  
✅ **Next step: Learn how to install third-party libraries in GitHub Actions before execution.**

# ****GitHub Actions: Creating Workflows, Running Commands, and Managing Workflow Executions****

## ****1. Creating a New Workflow****

* Workflows in GitHub Actions are defined as **YAML files** stored under .github/workflows/.
* A workflow can be created to **perform specific tasks** based on events (e.g., push, pull\_request).
* The workflow file should be **named appropriately** to indicate its purpose.
* Instead of modifying an existing workflow, a **new workflow can be created** for specific tasks.

## ****2. Installing and Using Third-Party Libraries in Workflows****

* Some **third-party CLI tools** may not be pre-installed in GitHub-hosted runners.
* To use such tools, they must be installed **before execution** using appropriate package managers (e.g., apt, yum, brew, npm).
* Example steps to install and use a third-party tool:
* steps:
* - name: Install CLI Tool
* run: sudo apt-get update && sudo apt-get install -y <tool\_name>
* - name: Run CLI Tool
* run: <tool\_name> --options
* **Why this is important?**
  + If a tool is not pre-installed, executing its command **will result in failure**.
  + Installing it before execution ensures the workflow completes successfully.

## ****3. Understanding Workflow Execution and File Persistence****

* Workflows execute **inside temporary virtual machines**.
* Any **files generated during workflow execution are not persisted** in the repository.
  + These files exist **only while the virtual machine is running**.
  + Once the job completes, the **VM is shut down, and the files are deleted**.
* If required, files can be persisted by **uploading artifacts** using:
* - name: Upload Artifact
* uses: actions/upload-artifact@v3
* with:
* name: my-output
* path: output.txt

## ****4. Multiple Workflows and Triggering Events****

* Multiple workflows can exist in a repository, each triggering based on **specific events**.
* If multiple workflows listen to the same event (e.g., push), **all will execute simultaneously**.
* Workflows can be selectively disabled to prevent execution when not needed.

## ****5. Disabling Unused Workflows****

* Unused workflows can be **manually disabled** to prevent execution.
* Disabling is useful when:
  + A workflow is outdated.
  + A new workflow replaces the old one.
  + Only specific workflows should run on a trigger event.
* Workflows can be disabled from the **GitHub Actions UI**.

## ****Key Takeaways****

✅ Workflows should be **structured logically**, and separate workflows should be used for distinct tasks.  
✅ **Third-party tools must be installed** if they are not pre-installed in the runner.  
✅ Any **files created during a workflow run are temporary** and deleted after execution.  
✅ **Multiple workflows can run on the same event** if not managed properly.  
✅ **Unused workflows can be disabled** to optimize execution.

# ****Executing Shell Scripts in GitHub Actions Workflows****

## ****1. Running Shell Scripts in a Workflow****

* Shell scripts can be executed in GitHub Actions using a **step with a run command**.
* Instead of writing multiple shell commands inside the workflow file, it's best practice to store them in a separate **.sh script file**.
* This approach keeps workflows **clean, maintainable, and reusable**.

## ****2. Creating and Executing a Shell Script****

1. **Create a shell script file** in the repository (e.g., script.sh).
2. **Move all required commands** from the workflow YAML file into script.sh.
3. **Modify the workflow** to execute the script instead of running inline commands:
4. steps:
5. - name: Execute Shell Script
6. run: ./script.sh

## ****3. Handling Permission Issues****

* By default, shell scripts may **lack execute permissions** in the GitHub-hosted runner.
* Running a script without execution permissions may result in an error like:
* Permission denied
* Exit code: 126
* To **fix this issue**, update the workflow to modify permissions before execution:
* steps:
* - name: Grant Execute Permission
* run: chmod +x script.sh
* - name: Execute Shell Script
* run: ./script.sh

## ****4. Debugging Workflow Failures****

* If a workflow fails, GitHub provides **detailed logs** under the "Actions" tab.
* Check logs to understand failure reasons, such as:
  + **Permission Denied** (Exit Code 126) → Fix using chmod +x.
  + **Missing Commands** (Command Not Found) → Ensure dependencies are installed.
* **Annotations** in the logs help identify where the error occurred.

## ****5. Workflow Execution & Debugging****

* After making changes, **commit and push** the updated workflow file.
* A new workflow run will **automatically start** (if triggered by push).
* If the script executes successfully, the logs will confirm its completion.

## ****Key Takeaways****

✅ **Use separate shell script files** instead of writing long inline commands in workflows.  
✅ **Set execution permissions** using chmod +x before running a script.  
✅ **Check logs in GitHub Actions UI** to debug failures.  
✅ **Ensure required dependencies/tools** are installed before script execution.

# ****Executing Multiple Jobs in a GitHub Actions Workflow****

## ****1. Introduction to Multi-Job Workflows****

* So far, we have worked with **single-job workflows**, where all steps run sequentially in a single virtual machine (VM).
* **Multi-job workflows** allow defining multiple jobs that can run **in parallel** or **in sequence**, depending on dependencies.
* Multi-job workflows are useful for **CI/CD pipelines**, where jobs can be structured into **build, test, and deploy** phases.

## ****2. Defining Multiple Jobs in GitHub Actions****

### ****Workflow Structure with Multiple Jobs****

A typical CI/CD pipeline consists of:

1. **Build Job (build\_job\_1)**: Compiles or packages the application.
2. **Test Job (test\_job\_2)**: Runs tests on the built artifacts.
3. **Deploy Job (deploy\_job\_3)**: Deploys the application.

### ****Example Workflow with Multiple Jobs****

jobs:

build\_job\_1:

runs-on: ubuntu-latest

steps:

- name: Install Dependencies

run: echo "Installing dependencies..."

- name: Build Project

run: echo "Building the project..."

- name: Create Output File

run: echo "Dragon" > dragon.txt

- name: Sleep for Stability

run: sleep 30

test\_job\_2:

runs-on: ubuntu-latest

steps:

- name: Sleep for Synchronization

run: sleep 10

- name: Test for 'Dragon' Keyword

run: grep "Dragon" dragon.txt

deploy\_job\_3:

runs-on: ubuntu-latest

steps:

- name: Deploy Application

run: echo "Deploying application..."

## ****3. Default Behavior of Multi-Job Workflows****

* By **default**, all jobs in a GitHub Actions workflow **run in parallel** if there are no dependencies between them.
* Jobs run on **separate virtual machines**, meaning they **do not share files or environment variables**.
* If **any one job fails**, the **entire workflow is marked as failed**.

### ****Issues Observed in Parallel Execution****

* The **test job (test\_job\_2) ran before the build job**, causing it to fail because dragon.txt did not exist yet.
* The **deploy job (deploy\_job\_3) did not wait for the test job**, leading to incorrect execution order.
* Since each job runs on a **separate virtual machine**, files created in one job are **not accessible** in another job.

## ****4. Fixing Parallel Execution Issues****

To **run jobs in sequence** and **share artifacts**, we must:

1. **Define Dependencies** (needs:) to enforce execution order.
2. **Use Artifacts** to share files between jobs.

### ****Modified Workflow with Dependencies and Artifact Sharing****

jobs:

build\_job\_1:

runs-on: ubuntu-latest

steps:

- name: Install Dependencies

run: echo "Installing dependencies..."

- name: Build Project

run: echo "Building the project..."

- name: Create Output File

run: echo "Dragon" > dragon.txt

- name: Upload Artifact

uses: actions/upload-artifact@v3

with:

name: build-output

path: dragon.txt

test\_job\_2:

runs-on: ubuntu-latest

needs: build\_job\_1 # Ensures test job runs after build

steps:

- name: Download Artifact

uses: actions/download-artifact@v3

with:

name: build-output

- name: Test for 'Dragon' Keyword

run: grep "Dragon" dragon.txt

deploy\_job\_3:

runs-on: ubuntu-latest

needs: test\_job\_2 # Ensures deploy job runs after test

steps:

- name: Download Artifact

uses: actions/download-artifact@v3

with:

name: build-output

- name: Deploy Application

run: echo "Deploying application..."

### ****Fixes Implemented****

✅ **needs: keyword** ensures correct execution order.  
✅ **upload-artifact and download-artifact** enable file sharing between jobs.  
✅ **Test job now runs only after the build job completes successfully.**  
✅ **Deploy job runs only after the test job completes successfully.**

## ****5. Key Takeaways****

✅ **Multi-job workflows allow breaking tasks into modular phases** like build, test, and deploy.  
✅ **Jobs run in separate virtual machines by default**, meaning they do not share files.  
✅ **By default, jobs run in parallel** unless explicitly controlled.  
✅ **To enforce execution order, use needs: dependencies.**  
✅ **To share files between jobs, use GitHub Actions' upload-artifact and download-artifact.**

This approach ensures a **structured, efficient, and reliable CI/CD pipeline** in GitHub Actions! 🚀

Here's a structured set of notes covering both sessions:

# ****GitHub Actions: Running Multiple Jobs and Job Dependencies****

## ****1. Introduction to Multiple Jobs in a Workflow****

* A GitHub Actions workflow can contain multiple jobs.
* In a CI/CD pipeline, jobs are typically divided into **Build**, **Test**, and **Deploy** phases.
* Each job runs in its **own separate virtual machine**.
* By default, all jobs in a workflow **run in parallel**, unless dependencies are specified.

## ****2. Implementing Multiple Jobs in a Workflow****

### ****Workflow Example:****

Three jobs are created:

1. **Build Job (build\_job1)**:
   * Runs on **Ubuntu latest**
   * Installs CLI tools, executes commands, stores output in a file
   * Sleeps for 30 seconds
2. **Test Job (test\_job2)**:
   * Runs on **Ubuntu latest** (separate machine from build job)
   * Sleeps for 10 seconds
   * Tests if the file (dragon.txt) created in the build job exists
3. **Deploy Job (deploy\_job3)**:
   * Runs on **Ubuntu latest**
   * Reads the file and simulates deployment using echo

## ****3. Issues with Default Parallel Execution****

* By default, all jobs run in parallel.
* The **Test** and **Deploy** jobs started before the **Build** job completed.
* The **Test Job failed** because the file (dragon.txt) was not yet created.
* **Deploy Job was skipped** because its dependency (**Test Job**) failed.

### ****Key Issues Identified:****

1. **Jobs executing in parallel instead of sequence**
2. **Files created in one job are not accessible in another job** (since each job runs on a separate VM)

## ****4. Controlling Job Execution Order using**** needs ****Keyword****

* To ensure jobs execute in a specific order, use **needs** syntax at the job level.

### ****Example: Enforcing Job Order****

jobs:

build\_job1:

runs-on: ubuntu-latest

steps:

- name: Build Step

run: echo "Building..."

test\_job2:

needs: build\_job1 # Runs after build\_job1

runs-on: ubuntu-latest

steps:

- name: Test Step

run: echo "Testing..."

deploy\_job3:

needs: test\_job2 # Runs after test\_job2

runs-on: ubuntu-latest

steps:

- name: Deploy Step

run: echo "Deploying..."

### ****How**** needs ****Works:****

* test\_job2 **waits for** build\_job1 to complete successfully before execution.
* deploy\_job3 **waits for** test\_job2 to complete successfully.
* If a job fails, all dependent jobs are **skipped** automatically.
* **Cycles are not allowed** (e.g., build\_job1 cannot depend on test\_job2 if test\_job2 already depends on build\_job1).

## ****5. Observations After Implementing**** needs

* Jobs now execute **sequentially** instead of parallel.
* The **deploy job was skipped** because its dependency (test\_job2) failed.
* The **test job still failed** because it couldn't access the file created in build\_job1.

### ****Next Steps:****

* We need a way to **share files** between jobs in separate virtual machines.
* This can be achieved using **artifacts** (to be covered in the next session).

### ****Additional Key Points Not Covered in the Original Text:****

✅ **GitHub Actions Plugin** in Visual Studio Code provides helpful syntax hints.  
✅ **Arrays in needs** allow a job to wait for multiple jobs before execution.  
✅ **Workflow graph updates automatically** when needs is used, showing job dependencies visually.

Let me know if you need any modifications or further details! 🚀

Here are the detailed notes on uploading and downloading artifacts in GitHub Actions, along with additional key points:

# ****Uploading and Downloading Artifacts in GitHub Actions****

## ****Objective****

* Learn how to upload artifacts (such as files generated in one job) and download them in subsequent jobs within a GitHub Actions workflow.
* Enable file sharing across different jobs that run on separate virtual machines.

## ****Uploading Artifacts in GitHub Actions****

### ****Why Upload Artifacts?****

* By default, each job runs in a separate virtual machine (VM), meaning files created in one job are not automatically available in another job.
* Uploading artifacts allows files to persist beyond the runtime of a single job and be accessed by other jobs.
* Artifacts can also be stored for later download, even after the workflow completes.

### ****Steps to Upload an Artifact****

1. **Use the actions/upload-artifact action** from GitHub Actions Marketplace.
2. **Define the artifact name** to uniquely identify it.
3. **Specify the file path** to upload.

#### ****Example (Uploading a File in Build Job)****

jobs:

build:

runs-on: ubuntu-latest

steps:

- name: Checkout Repository

uses: actions/checkout@v3

- name: Create a sample file

run: echo "This is a test file" > dragon.txt

- name: Upload the artifact

uses: actions/upload-artifact@v3

with:

name: dragon-text-file

path: dragon.txt

### ****Explanation****

* The upload-artifact action is used to upload dragon.txt.
* The name parameter specifies the artifact’s unique identifier.
* The path parameter specifies the file to be uploaded.

## ****Downloading Artifacts in GitHub Actions****

### ****Why Download Artifacts?****

* Jobs that depend on files from previous jobs need to retrieve those files.
* Downloading artifacts ensures that subsequent jobs can access necessary files.

### ****Steps to Download an Artifact****

1. **Use the actions/download-artifact action** from GitHub Actions Marketplace.
2. **Specify the artifact name** to download the correct file.

#### ****Example (Downloading the File in Test and Deploy Jobs)****

test:

runs-on: ubuntu-latest

needs: build

steps:

- name: Download the artifact

uses: actions/download-artifact@v3

with:

name: dragon-text-file

- name: Verify the file exists

run: cat dragon.txt

deploy:

runs-on: ubuntu-latest

needs: test

steps:

- name: Download the artifact

uses: actions/download-artifact@v3

with:

name: dragon-text-file

- name: Read the file

run: cat dragon.txt

### ****Explanation****

* The download-artifact action retrieves the file uploaded in the build job.
* The name parameter should match the artifact name used in the upload step.
* The file is now accessible in test and deploy jobs.

## ****Artifact Storage in GitHub Actions****

* Artifacts persist beyond job execution and can be downloaded from the GitHub UI.
* Artifacts are stored for **90 days** by default (can be configured).
* The maximum allowed artifact size is **500MB** in free accounts.

### ****How to Change Artifact Retention Period****

1. **Go to Repository Settings** → Click on Actions (left panel).
2. **Click on General Tab**.
3. **Modify Artifact Retention Period** (default: 90 days).

## ****Real-World Use Cases****

* **Java Applications**: Upload JAR or WAR files after the build phase and use them in test and deploy phases.
* **Log Files**: Save logs generated in test jobs for debugging failures.
* **Configuration Files**: Share configuration files between jobs.

## ****Key Takeaways****

✅ **Each job runs in a separate VM** – files are not automatically shared.  
✅ **Use upload-artifact to persist files across jobs.**  
✅ **Use download-artifact to retrieve files in dependent jobs.**  
✅ **Artifacts remain accessible for 90 days (default).**  
✅ **Maximum artifact size in free accounts is 500MB.**  
✅ **Useful for build outputs, logs, test reports, and deployment artifacts.**