Repaso de computación en la Nube y servicios AWS

Temas: Alta disponibilidad y escalabilidad en AWS

Objetivos

Al aprender sobre alta disponibilidad y elasticidad en AWS tendrás los conocimientos y habilidades valiosas para diseñar, implementar y administrar aplicaciones y sistemas escalables y resistentes a fallos en la nube.

Los objetivos de este repaso son profundizar estos temas en tres niveles: desde el punto de vista teórico, dsde el punto de vista páctico usando las herramientas de AWS y por codificación a través de conceptos rudimentarios de python.

Tareas

Prepara una presentación grupal para responder las siguientes preguntas y utiliza un ide de tu preferencia para las implementaciones solicitadas:

Preguntas

Ejercicio 1: Conceptos de escalabilidad vertical y horizontal

Objetivo: Entender las diferencias y aplicaciones de la escalabilidad vertical y horizontal.

Instrucciones:

- Define los conceptos de escalabilidad vertical y horizontal.
- Discute las ventajas y desventajas de cada tipo de escalabilidad.
- Proporciona ejemplos de escenarios en los que elegirías escalabilidad vertical y otros en los que optarías por escalabilidad horizontal. Justifica tus elecciones.
- Diseña un plan de escalabilidad para una aplicación web que tenga un crecimiento proyectado de usuarios del 100% en el próximo año.

Ejercicio 2: Resumen del modelo OSI

Objetivo: Comprender el modelo OSI y su aplicación en redes.

Instrucciones:

- Explica las siete capas del modelo OSI y proporciona un ejemplo de protocolo o tecnología para cada capa.
- Discute cómo el modelo OSI se aplica al diseño y funcionamiento de aplicaciones web.
- Describe cómo las diferentes capas del modelo OSI interactúan en una solicitud HTTP desde un navegador web a un servidor.

Ejercicio 3: Distribución del tráfico web con Amazon ELB

Objetivo: Entender cómo Amazon ELB distribuye el tráfico web.

Instrucciones:

- Explica qué es Amazon ELB y los tipos de balanceadores de carga que ofrece.
- Describe cómo se configura un ELB en una VPC y los beneficios de hacerlo.
- Proporciona un caso de estudio donde se mejore la disponibilidad y rendimiento de una aplicación utilizando Amazon ELB.

Ejercicio 4: Load Balancers y VPCs

Objetivo: Comprender cómo los balanceadores de carga funcionan dentro de una VPC.

Instrucciones:

- Explica cómo configurar un balanceador de carga dentro de una VPC.
- Discute los beneficios y desafíos de utilizar balanceadores de carga dentro de una VPC.
- Proporciona un ejemplo de configuración de seguridad y red para un balanceador de carga en una VPC.

Ejercicio 5: ALB y WAF

Objetivo: Entender el uso de Application Load Balancer (ALB) y Web Application Firewall (WAF) en AWS.

Instrucciones:

- Explica qué es un ALB y cómo difiere de otros tipos de balanceadores de carga en AWS.
- Describe cómo se configura y usa un WAF con un ALB para proteger una aplicación web.
- Proporciona un ejemplo de reglas de WAF que podrías implementar para proteger una aplicación contra ataques comunes como SQL injection y XSS.

Ejercicio 6: Network Load Balancer (NLB) y Gateway Load Balancer (GWLB)

Objetivo: Comprender las características y aplicaciones de NLB y GWLB.

Instrucciones:

- Explica qué es un Network Load Balancer (NLB) y cuándo sería más adecuado utilizarlo en lugar de otros tipos de balanceadores de carga.
- Describe el concepto de Gateway Load Balancer (GWLB) y sus aplicaciones en una arquitectura de red.
- Proporciona un caso de uso donde un NLB y un GWLB trabajen juntos para mejorar la disponibilidad y seguridad de una aplicación.

Ejercicio 7: Classic Load Balancer (CLB)

Objetivo: Conocer las funcionalidades y limitaciones del Classic Load Balancer.

Instrucciones:

- Explica qué es un Classic Load Balancer (CLB) y cómo se compara con ALB y NLB.
- Describe un escenario en el que utilizarías un CLB y otro en el que elegirías un ALB o NLB.
- Proporciona un ejemplo de configuración de un CLB en una arquitectura de aplicación web.

Ejercicio 8: Implementación de elasticidad con Amazon Auto Scaling

Objetivo: Comprender cómo implementar la elasticidad utilizando Amazon Auto Scaling.

Instrucciones:

- Explica qué es Amazon Auto Scaling y sus beneficios.
- Describe cómo configurar grupos de Auto Scaling y los diferentes tipos de políticas de escalado que se pueden aplicar.
- Proporciona un ejemplo de una política de escalado que ajuste dinámicamente la capacidad de una aplicación basada en la carga de tráfico.

Ejercicio 9: Grupos de Auto Scaling y plantillas de configuración

Objetivo: Entender la configuración y uso de grupos de Auto Scaling y plantillas de configuración.

Instrucciones:

- Explica qué es un grupo de Auto Scaling y cómo se configura.
- Describe el propósito de las plantillas de configuración en Auto Scaling y cómo se utilizan.
- Proporciona un caso de uso donde la configuración de plantillas y grupos de Auto Scaling mejora la resiliencia de una aplicación.

Ejercicio 10: Diseño de soluciones de alta disponibilidad Multi-Región

Objetivo: Diseñar soluciones que aseguren alta disponibilidad a nivel multi-región.

Instrucciones:

- Explica los beneficios y desafíos de diseñar una arquitectura de alta disponibilidad multi-región.
- Describe los componentes y servicios de AWS que utilizarías para diseñar una solución multi-región de alta disponibilidad.
- Proporciona un diseño de arquitectura detallado para una aplicación crítica que requiere alta disponibilidad en múltiples regiones, incluyendo consideraciones de balanceo de carga, replicación de datos y recuperación ante desastres.

AWS Lab Learner

Ejercicio 1: Configuración de Amazon ELB para Distribuir Tráfico Web

Objetivo: Configurar y utilizar un Application Load Balancer (ALB) para distribuir tráfico web.

Instrucciones:

- Inicia sesión en AWS Lab Learner.
- Crea una VPC con al menos dos subnets públicas en diferentes zonas de disponibilidad.
- Lanza dos instancias EC2 en diferentes subnets, configurando un servidor web básico en cada una.
- Crea un ALB y añade las instancias EC2 como objetivos del balanceador de carga.
- Configura reglas de listeners para enrutar el tráfico HTTP a las instancias EC2.
- Prueba la configuración accediendo al DNS del ALB y verifica que el tráfico se distribuye entre las instancias EC2.

Resultados esperados:

- Configuración exitosa de un ALB.
- Distribución de tráfico HTTP entre las instancias EC2.
- Verificación del balanceo de carga accediendo al DNS del ALB.

Ejercicio 2: Implementación de elasticidad con Auto Scaling Groups

Objetivo: Configurar un Auto Scaling Group (ASG) para ajustar dinámicamente la capacidad de una aplicación.

Instrucciones:

- Inicia sesión en AWS Lab Learner.
- Crea una plantilla de lanzamiento que especifique la configuración de la instancia EC2 (tipo de instancia, AMI, etc.).
- Configura un ASG utilizando la plantilla de lanzamiento, definiendo los parámetros de escalado (mínimo, máximo y número deseado de instancias).

- Establece políticas de escalado basadas en el uso de CPU, aumentando y disminuyendo el número de instancias según sea necesario.
- Genera una carga de trabajo en las instancias EC2 para activar las políticas de escalado.
- Monitorea el ajuste automático de la capacidad del ASG y verifica que las instancias se añaden y eliminan según las políticas de escalado.

Resultados esperados:

- Configuración exitosa de un ASG.
- Políticas de escalado implementadas y activadas.
- Ajuste automático de la capacidad del ASG basado en la carga de trabajo.

Ejercicio 3: Configuración de un ALB con WAF para proteger una aplicación web

Objetivo: Configurar un ALB con un Web Application Firewall (WAF) para proteger una aplicación web.

Instrucciones:

- Inicia sesión en AWS Lab Learner.
- Crea un ALB y añade las instancias EC2 como objetivos del balanceador de carga.
- Crea un WAF y define reglas de protección (bloquear IPs específicas, prevenir ataques de SQL injection y XSS).
- Asocia el WAF con el ALB.
- Prueba la configuración accediendo al DNS del ALB e intentando realizar ataques simulados para verificar que el WAF bloquea correctamente las solicitudes maliciosas.

Resultados esperados:

- Configuración exitosa de un ALB con WAF.
- Implementación de reglas de protección en WAF.
- Verificación de la protección del WAF mediante pruebas de ataques simulados.

Ejercicio 4: Implementación de un network load balancer (NLB)

Objetivo: Configurar un NLB para manejar tráfico de red de baja latencia y alto rendimiento.

Instrucciones:

- Inicia sesión en AWS Lab Learner.
- Crea una VPC con subnets públicas en diferentes zonas de disponibilidad.
- Lanza varias instancias EC2 en las subnets configuradas, asegurándose de habilitar un servicio que escuche en un puerto específico (por ejemplo, un servidor TCP).
- Configura un NLB y añade las instancias EC2 como objetivos del balanceador de carga.
- Configura listeners en el NLB para enrutar el tráfico TCP a las instancias EC2.

- Prueba la configuración utilizando una herramienta de prueba de red (como netcat o telnet) para verificar que el NLB distribuye el tráfico TCP correctamente.

Resultados esperados:

- Configuración exitosa de un NLB.
- Distribución de tráfico TCP entre las instancias EC2.
- Verificación del balanceo de carga utilizando herramientas de prueba de red.

Ejercicio 5: Diseño de una solución Multi-Región con alta disponibilidad

Objetivo: Diseñar e implementar una arquitectura multi-región para asegurar alta disponibilidad.

Instrucciones:

- Inicia sesión en AWS Lab Learner.
- Crea dos VPCs en diferentes regiones, cada una con subnets públicas y privadas.
- Lanza instancias EC2 en cada VPC, configurando un servidor web básico en cada una.
- Configura un ALB en cada región y añade las instancias EC2 como objetivos.
- Configura Amazon Route 53 para enrutar el tráfico entre las dos regiones utilizando políticas de enrutamiento basadas en latencia o geolocalización.
- Simula una falla en una región y verifica que el tráfico se enruta automáticamente a la otra región.

Resultados esperados:

- Configuración exitosa de una arquitectura multi-región con alta disponibilidad.
- Implementación de políticas de enrutamiento en Amazon Route 53.
- Verificación del enrutamiento de tráfico y conmutación por error entre regiones.

Ejercicio 6: Configuración de Amazon Auto Scaling con políticas de escalado personalizadas

Objetivo: Configurar un sistema de Auto Scaling con políticas de escalado personalizadas para una aplicación.

Instrucciones:

- Inicia sesión en AWS Lab Learner.
- Crea una plantilla de lanzamiento que especifique la configuración de la instancia EC2.
- Configura un ASG utilizando la plantilla de lanzamiento, definiendo los parámetros de escalado.
- Establece políticas de escalado personalizadas basadas en métricas de aplicación específicas (como uso de memoria, tasa de solicitudes por segundo, etc.).
- Genera una carga de trabajo en las instancias EC2 para activar las políticas de escalado.

- Monitorea el ajuste automático de la capacidad del ASG y verifica que las instancias se añaden y eliminan según las políticas de escalado personalizadas.

Resultados esperados:

- Configuración exitosa de un ASG con políticas de escalado personalizadas.
- Implementación de políticas de escalado basadas en métricas de aplicación.
- Ajuste automático de la capacidad del ASG basado en las políticas de escalado personalizadas.

Código

Ejercicio 1: Simulación de escalabilidad vertical y horizontal

Objetivo: Entender las diferencias y aplicaciones de la escalabilidad vertical y horizontal.

Instrucciones:

- Implementa una clase Server que represente un servidor con atributos como cpu_capacity, memory_capacity, y current_load.
- Implementa un método scale_up para simular la escalabilidad vertical, aumentando la capacidad del servidor.
- Implementa una clase ServerCluster para simular la escalabilidad horizontal, con métodos para añadir o quitar servidores del clúster.
- Diseña un sistema que distribuya la carga entre los servidores de un clúster, simulando la escalabilidad horizontal.

```
class Server:
    def __init__(self, cpu_capacity, memory_capacity):
        self.cpu_capacity = cpu_capacity
        self.memory_capacity = memory_capacity
        self.current_load = 0

    def scale_up(self, additional_cpu, additional_memory):
        self.cpu_capacity += additional_cpu
        self.memory_capacity += additional_memory

class ServerCluster:
    def __init__(self):
        self.servers = []

    def add_server(self, server):
        self.servers.append(server)
```

```
    def remove_server(self):
        if self.servers:
            self.servers.pop()

    def distribute_load(self, load):
        if not self.servers:
            print("No servers available.")
            return
        load_per_server = load / len(self.servers)
        for server in self.servers:
            server.current_load += load_per_server

# Example usage
server1 = Server(cpu_capacity=4, memory_capacity=8)
server2 = Server(cpu_capacity=4, memory_capacity=8)
cluster = ServerCluster()
cluster.add_server(server1)
cluster.add_server(server2)
cluster.distribute_load(10)
```

Ejercicio 2: Resumen del modelo OSI

Objetivo: Comprender el modelo OSI y su aplicación en redes.

Instrucciones:

- Implementa una clase OSIModel que represente las siete capas del modelo OSI.
- Define métodos en cada capa para simular la transferencia de datos a través de la red.
- Implementa una función que muestre cómo un paquete de datos se mueve a través de las siete capas del modelo OSI.

```
class OSIModel:
    def __init__(self, data):
        self.data = data

    def application_layer(self):
        print("Application Layer: Processing data", self.data)
        self.data = "application_processed(" + self.data + ")"
        self.presentation_layer()

    def presentation_layer(self):
```

```
        print("Presentation Layer: Formatting data", self.data)
        self.data = "presentation_processed(" + self.data + ")"
        self.session_layer()

    def session_layer(self):
        print("Session Layer: Managing session for data", self.data)
        self.data = "session_processed(" + self.data + ")"
        self.transport_layer()

    def transport_layer(self):
        print("Transport Layer: Segmenting data", self.data)
        self.data = "transport_processed(" + self.data + ")"
        self.network_layer()

    def network_layer(self):
        print("Network Layer: Routing data", self.data)
        self.data = "network_processed(" + self.data + ")"
        self.data_link_layer()

    def data_link_layer(self):
        print("Data Link Layer: Framing data", self.data)
        self.data = "datalink_processed(" + self.data + ")"
        self.physical_layer()

    def physical_layer(self):
        print("Physical Layer: Transmitting data", self.data)
        self.data = "physical_processed(" + self.data + ")"
        print("Final data transmitted:", self.data)

# Example usage
osi_model = OSIModel(data="Hello World")
osi_model.application_layer()
```

Ejercicio 3: Simulación de balanceadores de carga

Objetivo: Simular la distribución de tráfico web utilizando balanceadores de carga.

Instrucciones:

- Implementa una clase LoadBalancer que distribuya las solicitudes entre varios servidores.
- Crea métodos para simular la configuración de un Application Load Balancer (ALB) y un Network Load Balancer (NLB).

- Implementa una función que distribuya solicitudes HTTP y TCP entre los servidores configurados.

```
class LoadBalancer:
    def __init__(self):
        self.servers = []

    def add_server(self, server):
        self.servers.append(server)

    def distribute_http_requests(self, requests):
        print("Distributing HTTP requests...")
        for i, request in enumerate(requests):
            server = self.servers[i % len(self.servers)]
            server.handle_request(request)

    def distribute_tcp_requests(self, requests):
        print("Distributing TCP requests...")
        for i, request in enumerate(requests):
            server = self.servers[i % len(self.servers)]
            server.handle_request(request)

class Server:
    def __init__(self, name):
        self.name = name

    def handle_request(self, request):
        print(f"Server {self.name} handling request: {request}")

# Example usage
server1 = Server("Server 1")
server2 = Server("Server 2")
load_balancer = LoadBalancer()
load_balancer.add_server(server1)
load_balancer.add_server(server2)
load_balancer.distribute_http_requests(["Request 1", "Request 2", "Request 3"])
load_balancer.distribute_tcp_requests(["TCP Request 1", "TCP Request 2"])
```

Ejercicio 4: Simulación de elasticidad con auto scaling groups

Objetivo: Implementar la elasticidad mediante la simulación de grupos de Auto Scaling.

Instrucciones:

- Implementa una clase AutoScalingGroup que gestione un grupo de servidores.
- Define métodos para simular la creación de plantillas de configuración y las políticas de escalado.
- Implementa funciones que ajusten dinámicamente el número de servidores en el grupo basado en métricas simuladas como uso de CPU o tráfico de red.

```
class AutoScalingGroup:
    def __init__(self, template, min_size, max_size):
        self.template = template
        self.min_size = min_size
        self.max_size = max_size
        self.servers = [self.template.create_server() for _ in range(min_size)]
        self.cpu_usage_threshold = 70

    def scale_out(self):
        if len(self.servers) < self.max_size:
            self.servers.append(self.template.create_server())
            print("Scaled out: Added a server")

    def scale_in(self):
        if len(self.servers) > self.min_size:
            self.servers.pop()
            print("Scaled in: Removed a server")

    def adjust_capacity(self, cpu_usages):
        average_cpu_usage = sum(cpu_usages) / len(cpu_usages)
        if average_cpu_usage > self.cpu_usage_threshold:
            self.scale_out()
        elif average_cpu_usage < self.cpu_usage_threshold:
            self.scale_in()

class ServerTemplate:
    def create_server(self):
        return Server()

class Server:
    pass

# Example usage
template = ServerTemplate()
asg = AutoScalingGroup(template=template, min_size=2, max_size=5)
asg.adjust_capacity([50, 60, 80])
```

Ejercicio 5: Diseño de soluciones de alta disponibilidad Multi-Región

Objetivo: Diseñar una arquitectura de alta disponibilidad multi-región.

Instrucciones:

- Implementa una clase Region que simule una región con sus propios servidores y balanceadores de carga.
- Define métodos para replicar datos entre regiones y gestionar el tráfico utilizando DNS.
- Implementa una función que simule el enrutamiento de solicitudes entre múltiples regiones, incluyendo la conmutación por error en caso de falla en una región.

```
class Region:
    def __init__(self, name):
        self.name = name
        self.servers = []
        self.load_balancer = LoadBalancer()

    def add_server(self, server):
        self.servers.append(server)
        self.load_balancer.add_server(server)

    def handle_request(self, request):
        self.load_balancer.distribute_http_requests([request])

class MultiRegionHA:
    def __init__(self):
        self.regions = []

    def add_region(self, region):
        self.regions.append(region)

    def route_request(self, request):
        # Simulate routing based on availability and performance
        primary_region = self.regions[0]
        secondary_region = self.regions[1]
        try:
            primary_region.handle_request(request)
        except Exception as e:
            print(f"Primary region failed: {e}")
            secondary_region.handle_request(request)
```

```
# Example usage
region1 = Region("us-east-1")
region2 = Region("us-west-1")
server1 = Server("Server 1")
server2 = Server("Server 2")
region1.add_server(server1)
region2.add_server(server2)
ha_system = MultiRegionHA()
ha_system.add_region(region1)
ha_system.add_region(region2)
ha_system.route_request("Request 1")
```

Ejercicio 6: Simulación de un sistema de firewall para aplicaciones Web (WAF)

Objetivo: Implementar un sistema de firewall para aplicaciones web.

Instrucciones:

- Implementa una clase WebApplicationFirewall que permita definir y aplicar reglas de protección.
- Define reglas para bloquear solicitudes basadas en IP, prevenir ataques de SQL injection y XSS.
- Implementa una función que aplique el WAF a un conjunto de servidores detrás de un balanceador de carga.

```
class WebApplicationFirewall:
    def __init__(self):
        self.rules = []

    def add_rule(self, rule):
        self.rules.append(rule)

    def apply_rules(self, request):
        for rule in self.rules:
            if not rule(request):
                print(f"Request blocked by rule: {rule.__name__}")
                return False
        return True

def block_ip_rule(ip):
    blocked_ips = ["192.168.0.1"]
    return ip not in blocked_ips
```

```
def sql_injection_rule(query):
    if any(keyword in query.lower() for keyword in ["select", "drop", "insert", "delete"]):
        return False
    return True

def xss_rule(content):
    if "<script>" in content.lower():
        return False
    return True

# Example usage
waf = WebApplicationFirewall()
waf.add_rule(lambda req: block_ip_rule(req["ip"]))
waf.add_rule(lambda req: sql_injection_rule(req["query"]))
waf.add_rule(lambda req: xss_rule(req["content"]))

request = {"ip": "192.168.0.2", "query": "SELECT * FROM users", "content": "Hello World"}
if waf.apply_rules(request):
    print("Request passed through WAF")
```

Ejercicio 7: Simulación de Auto Scaling con políticas de escalado personalizadas

Objetivo: Configurar un sistema de Auto Scaling con políticas de escalado personalizadas.

Instrucciones:

- Implementa una clase CustomAutoScalingGroup que gestione un grupo de servidores.
- Define métodos para simular la creación de plantillas de configuración y las políticas de escalado basadas en métricas personalizadas.
- Implementa funciones que ajusten dinámicamente el número de servidores en el grupo basado en métricas simuladas como uso de memoria, tasa de solicitudes por segundo, etc.

```
class CustomAutoScalingGroup:
    def __init__(self, template, min_size, max_size, scaling_policy):
        self.template = template
        self.min_size = min_size
        self.max_size = max_size
        self.scaling_policy = scaling_policy
        self.servers = [self.template.create_server() for _ in range(min_size)]

    def scale_out(self):
```

```
        if len(self.servers) < self.max_size:
            self.servers.append(self.template.create_server())
            print("Scaled out: Added a server")

    def scale_in(self):
        if len(self.servers) > self.min_size:
            self.servers.pop()
            print("Scaled in: Removed a server")

    def adjust_capacity(self, metric_values):
        if self.scaling_policy(metric_values):
            self.scale_out()
        else:
            self.scale_in()

class ServerTemplate:
    def create_server(self):
        return Server()

class Server:
    pass

def custom_scaling_policy(metric_values):
    # Example custom scaling policy
    return sum(metric_values) / len(metric_values) > 70

# Example usage
template = ServerTemplate()
asg = CustomAutoScalingGroup(template=template, min_size=2, max_size=5,
scaling_policy=custom_scaling_policy)
asg.adjust_capacity([50, 60, 80])
```

Ejercicio 8: Simulación de un sistema de escalabilidad vertical y horizontal completo

Objetivo: Desarrollar un sistema completo que soporte tanto la escalabilidad vertical como horizontal, simulando un entorno de servidor con balanceo de carga y ajuste automático de capacidad.

Instrucciones:

- Diseña una clase Server que represente un servidor con atributos como cpu_capacity, memory_capacity, y current_load.

- Implementa métodos en la clase Server para simular la escalabilidad vertical (scale_up) y para manejar solicitudes (handle_request).
- Crea una clase ServerCluster que administre un grupo de servidores, implementando métodos para añadir y remover servidores, y para distribuir la carga entre ellos.
- Implementa un balanceador de carga dentro de la clase ServerCluster que distribuya las solicitudes de manera eficiente entre los servidores disponibles.
- Diseña una clase AutoScaler que ajuste automáticamente la capacidad del clúster, añadiendo o removiendo servidores según métricas de uso como la CPU y la memoria.
- Implementa un script que simule la llegada de solicitudes y el ajuste dinámico de la capacidad del clúster, generando una carga de trabajo variable y observando cómo el sistema escala vertical y horizontalmente.

Ejercicio 9: Implementación completa del modelo OSI con simulación de tráfico de red

Objetivo: Crear una simulación detallada del modelo OSI que incluya todas las capas y permita el seguimiento de un paquete de datos a través de cada capa.

Instrucciones:

- Diseña una clase OSIModel que contenga métodos para cada una de las siete capas del modelo OSI (Aplicación, Presentación, Sesión, Transporte, Red, Enlace de Datos y Física).
- Implementa métodos específicos en cada capa para procesar y transformar datos, simulando tareas reales realizadas en esas capas (por ejemplo, segmentación en la capa de Transporte, enrutamiento en la capa de Red, etc.).
- Crea una estructura de datos que represente un paquete de red y que pueda ser modificado por cada capa del modelo OSI.
- Diseña un sistema de registro que documente el estado del paquete en cada etapa del procesamiento, permitiendo rastrear cómo cambia el paquete a medida que pasa por las diferentes capas.
- Implementa un script que genere paquetes de datos y los procese a través del modelo OSI, simulando la transmisión de datos entre un cliente y un servidor, e imprimiendo los detalles del procesamiento en cada capa.

Ejercicio 10: Simulación de balanceadores de carga con políticas de distribución avanzadas

Objetivo: Implementar un sistema de balanceo de carga avanzado que soporte múltiples tipos de balanceadores (ALB, NLB, CLB) y políticas de distribución personalizadas.

Instrucciones:

- Diseña una clase base LoadBalancer con métodos genéricos para añadir y remover servidores, y para distribuir solicitudes.
- Implementa clases derivadas ApplicationLoadBalancer, NetworkLoadBalancer, y ClassicLoadBalancer que extiendan la clase base y añadan funcionalidades específicas.

- Crea diferentes políticas de distribución de carga (round-robin, least-connections, IP-hash) y permite que cada tipo de balanceador pueda configurarse con cualquiera de estas políticas.
- Implementa un sistema de monitoreo que registre métricas de rendimiento y estadísticas de distribución de carga para cada balanceador.
- Diseña un script que configure un entorno con múltiples balanceadores y servidores, genere tráfico de red simulado, y ajuste dinámicamente las políticas de distribución para optimizar el rendimiento.

Ejercicio 11: Implementación de elasticidad con Auto Scaling Groups y políticas personalizadas

Objetivo: Desarrollar un sistema de Auto Scaling Group completo con políticas de escalado personalizadas y simulación de carga.

Instrucciones:

- Diseña una clase ServerTemplate que defina la configuración de las instancias de servidor.
- Implementa una clase AutoScalingGroup que gestione un grupo de servidores, incluyendo métodos para añadir y remover servidores basados en plantillas.
- Define políticas de escalado personalizadas que utilicen diferentes métricas de rendimiento (CPU, memoria, latencia de red) para decidir cuándo escalar hacia arriba o hacia abajo.
- Crea un simulador de carga que genere tráfico variable y permita observar cómo el Auto Scaling Group ajusta su capacidad en respuesta a los cambios en la carga.
- Implementa un sistema de registro que documente todas las decisiones de escalado y el estado del grupo de servidores en cada momento, permitiendo un análisis detallado del comportamiento del sistema bajo diferentes condiciones de carga.

Ejercicio 12: Diseño de una arquitectura Multi-Región con alta disponibilidad

Objetivo: Crear una arquitectura de alta disponibilidad multi-región que incluya replicación de datos, balanceo de carga global, y conmutación por error automática.

Instrucciones:

- Diseña una clase Region que represente una región con sus propios servidores, balanceadores de carga, y bases de datos.
- Implementa una clase GlobalLoadBalancer que distribuya el tráfico entre regiones basándose en políticas de enrutamiento como latencia, geolocalización, y disponibilidad.
- Define un sistema de replicación de datos que sincronice bases de datos entre diferentes regiones, garantizando consistencia y disponibilidad de datos.
- Implementa un mecanismo de conmutación por error que redirija el tráfico a una región secundaria en caso de falla de la región primaria, asegurando continuidad del servicio.
- Diseña un script que simule tráfico global y fallos de región, observando cómo el sistema maneja la distribución de carga y la conmutación por error, y generando informes detallados del rendimiento y la disponibilidad.

Ejercicio 13: Diseño de un sistema de balanceo de carga y escalabilidad para una aplicación Web de alto tráfico

Objetivo: Crear un sistema que combine balanceo de carga y autoescalado para manejar tráfico web de manera eficiente.

Instrucciones:

- Diseña una clase LoadBalancer que distribuya solicitudes entre múltiples instancias de servidor, implementando diferentes algoritmos de balanceo de carga como round-robin, least-connections, y weighted distribution.
- Implementa una clase AutoScalingGroup que gestione un grupo de instancias de servidor, incluyendo métodos para escalar hacia arriba y hacia abajo basado en métricas de rendimiento como uso de CPU, memoria, y tráfico de red.
- Integra el balanceador de carga con el grupo de autoescalado para ajustar dinámicamente el número de instancias activas según la carga del sistema.
- Desarrolla una función que monitoree el rendimiento del sistema y ajuste las políticas de escalado para optimizar el uso de recursos.
- Implementa un script que simule un entorno de alto tráfico web, generando diferentes patrones de carga y observando cómo el sistema maneja la distribución de solicitudes y el ajuste de capacidad, generando informes detallados sobre el rendimiento y la utilización de recursos.

Ejercicio 14: Simulación de un sistema de monitoreo y recuperación automática para alta disponibilidad

Objetivo: Desarrollar un sistema que monitoree la salud de los servidores y realice recuperaciones automáticas en caso de fallos.

Instrucciones:

- Diseña una clase HealthMonitor que supervise la salud de los servidores, utilizando métricas como tiempo de respuesta, uso de CPU, memoria, y latencia de red.
- Implementa una clase RecoveryManager que tome acciones automáticas en caso de detección de fallos, como reiniciar servidores, lanzar nuevas instancias, y redirigir tráfico.
- Define un mecanismo para registrar y reportar eventos de monitoreo y acciones de recuperación, proporcionando detalles sobre el estado del sistema y las medidas tomadas.
- Desarrolla un sistema de balanceo de carga que integre el monitor de salud y el gestor de recuperación, asegurando que el tráfico se dirija siempre a servidores saludables.
- Implementa un script que simule fallos en el sistema, observando cómo el monitor de salud detecta problemas y el gestor de recuperación toma acciones correctivas, generando

informes detallados sobre los eventos de monitoreo y las acciones de recuperación realizadas.