Practicas Tema 10

Arquitectura Software

AWS Lambda, Microsoft Azure y Google Cloud Platform son los servicios más usados del mundo para las arquitecturas Serverless y Arquitectura Event-Drive.

● Investiga estos 3 servicios y describe qué ofrece el servicio gratuito y de pago.

AWS

Las empresas optan por construir sus aplicaciones en AWS debido **a** su **amplitud y profundidad de los servicios**.

La rica gama de herramientas, incluyendo bases de datos, análisis, gestión, IoT, seguridad y aplicaciones empresariales, hace de AWS la solución correcta para muchos equipos. No es de extrañar que AWS tenga la porción más significativa del mercado de la nube.

Azure

Azure ha superado ligeramente a AWS en el porcentaje de empresas que lo utilizan (80% Azure vs. 77% AWS)

Azure también ofrece varios **servicios para las empresas**, y la relación de larga data de Microsoft con este segmento hace que sea una elección fácil para algunos clientes. Azure, Office 365 y Microsoft Teams permiten a las organizaciones

proporcionar a los empleados software empresarial al tiempo que aprovechan los recursos de computación en la nube.

Plataforma de Google Cloud

Azure y AWS tienen fuertes capacidades de aprendizaje automático.
Pero Google Cloud Platform destaca gracias a su investigación y experiencia interna casi ilimitadas, la magia que ha estado alimentando al gigante de los motores de búsqueda a lo largo de los años.

Lo que hace diferente a GCP es su papel en el desarrollo de diversas **tecnologías de código abierto**.
Estamos hablando especialmente de los contenedores y el papel central de Google en la construcción de Kubernetes para la orquestación y la malla de servicio de Istio, hoy en día prácticamente tecnologías estándar de la industria.

La cultura de innovación de Google se presta muy bien a las startups y empresas que priorizan tales enfoques y tecnologías.

Facturación en AWS vs. Azure vs. Plataforma de Google Cloud

Además de la facturación por minuto, AWS, Azure y Google Cloud apoyan la facturación por segundo para varios servicios. AWS presentó por primera vez facturación por segundo en 2017 para instancias basadas en EC2 Linux y volúmenes de EBS, pero hoy en día, se aplica a muchos otros servicios.

La facturación por segundo funciona con un límite mínimo de 60 segundos en AWS. Azure permite cargas por segundo en su plataforma en la nube, pero este modelo de facturación está disponible para todos los casos, en su mayoría basados en contenedores.

Google Cloud siguió a AWS en la introducción de la facturación por segundo y ahora lo ofrece por algo más que casos basados en Linux. Esta forma de facturación se aplica a todos los casos basados en VM.

Comparación de los precios de almacenamiento en la nube

En qué difieren estos principales proveedores de nube en términos de precios de almacenamiento?

Here es una comparación de los precios en regiones similares: AWS US East (Northern Virginia), Azure East US, y Virginia del Norte (us-este4) en Google Cloud Platform.

Proveedor de nubes	Almacenamiento (GB/Mes)
Amazon S3	0,023 dólares
Azure	0,021 dólares
Plataforma de Google Cloud	0,023 dólares
Almacenamiento de objetos de oráculos	$0.0255

1. **AWS Lambda:**
    - Gratis*:* 1 millón de solicitudes gratuitas y 400,000 GB-segundo de tiempo de cómputo por mes. Así como, 3.2 millones de segundos de tiempo de cómputo.
    - Pago*:* El precio se basa en la cantidad de solicitudes y la duración de la ejecución. También hay costos asociados con el uso de otros servicios vinculados, como almacenamiento y transferencia de datos.
2. **Microsoft Azure:**
    - Gratis*:* 1 millón de solicitudes gratuitas y 400,000 GB-segundos de tiempo de ejecución por mes. Además, proporciona 5 funciones gratuitas de consumo.
    - Pago*:* Como AWS , el coste depende en la cantidad de solicitudes y el tiempo de ejecución. Pudiendo incurrir en costos adicionales por el uso de más servicios.
3. **Google Cloud Platform:**
    - Gratis*:* 2 millones de invocaciones gratuitas, 1 millón mas que las otras dos posibilidades y 400,000 GB-segundos de tiempo de ejecución por mes. Además esta opción incluye 5 GB de almacenamiento y 2 millones de consultas gratuitas por mes para su base de datos NoSQL Firestore.
    - Pago*:* Los precios se basan en la cantidad de invocaciones, el tiempo de ejecución y los recursos consumidos durante la ejecución de las funciones.

● El problema de estos servicios es que piden demasiada información para usar la versión gratuita (datos personales, dirección e incluso método de pago para evitar bots) ¿Qué otros servicios podríamos usar de código abierto sin dar nuestra información? Busca información de al menos 2 servicios de este tipo.

Hemos buscado 2 optativas a estas tecnologías

OpenFaaS: Es un framework de funciones sin servidor de código abierto que se puede ejecutar en cualquier clúster de Kubernetes. No requiere información personal para su uso, ya que se puede implementar en entornos locales o en la nube sin la necesidad de proporcionar datos personales.

Kubeless: Es otra plataforma sin servidor basada en Kubernetes y de código abierto. Permite ejecutar funciones en cualquier clúster de Kubernetes sin la necesidad de proporcionar información personal. Al ser parte del ecosistema Kubernetes, puede ejecutarse en entornos locales o en la nube.

Estas dos opciones son aceptables por varias razones:

Portabilidad y Multi-nube: Puedes ejecutar OpenFaaS y Kubeless en entornos de Kubernetes, lo que permite la portabilidad entre proveedores de nube.

Independencia de proveedor: Evitas el bloqueo a un proveedor específico y puedes cambiar entre proveedores sin tener que reescribir tu código de funciones.

Control y Personalización: Ofrecen mayor control sobre la infraestructura y permiten personalizar la configuración de Kubernetes según tus necesidades.

Costos: En algunos casos, ejecutar funciones en entornos basados en Kubernetes puede ser más rentable que utilizar servicios nativos de la nube, dependiendo de la carga de trabajo y la estructura de precios.

Ecosistema de Kubernetes: Se benefician del rico ecosistema de Kubernetes, simplificando la integración con otras aplicaciones y servicios.

Flexibilidad en Tecnologías: Admiten una variedad de lenguajes y marcos de trabajo, brindando flexibilidad a los desarrolladores.

Principios SOLID

1 . El siguiente código viola el principio de responsabilidad única. Propón una

solución. Recuerda: Cada clase ha de tener un único propósito, responsabilidad y motivo

para el cambio.

```
class Empleado {
void calcularSalario() {
// lógica para calcular salario
```

```
}
void generarReporte() {
// lógica para generar reporte
}
}
```

Solución: Hemos separado las responsabilidades de calcular el salario y generar el informe en dos clases diferentes siguiendo el principio de responsabilidad única. Quedaría de la siguiente forma.

```
class Empleado {
    void calcularSalario() {
        // lógica para calcular salario
    }
}

class ReporteEmpleado {
    void generarReporte() {
        // lógica para generar reporte
    }
}
```

2. El siguiente código viola el principio de Abierto/Cerrado. Propón una solución.

Recuerda: Una clase ha de estar abierta para añadir más funcionalidades, pero cerrada

para ser modificada

El siguiente código viola el principio de Abierto/Cerrado. Propón una solución.
Recuerda: Una clase ha de estar abierta para añadir más funcionalidades, pero cerrada
para ser modificada

```
class Descuento {
double aplicarDescuento(double precio) {
// lógica para aplicar descuento directo
}
}
```

Solución: se introduce una interfaz `Descuento` y una implementación específica `DescuentoDirecto`. Ahora, la clase `Descuento` está abierta para extensiones (nuevas implementaciones) pero cerrada para modificaciones.

```
interface Descuento {
    double aplicarDescuento(double precio);
}

class DescuentoDirecto implements Descuento {
    @Override
    double aplicarDescuento(double precio) {
        // lógica para aplicar descuento directo
    }
}
```

3. El siguiente código viola el principio de sustitución de Liskov. Propón una solución. Recuerda: se debe aplicar correctamente la herencia

```
class Ave {
void volar() {
// lógica para volar
}
}
class Pinguino extends Ave {
void volar() {
// ¡Un pinguino no puede volar!
}
}
```

El siguiente código viola el principio de sustitución de Liskov. Propón una solución. Recuerda: se debe aplicar correctamente la herencia

class Ave {

void volar() {

// lógica para volar

}

}

class Pinguino extends Ave {

void volar() {

// ¡Un pinguino no puede volar!

}

}

Solución: Se ha modificado el método `volar` en la clase `Pinguino` para lanzar una excepción, indicando que un pingüino no puede volar.

```
class Ave {
    void volar() {
        // lógica para volar
    }
}

class Pinguino extends Ave {
    @Override
    void volar() {
        throw new UnsupportedOperationException("Un pinguino
    }
}
```

4. El siguiente código viola el principio de inversión de dependencias. Propón una solución. Recuerda: las clases deben ser modulares para probarlas con facilidad

```
class Switch {
Bombilla bombilla;
Switch(Bombilla bombilla) {
this.bombilla = bombilla;
}
void presionarBoton() {
if (bombilla.estaEncendida()) {
bombilla.apagar();
} else {
bombilla.encender();
}
```

```
}
}
class bombilla {
boolean estaEncendida() {
// lógica para verificar si la bombilla está encendida
}
void encender() {
// lógica para encender la bombilla
}
void apagar() {
// lógica para apagar la bombilla
}
}
```

Solución Se introdujo una interfaz `DispositivoEncendido` que define los métodos `estaEncendido`, `encender` y `apagar`. La clase `Bombilla` ahora implementa esta interfaz. La clase `Switch` ahora depende de la interfaz en lugar de una implementación concreta, siguiendo así el principio de inversión de dependencias.

```
interface DispositivoEncendido {
    boolean estaEncendido();
    void encender();
    void apagar();
}

class Bombilla implements DispositivoEncendido {
    private boolean encendida;

    @Override
    boolean estaEncendido() {
        return encendida;
    }

    @Override
    void encender() {
        encendida = true;
        // lógica para encender la bombilla
    }
```

```
    @Override
    void apagar() {
        encendida = false;
        // lógica para apagar la bombilla
    }
}

class Switch {
    DispositivoEncendido dispositivo;

    Switch(DispositivoEncendido dispositivo) {
        this.dispositivo = dispositivo;
    }

    void presionarBoton() {
        if (dispositivo.estaEncendido()) {
            dispositivo.apagar();
        } else {
            dispositivo.encender();
        }
    }
}
```