Taller 1: Construcción de pipelines en Cloud

Link del repositorio: https://github.com/Br14nMat/microservice-app-example.git

1. 2.5% Estrategia de branching para desarrolladores

Se utilizo la estrategia de branching GitFlow, la cual organiza el trabajo en ramas para facilitar el desarrollo colaborativo y control de versiones. Su estructura principal es:

- **main**: contiene siempre el código en producción.
- **develop**: rama base para el desarrollo. De aquí salen las demás ramas.
- **feature/***: se crean desde develop para nuevas funcionalidades. Se fusionan de nuevo en develop.
- **release/***: se crean desde develop para preparar una nueva versión. Al finalizar, se fusionan en main y develop.
- **hotfix/***: se crean desde main para corregir errores urgentes en producción. Se fusionan en main y develop.

2. 2.5% Estrategia de branching para operaciones

Se creó una rama principal llamada ops, la cual centraliza todo el código relacionado con las operaciones del proyecto. A partir de esta, se desprenden dos ramas especializadas:

- **infra/ops**: contiene la infraestructura como código, implementada con Terraform. Aquí se gestionan todos los recursos y configuraciones necesarias para los entornos del sistema.
- **pipelines/ops**: agrupa los cinco pipelines de desarrollo y el pipeline de infraestructura, facilitando la automatización y despliegue continuo desde una única rama organizada.

3. 15.0% Patrones de diseño de nube (mínimo dos)

1. **Patrón Sidecar (implementado con Redis)**

Concepto:

El patrón Sidecar consiste en desplegar componentes auxiliares junto a la aplicación principal en el mismo entorno de ejecución, compartiendo recursos, pero manteniendo una separación clara de responsabilidades.

Implementación en la infraestructura:

- Redis funciona como un servicio auxiliar (sidecar) desplegado en un contenedor aislado para el log_processor y todos_api
- Proporciona capacidades de cola/mensajería sin acoplamiento directo
- Configuración en la infraestructura:

```
redis = {
    name      = "redis"
    image     = "docker.io/redis:7.0"
    port      = 6379
    cpu       = 0.5
    memory    = "1Gi"
    min_replicas = 1
    max_replicas = 1
    env_vars = {}
  }
```

- Características de la implementación:
    - Servicio independiente pero compartido
    - Desacoplamiento del servicio principal
    - Mismo entorno de ejecución (microservices-env)
    - Recursos dedicados (0.5 CPU, 1Gi memoria)
    - Comunicación vía DNS interno

2. Patrón Bulkhead

Concepto:
Patrón de resiliencia que segmenta los recursos de la aplicación en compartimentos independientes, donde cada componente opera con sus propios recursos garantizados (CPU, memoria, réplicas), evitando que fallos o sobrecargas en un módulo afecten el funcionamiento del sistema completo

Implementación en la infraestructura:

1. **Agrupación lógica por servicios**:
    - **Críticos**: auth-api y users-api (mayor CPU/memoria)
    - **Estándar**: frontend y todos-api
    - **Background**: log-processor

2. **Mecanismos de aislamiento**:

    - Límites estrictos de recursos por servicio:

```
auth_api = {
  cpu    = 0.75  # 75% de un núcleo
  memory = "1.5Gi"
}
```

    - Escalado independiente:

```
min_replicas = 1
max_replicas = 4  # Para servicios críticos
```

3. **Resumen de Configuración por Grupos**:

    Servicios Críticos (Auth-API, Users-API)
    - **CPU**: 0.75 núcleos
    - **Memoria**: 1.5 GiB
    - **Réplicas**: 1-4 *(*alto escalamiento para disponibilidad*)*
    - **Propósito**: Garantizar operación continua de autenticación y gestión de usuarios

    Servicios Estándar (Frontend, Todos-API)
    - **CPU**: 0.5 núcleos
    - **Memoria**: 1 GiB
    - **Réplicas**: 1-3 *(Frontend)* / 1-2 (Todos-API)
    - **Propósito**: Balance entre rendimiento y costos para cargas moderadas

    Servicios Background (Log-Processor)
    - **CPU**: 0.25 núcleos
    - **Memoria**: 0.5 GiB
    - **Réplicas**: 1 *(fijo, no escalable)*
    - **Propósito**: Procesamiento asíncrono sin afectar servicios principales

Beneficios Clave de la Implementación

Para Sidecar (Redis):

- **Desacople**: Los microservicios no manejan directamente la cola de mensajes
- **Reutilización**: Mismo servicio Redis para múltiples componentes
- **Escalabilidad independiente**: Redis puede escalar sin afectar otros servicios
-

Para Bulkhead:

- **Aislamiento de fallos**: Un problema en log-processor no colapsa auth-api
- **Priorización de recursos**: Servicios críticos garantizan performance
- **Escalado selectivo**: Se puede manejar carga en componentes específicos

4. 15.0% Diagrama de arquitectura

5. 15.0% Pipelines de desarrollo (incluidos los scripts para las tareas que lo necesiten)

Los pipelines de Azure DevOps de desarrollo (auth-api, frontend, log-message-processor, todos-api, users-api) siguen una estructura común, ya que todos realizan la misma función básica: construir una imagen Docker a partir de un Dockerfile y subirla a un Azure Container Registry (ACR).

Estructura General de los Pipelines

- Todos los pipelines monitorean cambios en master.
- Cada pipeline solo se activa si hay modificaciones en su respectiva carpeta del microservicio.

Resources (Recursos)

Todos los pipelines usan el mismo repositorio como fuente.

Variables (Configuración Reutilizable)
Contienen parámetros clave para la construcción y despliegue de la imagen Docker

- dockerRegistryServiceConnection: ID de la conexión con ACR
- imageRepository: Nombre del repositorio de la imagen
- containerRegistry: URL del ACR destino
- dockerfilePath: Ruta al Dockerfile del microservicio
- tag: Tag de la imagen (usualmente el Build ID y latest)

Stages (Etapas del Pipeline)

Todos los pipelines tienen una única etapa (Build) con un solo job que ejecuta:

- **Build and Push (Construir y Subir Imagen Docker)**
- Usa la tarea Docker@2 para:
    1. Construir la imagen desde el Dockerfile.
    2. Subirla al ACR con dos tags:
        - **Build ID** (ej: 1234).
        - **latest** (siempre actualizado).

6. 5.0% Pipelines de infraestructura (incluidos los scripts para las tareas que lo necesiten)

Este pipeline se encarga de gestionar la infraestructura en Azure utilizando Terraform, con dos etapas principales:

1. **Bajar los recursos existentes** (Container Apps) antes de un nuevo despliegue.
2. **Desplegar infraestructura** desde cero usando Terraform.

Además, monitorea cambios en otros pipelines (microservicios) para posible re-ejecución.

Estructura General

Trigger (Activación)

Se ejecuta cuando hay cambios en:

- La carpeta /infra (definición de Terraform).
- Cualquier archivo YAML de los pipelines de microservicios.

Resources (Recursos)

Depende de otros pipelines (microservicios). Si alguno se ejecuta, este pipeline podría reaccionar

Variables

- **Credenciales de Terraform**: Usa un grupo de variables (terraform-creds).
- **Configuración de entorno**:
    - Versión de Terraform (1.5.5).
    - Grupo de recursos (taller1).
    - Entorno de Container Apps (microservices-env).

Stages (Etapas)

1. Destroy_Existing (Limpiar recursos previos)

- **Condición**: Se ejecuta en CI, manual o trigger.
- **Acción**:
    - Usa Azure CLI para eliminar todas las Container Apps en el grupo de recursos.
    - Si no existe el entorno, omite la destrucción.

2. Deploy_Infra (Desplegar con Terraform)

- **Depende de Destroy_Existing**.
- **Pasos**:
    1. Instala Terraform.
    2. Ejecuta terraform init, validate, plan y apply sobre la carpeta /infra.
    3. Usa variables de entorno para autenticación en Azure.

7. 20.0% Implementación de la infraestructura

Conceptos clave para la implementación:

¿Qué son Azure Container Apps?
Es un servicio serverless de Azure para ejecutar microservicios y aplicaciones en contenedores sin gestionar infraestructura subyacente (Kubernetes, nodos, etc.).

- **Ventajas**:
    - Escalado automático (hasta 0 réplicas si no hay tráfico).
    - Integración nativa con ACR (Azure Container Registry).
    - Redes privadas entre apps (service discovery automático).

¿Qué es un "Entorno" de Container Apps?

Es un espacio aislado donde se despliegan las aplicaciones contenerizadas:

- Proporciona red compartida y configuración de logging/monitoreo.
- Permite comunicación segura entre apps (ej: auth-api → users-api).

Decisión de Usar Container Apps

Se eligió esta tecnología porque:

1. **Simplifica el despliegue** de microservicios sin configurar Kubernetes (AKS).
2. **Escalado automático**: Cada app escala según demanda (definido en min_replicas/max_replicas).
3. **Service Discovery**: Las apps se comunican por nombre (ej: http://users-api:8083).

Infraestructura como Código (Terraform)

Proveedor y Configuración Básica

- Autenticación en Azure usando subscription_id.
- Reuso de recursos existentes:

```
data "azurerm_resource_group" "taller1" {}       # Grupo de recursos
data "azurerm_container_registry" "taller1" {}   # ACR para imágenes Docker
```

Creación del Entorno

Se define un entorno dedicado para agrupar todos los microservicios:

```
resource "azurerm_container_app_environment" "microservices" {
  name                = "microservices-env"
  resource_group_name = data.azurerm_resource_group.taller1.name
  location            = data.azurerm_resource_group.taller1.location
}
```

Container Apps (Microservicios)

Se definen 6 aplicaciones en 3 grupos para implementar el patron Bulkhead:

Grupo	Apps	Características
Servicios Críticos	auth-api, users-api	Mayor CPU/memoria (0.75 CPU, 1.5Gi). Escalado hasta 4 réplicas.
Frontend y APIs	frontend, todos-api	Configuración moderada (0.5 CPU, 1Gi).
Procesamiento	log-processor, redis	Bajo consumo (0.25 CPU). Redis para colas de mensajes.

Redis Sidecar

Para gestionar las colas de mensajes entre servicios, se implementó Redis como sidecar dentro del mismo Container App que los microservicios que lo requieren. Esta configuración permite:

El sidecar de Redis actúa como broker interno para el procesamiento asíncrono de mensajes, particularmente para el subsistema de logs, donde el servicio log-message-processor consume los mensajes almacenados en Redis.

Comunicación y Seguridad

- **Service Discovery**: Las apps se refieren entre sí por nombre (ej: http://users-api:8083).
- **Secrets**: Credenciales de ACR inyectadas automáticamente:

```
secret {
  name  = "acr-password"
  value = azurerm_container_registry.taller1.admin_password
}
```

8. 15.0% Demostración en vivo de cambios en el pipeline.

Video donde se modifica el index.html del frontend que dispara el pipeline del frontend que buildea la imagen y la sube al ACR. Luego el pipe de infraestructura detecta que se disparo el pipeline del frontend, entonces espera que se ejecuta para comenzar su ejecución en donde baja los container apps y despliega los modificados.

https://youtu.be/4UH4LawsMKQ

9. 10.0% Entrega de los resultados: debe incluir la documentación necesaria para todos los elementos desarrollados.

Entrega de Resultados y Documentación

El presente taller implementó una solución completa de microservicios en la nube utilizando Azure Container Apps, con pipelines automatizados y patrones de diseño cloud. Cada sección del reporte sirve como documentación técnica de los componentes desarrollados:

Gestión de Código

- **Estrategias de branching** documentadas en puntos 1-2 (GitFlow para desarrollo y ramas ops especializadas)
- **Repositorio único** con estructura clara: /infra (Terraform), los pipelines en el root (Azure DevOps), y carpetas por cada microservicio

Patrones de Diseño Implementados

- **Sidecar**: Redis como cola de mensajería (detalles técnicos en punto 3)
- **Bulkhead**: Segmentación de recursos (CPU/memoria/escalado) por grupos funcionales (punto 3 y 7)

Automatización CI/CD

- **5 pipelines de desarrollo** (punto 5): Construcción y push de imágenes Docker a ACR
- **Pipeline de infraestructura** (punto 6): Gestión de recursos con Terraform (destroy/apply)
- **Integración entre pipelines**: Trigger cruzado al modificar infraestructura o microservicios

Infraestructura como Código

- **Terraform completo** en /infra (punto 7):
    - Entorno de Container Apps (microservices-env)
    - 6 aplicaciones contenerizadas con configuración granular
    - Salidas (outputs) con URLs de acceso
- **Variables sensibles** gestionadas mediante grupos en Azure DevOps