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

The GitFlow branching strategy was used, which organizes work into branches to facilitate collaborative development and version control. Its main structure is:

- **main**: always contains the code in production.
- **develop**: base branch for development. The other branches branch off from here.
- **feature**/*: created from develop for new features. They are merged back into develop.
- **release**/*: created from develop to prepare a new version. Upon completion, they are merged into main and develop.
- **hotfix**/*: created from main to fix urgent bugs in production. They are merged into main and develop.

2. 2.5% Estrategia de branching para operaciones

A main branch called ops was created, which centralizes all the code related to the project's operations. Two specialized branches branch off from this branch:

- infra/ops: Contains the infrastructure as code, implemented with Terraform. All the resources and configurations necessary for the system environments are managed here.

- pipelines/ops: Groups the five development pipelines and the infrastructure pipeline, facilitating automation and continuous deployment from a single, organized branch.

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

1. Sidecar Pattern (implemented with Redis)

Concept:

The Sidecar pattern consists of deploying auxiliary components alongside the main application in the same execution environment, sharing resources but maintaining a clear separation of responsibilities.

Infrastructure Deployment:

- Redis functions as an auxiliary service (sidecar) deployed in an isolated container for the log_processor and todos_api.
- Provides queuing/messaging capabilities without direct coupling.
- Infrastructure Configuration:

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

= 1 max_replicas = 1
env_vars = {}
}

Implementation characteristics:

- Independent but shared service
- Decoupling from the main service
- Same execution environment (microservices-env)
- Dedicated resources (0.5 CPU, 1GB of memory)
- Communication via internal DNS**2.**

Patrón Bulkhead

Concept:

Resilience pattern that segments application resources into independent compartments, where each component operates with its own guaranteed resources (CPU, memory, replicas), preventing failures or overloads in one module from affecting the operation of the entire system

Infrastructure Implementation:

1. Logical grouping by services:

   - Critical: auth-api and users-api (highest CPU/memory)
   - Standard: frontend and all-api
   - Background: log-processor

2. Isolation mechanisms:

   - Strict resource limits per service**:**

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

- Independent scaling:

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

2. **Configuration Summary by Group:**

Critical Services (Auth-API, Users-API)

- **CPU:** 0.75 cores
- **Memory:** 1.5 GiB
- **Replicas:** 1-4 (high scaling for availability)
- **Purpose:** Ensure continuous operation of authentication and user management

Standard Services (Frontend, All-API)

- CPU**:** 0.5 cores
- **Memory**: 1 GiB
- **Replicas**: 1-3 (Frontend) / 1-2 (All-API)
- **Purpose**: Balance between performance and costs for moderate loads

Background Services (Log-Processor)

- **CPU**: 0.25 cores
- **Memory**: 0.5 GiB
- **Replicas**: 1 (fixed, non-scalable)
- **Purpose**: Asynchronous processing without affecting core services

Key Implementation Benefits

For Sidecar (Redis):

- **Decoupling**: Microservices do not directly manage the message queue
- **Reusability**: Same Redis service for multiple components
- **Independent scalability**: Redis can scale without affecting other services

For Bulkhead:

- **Fault isolation**: A log-processor issue does not crash auth-api
- **Resource prioritization**: Critical services guarantee performance
- **Selective scaling**: Load can be managed on specific components

4. 15.0% Diagrama de arquitectura https://github.com/Br14nMat/microserviceapp example/blob/master/docs/diagrama_deployment.pdf

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

The Azure DevOps development pipelines (auth-api, frontend, log-message-processor, todosapi, users-api) follow a common structure, as they all perform the same basic function: building a Docker image from a Dockerfile and uploading it to an Azure Container Registry (ACR).

General Pipeline Structure

- All pipelines monitor changes in master.
- Each pipeline is only triggered if there are changes in its respective microservice folder.Resources

(Resources)

All pipelines use the same repository as a source.

Variables (Reusable Configuration)

Contain key parameters for building and deploying the Docker image.

- dockerRegistryServiceConnection: Connection ID to the ACR
- mageRepository: Name of the image repository
- containerRegistry: URL of the target ACR
- dockerfilePath: Path to the microservice's Dockerfile - tag: Image tag (usually the Build ID and latest)

Stages (Pipeline Stages)

All pipelines have a single stage (Build) with a single job that executes:

Build and Push (Build and Push Docker Image)

- Use the Docker@2 task to:

1. Build the image from the Dockerfile.
2. Push it to the ACR with two tags:
    - Build ID (e.g., 1234).
    - latest (always up-to-date).

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

This pipeline manages infrastructure in Azure using Terraform, with two main stages:

1. **Deploy existing resources** (container apps) before a new deployment.
2. **Deploy infrastructure** from scratch using Terraform.

Additionally, it monitors changes in other pipelines (microservices) for possible re-execution.

General Structure

Trigger (Activation)
Executes when there are changes to:

• The /infra folder (Terraform definition).
• Any YAML files in the microservices pipelines.

Resources (Recursos)

It depends on other pipelines (microservices). If any of them are running, this pipeline might react.

Variables

- Terraform Credentials: Use a variable group (terraformcreds).

- **Environment Configuration:**

- Terraform version (1.5.5).
- Resource group (workshop1).
- Container Apps environment (microservicesenv).

Stages (Etapas)

1. Destroy_Existing (Clean up previous resources)

Condition: Runs as a CI, manual, or trigger.

Action:

- Use Azure CLI to delete all Container Apps in the resource group.
- If the environment doesn't exist, skip the destruction.

2. Deploy_Infra (Deploy with Terraform)

Depends on Destroy_Existing.
Steps:
1. Install Terraform.
2. Run terraform init, validate, plan, and apply on the /infra folder.
3. Use environment variables for authentication in Azure.

6. 20.0% Implementación de la infraestructura

Key concepts for implementation:

What are Azure Container Apps?

It's an Azure serverless service for running microservices and containerized applications without managing underlying infrastructure (Kubernetes, nodes, etc.).

• Advantages:

- Automatic scaling (down to 0 replicas if there's no traffic).
- Native integration with ACR (Azure Container Registry).
- Private networking between apps (automatic service discovery).

What is a Container Apps "Environment"?

It is an isolated space where containerized applications are deployed:
- It provides shared networking and logging/monitoring configuration.
- It allows secure communication between apps (e.g., auth-api → users-api).

Decision to Use Container Apps

This technology was chosen because:

1. It simplifies the deployment of microservices without configuring Kubernetes (AKS).
2. Automatic scaling: Each app scales on demand (defined in min_replicas/max_replicas).
3. Service Discovery: Apps communicate by name (e.g., http://users-api:8083).

Infrastructure as Code (Terraform)

Provider and Basic Configuration

- Authentication to Azure using subscription_id.
- Reuse of existing resources:

```
data "azurerm_resource_group" "taller1" {}  # Data resource group
"azurerm_container_registry" "taller1" {} # ACR for Docker images
```

Creating the Environment

A dedicated environment is defined to house all the microservices:

```
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)

6 applications are defined in 3 groups to implement the Bulkhead pattern:

Grupo	**Apps**	**Características**
Critial Services	auth-api, users-api	Increased CPU/memory (0.75 CPU, 1.5Gi). Scalable up to 4 replicas.
Frontend y APIs	frontend, todos-api	Moderate configuration (0.5 CPU, 1Gi).
Prosecution	log-processor, redis	Low power consumption (0.25 CPU). Redis for message queues.

Redis Sidecar

To manage message queues between services, Redis was implemented as a sidecar within the same Container App as the microservices that require it. This configuration allows:

The Redis sidecar acts as an internal broker for asynchronous message processing, particularly for the log subsystem, where the log-message-processor service consumes messages stored in Redis.

Communication and Security

- **Service Discovery**: Apps refer to each other by name (e.g., **http://usersapi:8083).**
- **Secrets:** Automatically injected ACR credentials:

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

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

Video where the frontend's index.html is modified, triggering the frontend pipeline that builds the image and uploads it to ACR. The infrastructure pipeline then detects that the frontend pipeline has been triggered, then waits for it to execute before beginning its execution, where it downloads the container apps and deploys the modified ones.

https://youtu.be/4UH4LawsMKQ

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

Delivery of Results and Documentation

This workshop implemented a complete cloud microservices solution using Azure Container Apps, with automated pipelines and cloud design patterns. Each section of the report serves as technical documentation for the developed components:

Code Management

- Branching strategies documented in steps 1-2 (GitFlow for development and specialized ops branches)
- Single repository with a clear structure: /infra (Terraform), pipelines in the root (Azure DevOps), and folders for each microservice

Implemented Design Patterns

- Sidecar: Redis as a messaging queue (technical details in point 3)
- Bulkhead: Resource segmentation (CPU/memory/scaling) by functional groups (points 3 and 7)

CI/CD Automation

• **5 development pipelines** (point 5): Building and pushing Docker images to ACR

• **Infrastructure pipeline** (point 6): Resource management with Terraform (destroy/apply)

• **Pipeline integration:** Cross-trigger when modifying infrastructure or microservices

Infrastructure as Code

Full Terraform in /infra (point 7):

- - Container Apps environment (microservices-env)
- - 6 containerized applications with granular configuration
- - Outputs with access URLs

• **Sensitive variables** managed through groups in Azure DevOps