Modernización a microservicios nativos en la Nube con Microsoft Azure Kubernetes Service

Nombre: Steven Vallejo Sacoto

Profesión: Tecnólogo Superior en Ciberseguridad

Fecha del informe: 14/10/2025

Descripción:

Este proyecto se realizó utilizando **Fedora (Linux)**.
Se simula la modernización de una aplicación monolítica "on-premise" a una arquitectura de microservicios nativa en la nube utilizando una prueba gratuita de Azure. Se despliega una aplicación de varios niveles (frontend, API de negocio, base de datos) en **Azure Kubernetes Service (AKS)**. La infraestructura completa, incluyendo las redes, balanceadores de carga y el clúster de AKS, se definirá y desplegará usando **Bicep (IaC)**. Se implementará **Azure Monitor for Containers** para obtener visibilidad total del rendimiento y la salud de la aplicación.

Próximamente actualización del índice

Resumen ejecutivo y análisis de costo

Resumen de servicios implementados

Se ha desplegado con éxito una arquitectura de microservicios nativa en la nube en Microsoft Azure, simulando la modernización de una aplicación monolítica. La solución es resiliente, escalable y observable.

Arquitectura y Servicios Clave:

Servicio de Azure	Propósito en el Proyecto	Tipo de Servicio
Azure Kubernetes Service (AKS)	Orquestador de contenedores. Gestiona el ciclo de vida de la aplicación.	PaaS (Contenedores)
Azure Container Registry (ACR)	Registro privado para almacenar y gestionar las imágenes de Docker de forma segura.	PaaS
Azure Virtual Network (VNet)	Proporciona una red privada y aislada para el clúster de AKS, mejorando la seguridad.	IaaS (Redes)
Azure Load Balancer	Creado automáticamente por el Service de Kubernetes para exponer la aplicación a Internet.	PaaS (Redes)

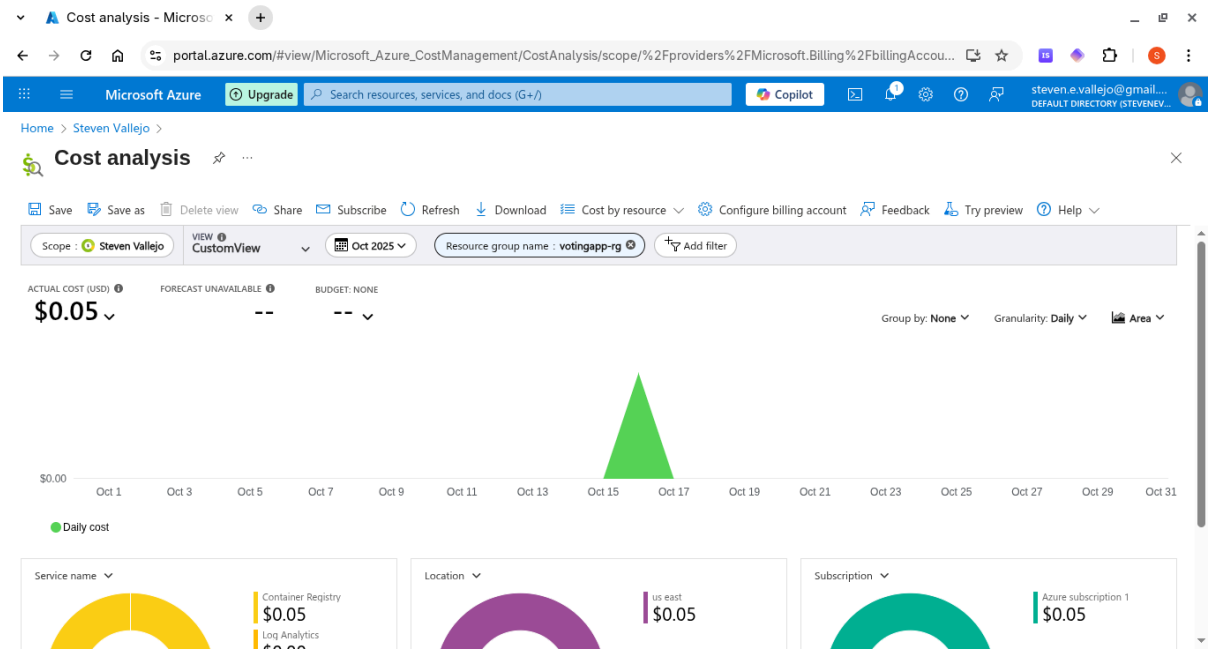
Azure Monitor for Containers	Solución de monitoreo que proporciona visibilidad total sobre el rendimiento y la salud del clúster y los contenedores.	PaaS (Monitoreo)
Log Analytics Workspace	Repositorio centralizado para almacenar logs y métricas de todos los componentes.	PaaS
Identidades Administradas	Utilizadas para que AKS se autentique de forma segura en ACR sin necesidad de contraseñas.	PaaS (Identidad)
Bicep (Infraestructura como Código)	Lenguaje declarativo usado para automatizar el despliegue de toda la infraestructura.	-

Análisis de Costos del Proyecto

La gestión de costos fue un pilar fundamental del proyecto. La estrategia se centró en utilizar recursos económicos y aplicar prácticas operativas para minimizar el gasto. El costo del proyecto fue de **$0.05**

Conclusiones Clave de Costos:

- **El Clúster de AKS es el principal motor de costos.** Al usar VMs de la serie B (B2s) y detener el clúster cuando no se usa es una medida efectiva.
- Los servicios PaaS como ACR y Log Analytics son muy rentables a pequeña escala.
- La automatización con Bicep no tiene costo directo, pero ahorra costos indirectos al evitar errores humanos y permitir despliegues y limpiezas rápidas.



Proyecto: Modernización a microservicios nativos en la nube con Microsoft Azure Kubernetes Service

Objetivo: Demostrar una migración exitosa de un monolito a una arquitectura de microservicios resiliente, escalable y observable en Azure, utilizando prácticas de Infraestructura como Código (IaC) para garantizar la repetibilidad y la gobernanza.

Fase 0: Arquitectura y planificación

1. **La aplicación a modernizar (Simulada):**
    - Utilizaré una aplicación de ejemplo de **Aplicación de Votación** estándar que simula perfectamente un entorno de microservicios para no perder el tiempo en dependencias.
    - **Microservicio 1 (Frontend):** Una aplicación web simple (Python/Flask o Node.js/React) que permite a los usuarios votar.
    - **Microservicio 2 (API de Backend):** Una API REST (Python/FastAPI o Node.js/Express) que recibe los votos del frontend y los persiste.
    - **Componente de datos (Base de datos):** Una base de datos simple en memoria como **Redis** para almacenar los votos. Se desplegará también como un contenedor dentro de Kubernetes para simplificar la gestión en este proyecto.

2. **Arquitectura de la solución en Azure:**
    - **Orquestación: Azure Kubernetes Service (AKS)** será el centro de la operación, gestionando el ciclo de vida de nuestros contenedores.
    - **Registro de imágenes: Azure Container Registry (ACR)** será nuestro repositorio privado y seguro para almacenar las imágenes de Docker de cada microservicio.
    - **Redes:** Crearemos una **Red Virtual (VNet)** dedicada con subredes específicas para el clúster de AKS, siguiendo buenas prácticas de aislamiento.
    - **Automatización:** Utilizaremos **Bicep** como único lenguaje para definir y desplegar toda la infraestructura. El objetivo es un despliegue manos libres.
    - **Observabilidad: Azure Monitor for Containers** (habilitado a través de un Log Analytics Workspace) nos ayudará a monitorear la salud y el rendimiento.

3. **Estrategia de gestión de presupuesto y tiempo:**
    - **Control de costos:**
        - **SKU de VMs:** Para los nodos de AKS, utilizaremos la serie **B (B-series)**, que es la más económica y puede "acumular" CPU cuando está inactiva y usarla en ráfagas.
        - **Apagado:** El clúster de AKS es el principal consumidor de crédito. **Lo detendremos al final de cada día de trabajo en el caso que se requiera** con el comando az aks stop.
        - **Limpieza:**Todos los recursos se crearán en un único **grupo de recursos**. Al finalizar el proyecto, eliminaremos este grupo para detener instantáneamente todo el consumo de crédito.

Plan de Ejecución

- **Fase 1 - Preparación y containerización.**
    - **Punto 1:** Configurar el entorno de desarrollo local (Azure CLI, Docker Desktop, VS Code, kubectl).
    - **Punto 2:** Escribir los Dockerfiles para el microservicio frontend y la API de backend.
    - **Punto 3:** Construir las imágenes localmente y verificar que funcionen (docker build, docker run).

- **Fase 2 - Infraestructura como Código con Bicep.**
    - **Punto 1:** Desarrollar el archivo Bicep principal (main.bicep).
    - **Punto 2:** Crear módulos de Bicep para la red (VNet), el registro (ACR) y el clúster (AKS) para mantener el código limpio.
    - **Punto 3:** Desplegar la infraestructura en Azure usando **az deployment group create**. Depurar y validar que todos los recursos se creen correctamente.

- **Fase 3 - Despliegue de la Aplicación en AKS.**
    - **Punto 1:** Subir las imágenes de Docker a Azure Container Registry (docker push).
    - **Punto 2:** Escribir los manifiestos de Kubernetes (.yaml):
        - Deployment para el frontend, la API y Redis.
        - Service para exponer la API internamente y el frontend externamente (usando un tipo LoadBalancer).
    - **Punto 3:** Conectarse al clúster (az aks get-credentials) y desplegar la aplicación (kubectl apply).

- **Fase 4 - Validación, Monitoreo y Pruebas.**
    - **Punto 1:** Verificar que la aplicación esté funcionando accediendo a la IP pública del Service del frontend.
    - **Punto 2:** Simular una falla (ej. eliminando un pod de la API) y ver cómo Kubernetes lo recrea automáticamente, demostrando la alta disponibilidad. Se crearán alertas de que pondremos a prueba (Ej: Error HTMl)
    - **Punto 3:** Demostrar la escalabilidad (Escalado horizontal)
    - **Punto 4:** Explorar Azure Monitor for Containers (Observabilidad)

- **Fase 5 - Limpieza.**
    - **Punto 1: Eliminar el grupo de Recursos**

Fase 1: Preparación del entorno (Fedora) y Containerización

Objetivo: Tener las imágenes de Docker de los microservicios construidas, probadas localmente en Fedora y listas para ser subidas a un registro. Costo de Azure en esta fase: **$0**.

Punto 1.1: Configuración del entorno de desarrollo en Fedora

1. **Instalar Azure CLI:** Utilizaremos el gestor de paquetes dnf y el repositorio oficial de Microsoft.
    - Abre una terminal en Fedora.

    Importamos la clave de Microsoft GPG:

    ```
    sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
    ```

    Añadimos el repositorio de Azure CLI:

    ```
    sudo dnf install -y dnf-plugins-core

    sudo dnf config-manager --add-repo
    https://packages.microsoft.com/yumrepos/azure-cli
    ```

    Instalamos Azure CLI:

    ```
    sudo dnf install -y azure-cli
    ```

    - **Verificación:** Cierra y vuelve a abrir tu terminal, luego ejecuta `az --version`.
    - **Inicio de sesión:** Ejecuta `az login` para autenticarte en tu cuenta de Azure.

```
stevenvallejo@fedora:/$ az --version
azure-cli                         2.68.0 *

core                              2.68.0 *
telemetry                          1.1.0

Dependencies:
msal                              1.32.3
azure-mgmt-resource               23.1.1

Python location '/usr/bin/python3'
Extensions directory '/home/stevenvallejo/.azure/cliextensions'

Python (Linux) 3.13.7 (main, Aug 14 2025, 00:00:00) [GCC 15.2.1 20250808 (Red Hat 15.2.1-1)]

Legal docs and information: aka.ms/AzureCliLegal


You have 2 update(s) available. Consider updating your CLI installation with 'az upgrade'
stevenvallejo@fedora:/$ az login
/usr/lib/python3.13/site-packages/azure/cli/core/aaz/_command.py:132: FutureWarning: functools.partial will be a method descriptor in future Python ve
rsions; wrap it in staticmethod() if you want to preserve the old behavior
  if self.AZ_PREVIEW_INFO:
/usr/lib/python3.13/site-packages/azure/cli/core/aaz/_command.py:133: FutureWarning: functools.partial will be a method descriptor in future Python ve
rsions; wrap it in staticmethod() if you want to preserve the old behavior
  self.preview_info = self.AZ_PREVIEW_INFO(cli_ctx=self.cli_ctx)
/usr/lib/python3.13/site-packages/azure/cli/core/aaz/_command.py:50: FutureWarning: functools.partial will be a method descriptor in future Python ver
sions; wrap it in staticmethod() if you want to preserve the old behavior
  if self.AZ_PREVIEW_INFO:
/usr/lib/python3.13/site-packages/azure/cli/core/aaz/_command.py:51: FutureWarning: functools.partial will be a method descriptor in future Python ver
sions; wrap it in staticmethod() if you want to preserve the old behavior
  self.group_kwargs['preview_info'] = self.AZ_PREVIEW_INFO(cli_ctx=self.cli_ctx)
A web browser has been opened at https://login.microsoftonline.com/organizations/oauth2/v2.0/authorize. Please continue the login in the web browser.
If no web browser is available or if the web browser fails to open, use device code flow with `az login --use-device-code`.
```

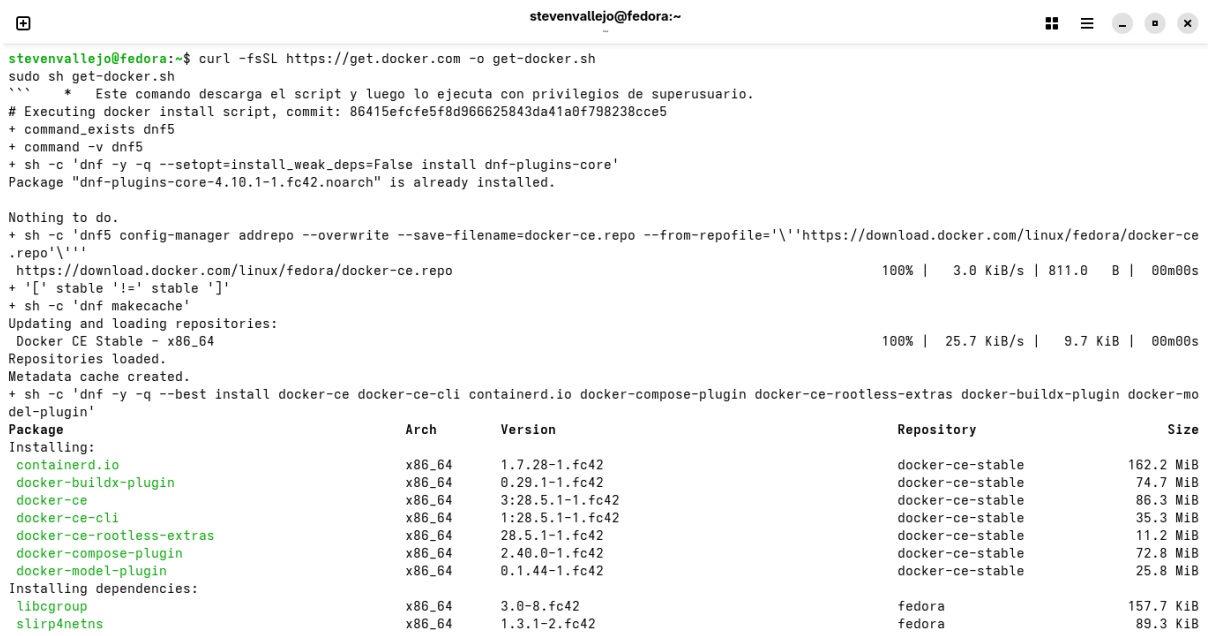
2. **Limpieza inicial (Si es que se lo requiere)**

sudo dnf remove docker docker-client docker-client-latest docker-common docker-latest docker-latest-logrotate docker-logrotate docker-engine

Instalamos Docker usando script oficial:

expand_less curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh

* Este comando descarga el script y luego lo ejecuta con privilegios de superusuario.



3. **Iniciar, habilitar y configurar Docker:**

   Iniciamos el servicio de Docker:

   ```
   sudo systemctl start docker
   ```

   Habilitamos para que inicie con el sistema:

   ```
   sudo systemctl enable docker
   ```

   Añadimos un usuario al grupo docker: Esto te permitirá ejecutar comandos de Docker sin sudo.

   ```
   sudo usermod -aG docker $USER
   ```

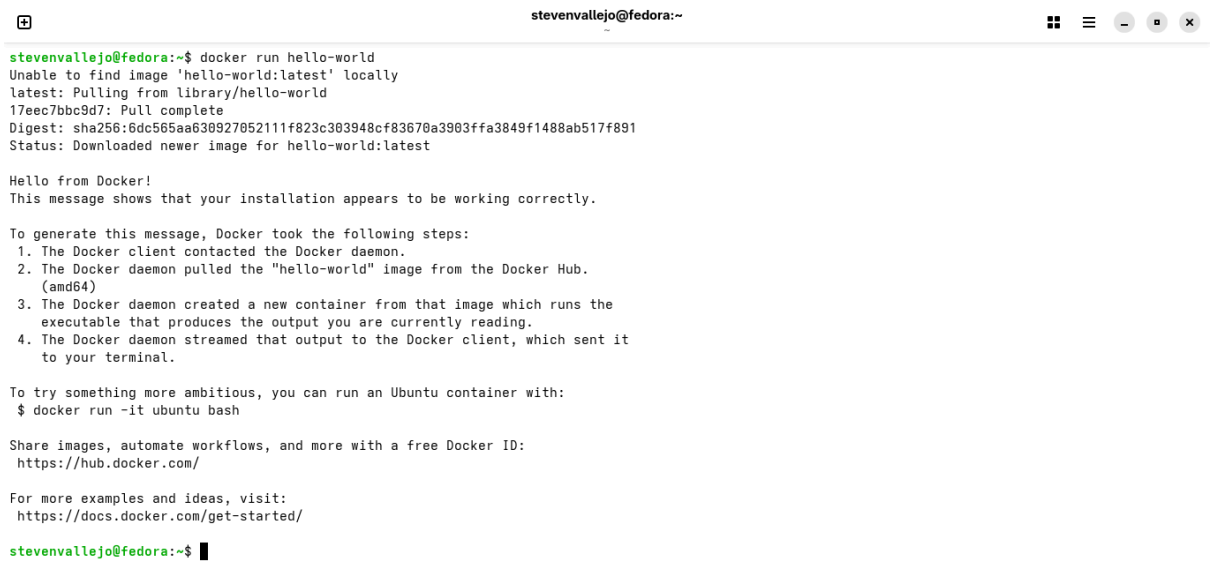
   Este cambio requiere que **cierres tu sesión actual y vuelvas a iniciarla** para que tu membresía al grupo se actualice.

Verificación:

Después de volver a iniciar sesión, abre una nueva terminal.

Ejecuta el siguiente comando **sin sudo** para comprobar los cambios:

docker run hello-world



4. Instalar Visual Studio Code: Importar la clave y añadir el repositorio de Microsoft:

```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
```

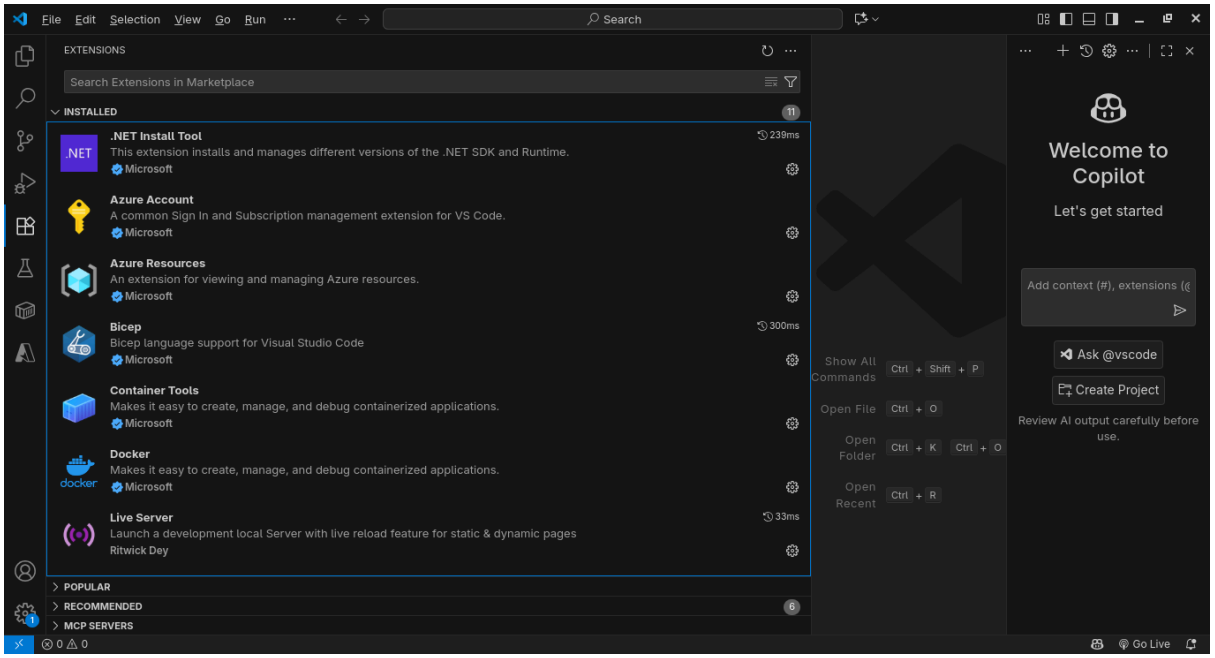
```
printf "[vscode]\nname=Visual Studio Code\nbaseurl=https://packages.microsoft.com/yumrepos/vscode\nenabled=1\ngpgcheck=1\ngpgkey=https://packages.microsoft.com/keys/microsoft.asc" | sudo tee /etc/yum.repos.d/vscode.repo > /dev/null
```

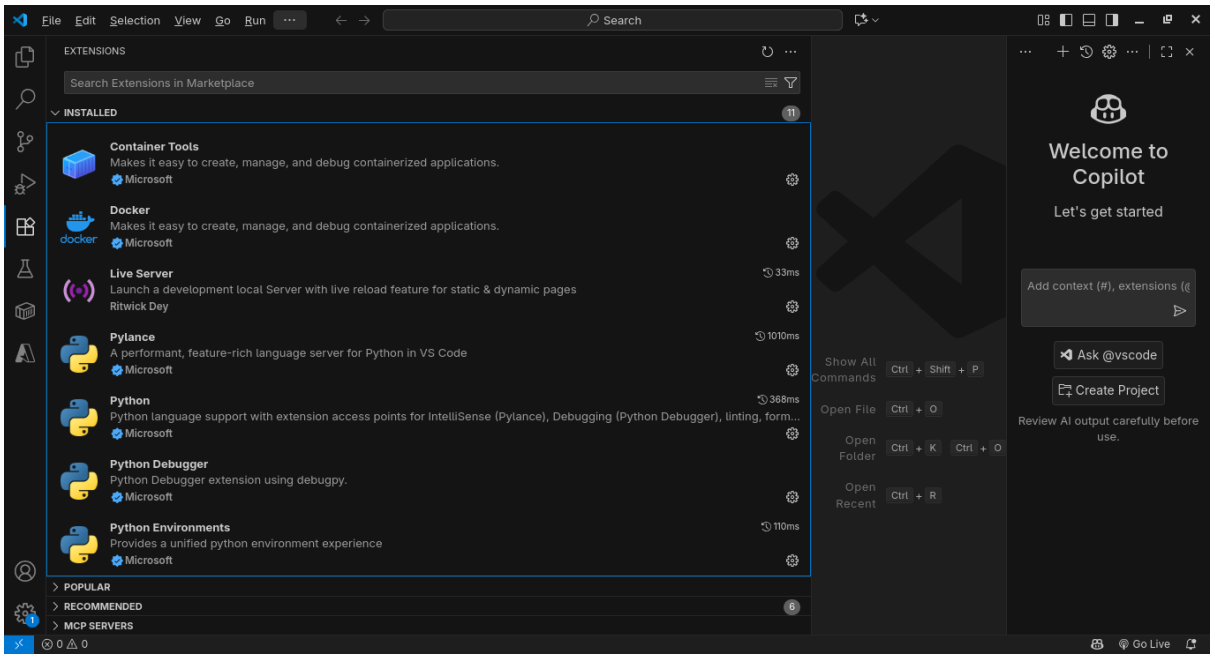
Instalamos VS Code:

```
sudo dnf check-update
```

```
sudo dnf install -y code
```

Dentro de VS Code, instalamos las extensiones **Docker, Azure Account, Bicep, y Python**




File Edit Selection View Go Run
Search
EXTENSIONS
Search Extensions in Marketplace
INSTALLED 11
Container Tools
Makes it easy to create, manage, and debug containerized applications.
Microsoft
Docker
Makes it easy to create, manage, and debug containerized applications.
Microsoft
Live Server 33ms
Launch a development local Server with live reload feature for static & dynamic pages
Ritwick Dey
Pylance 1010ms
A performant, feature-rich language server for Python in VS Code
Microsoft
Python 368ms
Python language support with extension access points for IntelliSense (Pylance), Debugging (Python Debugger), linting, form...
Microsoft
Python Debugger
Python Debugger extension using debugpy.
Microsoft
Python Environments 110ms
Provides a unified python environment experience
Microsoft
POPULAR
RECOMMENDED 8
MCP SERVERS
Show All Commands Ctrl + Shift + P
Open File Ctrl + O
Open Folder Ctrl + K Ctrl + O
Open Recent Ctrl + R
Welcome to Copilot
Let's get started
Add context (#), extensions (@
Ask @vscode
Create Project
Review AI output carefully before use.
0 0
Go Live

5. **Instalar kubectl:** Usaremos Azure CLI, ya que es el método más simple y es independiente del sistema operativo.

```
sudo az aks install-cli
```

Verificación: Ejecuta kubectl version --client.

Ahora **Fedora** tiene **az**, **docker**, **code** y **kubectl** para trabajar y nuestro comando docker se ejecuta sin sudo.

Punto 1.2: Crear y "Containerizar" los microservicios

Objetivo: Crear la estructura de carpetas y todos los archivos de código fuente y de configuración de Docker para nuestros dos microservicios: backend y frontend.

Estructura del proyecto:

```
aks-modernization/
├── frontend/
│   ├── app.py
│   ├── requirements.txt
│   ├── templates/
│   │   └── index.html
│   └── Dockerfile
└── backend/
    ├── app.py
    ├── requirements.txt
    └── Dockerfile
```

Crear la Estructura de directorios

Primero, abre tu terminal. Asegúrate de estar en tu directorio home o donde quieras crear el proyecto.

1. **Creamos la carpeta principal del proyecto:**

   ```
   mkdir aks-modernization
   cd aks-modernization
   ```

2. **Creamos las carpetas para cada microservicio y sus plantillas:**

   ```
   mkdir backend

   mkdir frontend

   mkdir frontend/templates
   ```

3. **Verifica la estructura:** Ejecuta el comando ls -R. Te saldrá esto:

Crear los archivos del microservicio backend

Navega a la carpeta del backend: cd backend.

1. **Archivo de dependencias (requirements.txt):**
    - Crea el archivo. Usamos el comando touch requirements.txt y luego abrirlo en VS Code, o directamente crear el archivo desde VS Code.

    Pega lo siguiente:

    ```
    # backend/requirements.txt
    fastapi
    uvicorn[standard]
    redis
    ```

        - **fastapi**: El framework para construir nuestra API.
        - **uvicorn**: El servidor que ejecutará nuestra aplicación FastAPI.
        - **redis**: La librería de Python para comunicarnos con la base de datos Redis.

2. **Archivo de la Aplicación (app.py):**

    - Crea el archivo con touch app.py.

    Pega el siguiente código:

    ```
    # backend/app.py

    import os

    from fastapi import FastAPI

    from redis import Redis

    app = FastAPI()



    # Conexión a Redis. El nombre 'redis-service' será el que usemos en Kubernetes.
    ```

```
# Para pruebas locales, usaremos 'localhost' por defecto.
redis_host = os.environ.get('REDIS_HOST', 'localhost')
redis_conn = Redis(host=redis_host, port=6379, db=0,
decode_responses=True)

@app.get("/")
def read_root():
return {"message": "Backend API is running"}

@app.post("/vote/{option}")
def cast_vote(option: str):
# Incrementa el contador para la opción votada ('dogs' o 'cats')
count = redis_conn.incr(option)
return {"option": option, "count": count}

@app.get("/results")
def get_results():
dogs_count = redis_conn.get('dogs') or 0
cats_count = redis_conn.get('cats') or 0
return {"dogs": dogs_count, "cats": cats_count}
```

3. **El Plano del Contenedor (Dockerfile):**

   Crea un archivo llamado Dockerfile (sin extensión).

   Pega el siguiente contenido:

```
# Build

FROM python:3.9-slim as builder


# Previene que Python escriba archivos .pyc y almacene en búfer la salida

ENV PYTHONDONTWRITEBYTECODE 1

ENV PYTHONUNBUFFERED 1


WORKDIR /app

COPY requirements.txt .


# Crear un entorno virtual, activarlo y usarlo para instalar dependencias

RUN python -m venv /opt/venv

ENV PATH="/opt/venv/bin:$PATH"

RUN pip install --no-cache-dir -r requirements.txt


# Final

FROM python:3.9-slim


WORKDIR /app
```

```
# Copia el entorno virtual completo de la fase de construcción
COPY --from=builder /opt/venv /opt/venv

# Copia el código fuente de la aplicación
COPY . .

# Activa el entorno virtual para que el CMD pueda encontrar uvicorn
ENV PATH="/opt/venv/bin:$PATH"

EXPOSE 80
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "80"]
```

Se **crearon** los **3 archivos** necesarios para el **servicio de API**. La carpeta backend está completa.

Crear los archivos del microservicio frontend

Regresamos a la carpeta principal (cd ..) y luego entra en la del frontend (cd frontend).

1. **Archivo de dependencias (requirements.txt):**
    - Crea el archivo requirements.txt.

    Pega el siguiente texto:

    ```
    # frontend/requirements.txt
    flask
    requests
    ```

    - **flask**: Un framework web simple para servir nuestra página HTML.
    - **requests**: La librería que usaremos para hacer llamadas a nuestra API de backend.

2. **Plantilla HTML (templates/index.html):**
    - Navega a la carpeta templates.
    - Crea el archivo index.html.

    código HTML:

```
  <!-- frontend/templates/index.html -->
<!DOCTYPE html>
<html>
<head>
   <title>Azure Voting App</title>
   <style>
      body { font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Helvetica, Arial, sans-serif; text-align: center; background-color: #f0f8ff; margin-top: 50px; }
```

```
        .container { max-width: 600px; margin: auto; padding: 20px; background: white; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); }
        button { background-color: #0078d4; color: white; border: none; padding: 15px 30px; margin: 10px; border-radius: 5px; cursor: pointer; font-size: 16px; transition: background-color 0.2s; }
        button:hover { background-color: #005a9e; }
        h2 { color: #333; }
        #results { font-size: 18px; color: #555; margin-top: 20px; }
    </style>
</head>
<body>
    <div class="container">
        <h2>¿Qué prefieres?</h2>
        <form action="/vote" method="post">
            <button name="vote" value="cats">Gatos</button>
            <button name="vote" value="dogs">Perros</button>
        </form>
        <div id="results">
            <p>Gatos: <b>{{ results.cats }}</b> | Perros: <b>{{ results.dogs }}</b></p>
        </div>
    </div>
</body>
</html>
```

3. **Archivo de la aplicación (app.py):**
    - Regresa a la carpeta frontend (cd ..).
    - Crea el archivo app.py.

Código:

```
# frontend/app.py
import os
import requests
from flask import Flask, render_template, request, redirect, url_for

app = Flask(__name__)


# La URL del backend será inyectada como variable de entorno en Kubernetes.

# Para pruebas locales, apuntará al puerto del backend que expondremos.

backend_url = os.environ.get('BACKEND_API_URL', 'http://localhost:8000')


@app.route("/")

def index():

    try:

        # Obtiene los resultados actuales de la API de backend

        response = requests.get(f"{backend_url}/results")

        results = response.json()

    except requests.exceptions.ConnectionError:

        print(f"ERROR: No se pudo conectar a la API de backend en {backend_url}")

        results = {"dogs": "N/A", "cats": "N/A"}
```

```
    return render_template('index.html', results=results)


@app.route("/vote", methods=['POST'])
def vote():
    option = request.form['vote']
    try:
        # Envia el voto a la API de backend
        requests.post(f"{backend_url}/vote/{option}")
    except requests.exceptions.ConnectionError:
        print(f"ERROR: No se pudo conectar a la API de backend en {backend_url}")

    return redirect(url_for('index'))


if __name__ == '__main__':
    # Puerto 8080.
    app.run(host='0.0.0.0', port=8080, debug=True)
```

4. **El Plano del Contenedor (Dockerfile):**
    - Crea el archivo Dockerfile en frontend/.

Contenido:

```
# Build
FROM python:3.9-slim as builder


# Previene que Python escriba archivos .pyc y almacene en búfer la salida
```

```
ENV PYTHONDONTWRITEBYTECODE 1

ENV PYTHONUNBUFFERED 1


WORKDIR /app

COPY requirements.txt .


# Crea un entorno virtual, activarlo y usarlo para instalar dependencias

RUN python -m venv /opt/venv

ENV PATH="/opt/venv/bin:$PATH"

RUN pip install --no-cache-dir -r requirements.txt


# Final

FROM python:3.9-slim


WORKDIR /app


# Copia el entorno virtual completo de la fase de construcción

COPY --from=builder /opt/venv /opt/venv


# Copia el código fuente de la aplicación

COPY . .
```

Activar el entorno virtual

ENV PATH="/opt/venv/bin:$PATH"

EXPOSE 8080

CMD ["python", "app.py"]

Punto 1.3: Construir y probar las imágenes localmente

Objetivo: Compilar los Dockerfiles en imágenes de contenedor funcionales y ejecutarlas juntas en tu máquina Fedora para verificar que la aplicación completa funciona antes de tocar Azure.

Construir la imagen del backend

1. Abre tu terminal y ve al directorio aks-modernization.

   Navega a la carpeta del backend:

   ```
   cd backend/
   ```

2. Ejecuta el comando de construcción de Docker:

   ```
   docker build -t voting-app-backend .
   ```

   - docker build: Comando para construir una imagen.
   - -t voting-app-backend: La opción -t (tag) le da un nombre legible (voting-app-backend) a laimagen.
   - .: Le dice a Docker que use el Dockerfile que se encuentra en el directorio actual.

3. Esto puede tardar un minuto la primera vez, ya que descargó la imagen base de Python.

Construir la imagen del frontend

1. Regresa a la carpeta raíz del proyecto y navega a la del frontend:

   cd ../frontend/

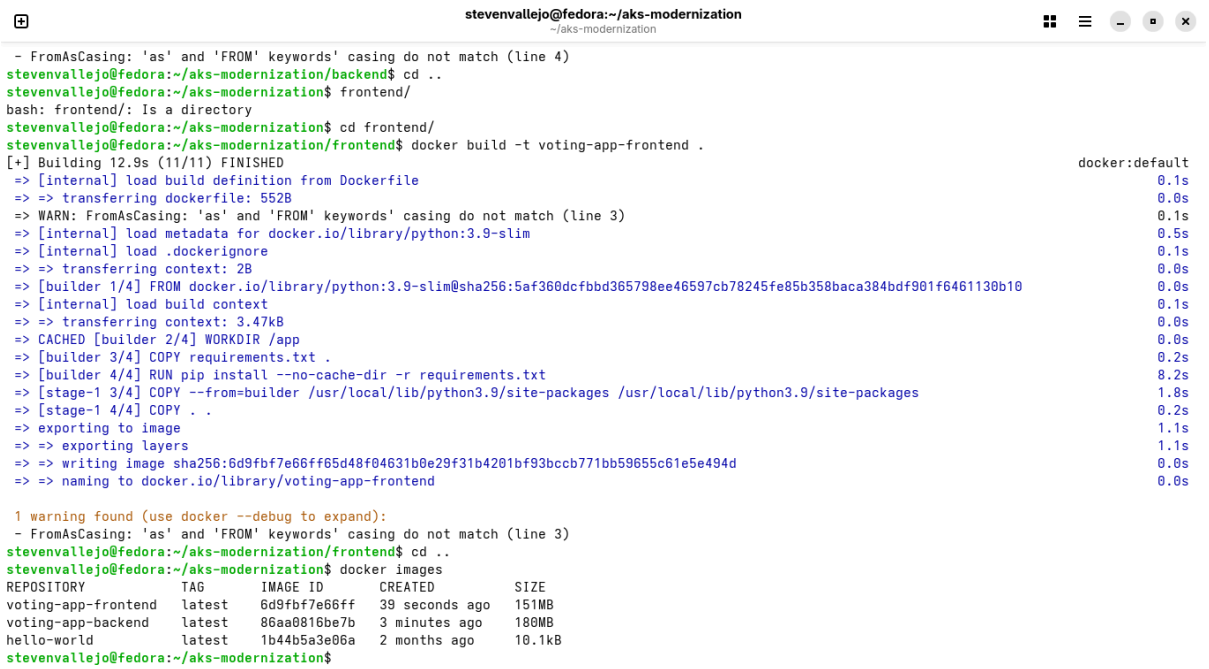
2. Ejecuta el comando de construcción para el frontend:

   docker build -t voting-app-frontend .

Verificar que las imágenes se crearon

Ejecuta el siguiente comando para listar todas las imágenes de Docker en la terminal:

docker images



Deberías ver voting-app-frontend y voting-app-backend en la parte superior de la lista.

Si ves las **dos imágenes** en la lista, significa que la construcción se realizó con **éxito**

Ejecutar la aplicación completa localmente

Lanzaremos los tres contenedores para que trabajen juntos: la base de datos Redis, el backend y frontend.

1. **Iniciar el contenedor de redis:**
   Usaremos una imagen oficial de Redis de Docker Hub. Docker la descargará automáticamente.

   ```
   docker run --name redis -p 6379:6379 -d redis:alpine
   ```

   - --name redis: Nombra al contenedor "redis" para identificarlo fácilmente.
   - -p 6379:6379: Mapea el puerto 6379 de la computadora al puerto 6379 dentro del contenedor.
   - -d: Modo "detached", para que se ejecute en segundo plano.
   - redis:alpine: La imagen de Redis, versión alpine muy ligera.

2. **Iniciar el contenedor del backend:**

   ```
   docker run --name backend-container -p 8000:80 -d voting-app-backend
   ```

   - --name backend-container: Nombra al contenedor.
   - -p 8000:80: Mapea el puerto 8000 de la computadora al puerto 80 del contenedor donde está escuchando uvicorn. El código de frontend buscará la API en localhost:8000.

3. **Iniciar el Contenedor del Frontend:**

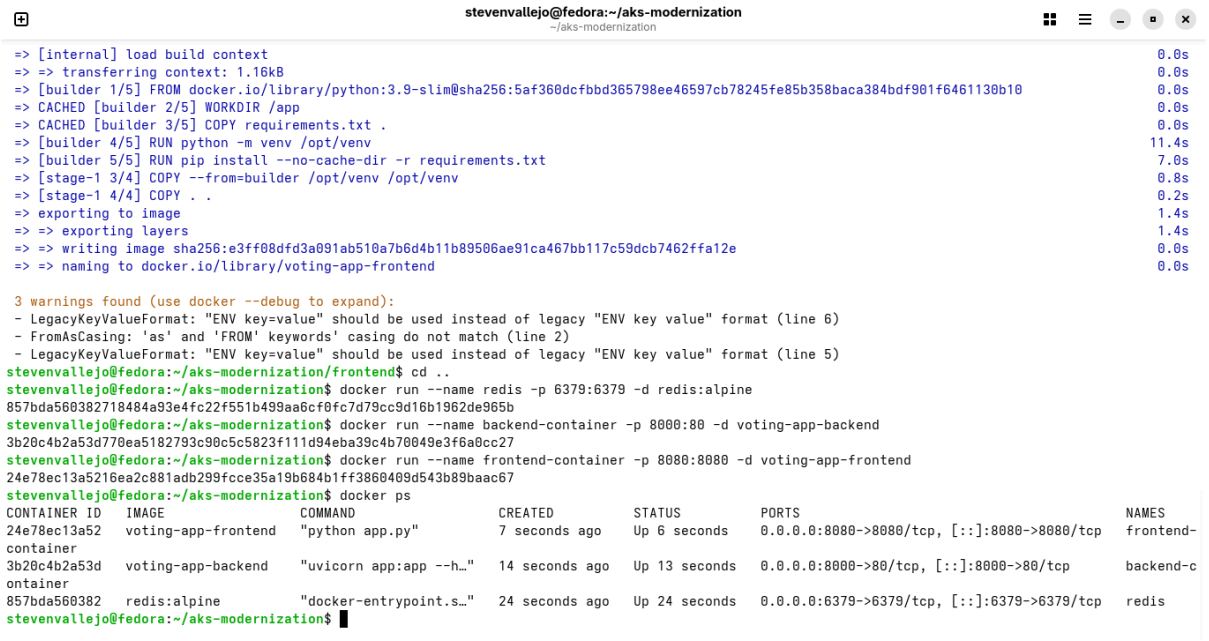
   ```
   docker run --name frontend-container -p 8080:8080 -d voting-app-frontend
   ```

   - --name frontend-container: Nombra al contenedor.
   - -p 8080:8080: Mapea el puerto 8080 de la computadora al puerto 8080 del contenedor donde escucha Flask. Este es el puerto que abriremos en el navegador.

4. **Verificar que los contenedores están corriendo:**

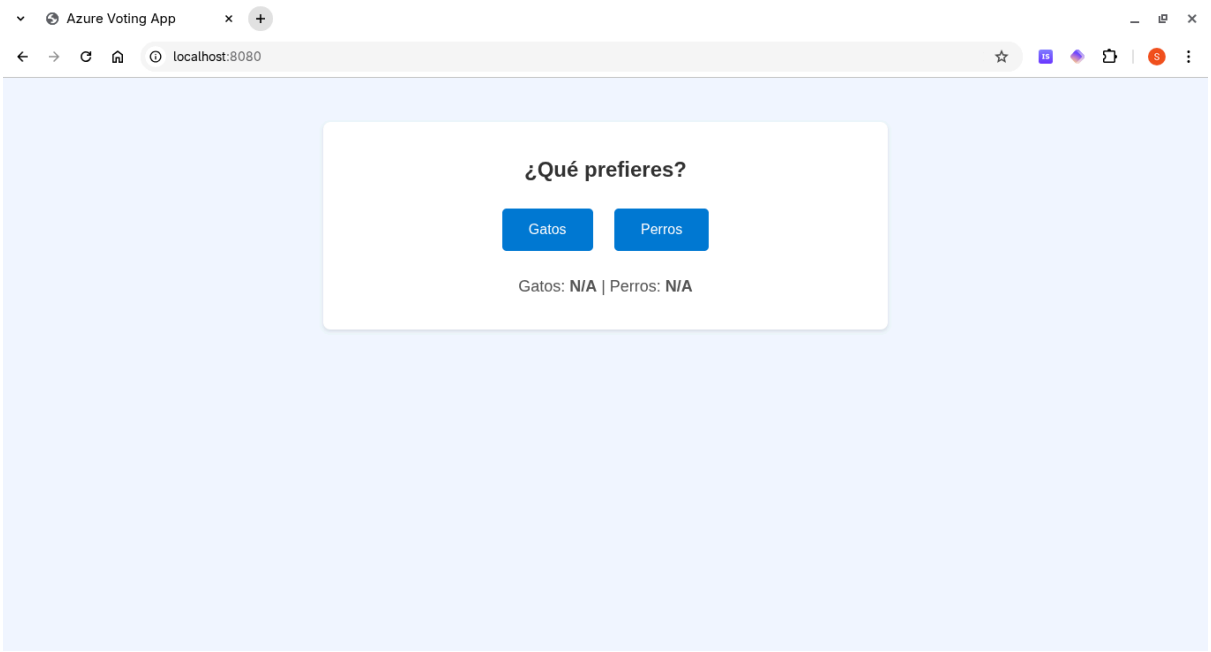
docker ps

Deberías ver una lista con los tres contenedores (redis, backend-container, frontend-container) en estado "Up".



Verificación

1. Abre tu navegador web.
2. Ve a la dirección: http://localhost:8080



Limpieza del entorno

Dejaremos nuestro espacio de trabajo limpio.

Detenemos los contenedores con:

```
docker stop frontend-container backend-container redis
```

Eliminar los contenedores con:

```
docker rm frontend-container backend-container redis
```

```
stevenvallejo@fedora:~/aks-modernization$ docker stop frontend-container backend-container redis
frontend-container
backend-container
redis
stevenvallejo@fedora:~/aks-modernization$ docker rm frontend-container backend-container redis
frontend-container
backend-container
redis
stevenvallejo@fedora:~/aks-modernization$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
stevenvallejo@fedora:~/aks-modernization$
```

Fase 2: Infraestructura como código (IaC) con Bicep

Objetivo: Definir y desplegar toda la infraestructura de Azure (Red, Registro de Contenedores, Clúster de Kubernetes) utilizando Bicep. Al finalizar esta fase, tendremos un clúster de AKS vacío, monitoreado y listo para recibir nuestra aplicación. **Advertencia: A partir de este momento, iniciamos el consumo del crédito de Azure.**

Punto 2.1: Definir la arquitectura y estructura del proyecto

1. **Convenciones de nomenclatura:** Usaremos un nombre base para agrupar visualmente todos nuestros recursos en el portal de Azure.
    - **Nombre base:** votingapp
    - **Grupo de recursos:** votingapp-rg
    - **Red virtual:** votingapp-vnet
    - **Registro de contenedores (ACR):** votingappacr<stevenvallejo10190>. **Importante:** El nombre de ACR debe ser único a nivel global en todo Azure y solo puede contener letras minúsculas y números.
    - **Clúster de Kubernetes (AKS):** votingapp-aks

2. **Estructura de Archivos Bicep:** La modularidad es clave para la reutilización y la legibilidad. Dentro de tu carpeta aks-modernization, crea la siguiente estructura:

    - Primero, crea la carpeta infra, asegúrate de estar en la raíz del proyecto: aks-modernization/

        ```
        mkdir infra

        cd infra
        ```

    - Ahora, crea la carpeta de módulos y los archivos vacíos:

        ```
        mkdir modules
        ```

touch main.bicep main.parameters.json modules/acr.bicep modules/aks.bicep modules/vnet.bicep

Verifica la estructura: Ejecuta ls -R

- **main.bicep**: Será el orquestador que llamará a nuestros módulos.
- **main.parameters.json**: Separará la configuración (nombres, ubicaciones) del código, permitiendo desplegar el mismo código en diferentes entornos (ej. dev, prod) solo cambiando este archivo.
- **modules/**: Contendrá nuestros ladrillos de Bicep, cada uno responsable de un único recurso.

Punto 2.2: Escribir el código de la infraestructura (Bicep)

Abre la carpeta infra en VS Code para una mejor edición.

1. El archivo de parámetros (main.parameters.json)

Este archivo define el "qué" y el "dónde".

Pega el siguiente contenido en infra/main.parameters.json:

{

```
 "$schema":
"https://schema.management.azure.com/schemas/2019-04-01/deploymentParam
eters.json#",
 "contentVersion": "1.0.0.0",
 "parameters": {
   "resourceNamePrefix": {
     "value": "votingapp"
   },
   "location": {
     "value": "eastus"
   },
   "acrName": {
     "value": "votingappacrsv2510"
   }
 }
}
```

}

- **Cambia** votingappacr<REEMPLAZAR> por tu nombre único.

2. El Módulo de Red (modules/vnet.bicep)

Creamos una red aislada para nuestro clúster.

Contenido en infra/modules/vnet.bicep:

```
// Define el nombre de la red virtual que se creará
param vnetName string

// Define la ubicación (región de Azure) donde se creará la red
param location string

// Define el nombre de la subred para los nodos de AKS
param aksSubnetName string = 'aks-subnet'

// Creación del recurso de la Red Virtual (VNet)
resource vnet 'Microsoft.Network/virtualNetworks@2023-05-01' = {
 name: vnetName
 location: location
 properties: {
   addressSpace: {
     addressPrefixes: [
       '10.0.0.0/16' // Rango de IPs principal para toda la red
     ]
   }
   subnets: [
     {
       name: aksSubnetName
```

```
      properties: {
        addressPrefix: '10.0.0.0/24' // Rango de IPs específico para los nodos de AKS
      }
    }
  ]
}

// 'output' expone información del recurso creado para que otros módulos puedan usarla.
// Exponemos el ID de la subred, que el clúster de AKS necesitará.
output aksSubnetId string = vnet.properties.subnets[0].id
```

3. El módulo de registro de contenedores (modules/acr.bicep)

El almacén privado para nuestras imágenes de Docker.

Contenido en infra/modules/acr.bicep:

```
// Define el nombre único global para el Azure Container Registry (ACR)
param acrName string

// Define la ubicación (debe ser la misma que la del resto de recursos)
param location string

// Creación del recurso ACR
```

```
resource acr 'Microsoft.ContainerRegistry/registries@2023-07-01' = {
 name: acrName
 location: location
 // SKU 'Basic' es el más económico y suficiente para este proyecto
 sku: {
   name: 'Basic'
 }
 properties: {
   // Habilitamos el usuario administrador. Facilita el inicio de sesión desde la CLI local,
   adminUserEnabled: true
 }
}

// Exponemos el ID del ACR. El clúster de AKS lo necesita para obtener permisos de extracción.
output acrId string = acr.id
```

4. El Módulo del Clúster de Kubernetes (modules/aks.bicep)

El cerebro de nuestra operación. Este es el módulo más complejo.

Contenido en infra/modules/aks.bicep:

```
// Define el nombre para el clúster de AKS

param clusterName string



// Define la ubicación

param location string



// Recibe el ID de la subred que creamos en el módulo vnet.bicep

param aksSubnetId string



// CORRECCIÓN: Nuevo parámetro para recibir el ID del Log Analytics
Workspace

param logAnalyticsWorkspaceId string



// Creación del recurso Clúster de Kubernetes (AKS)

resource aksCluster
'Microsoft.ContainerService/managedClusters@2023-09-01' = {

 name: clusterName

 location: location

 identity: {

   type: 'SystemAssigned'

 }

 properties: {

   dnsPrefix: clusterName
```

```
  agentPoolProfiles: [
    {
      name: 'agentpool'
      count: 1
      vmSize: 'Standard_B2s'
      osType: 'Linux'
      mode: 'System'
      vnetSubnetID: aksSubnetId
    }
  ]
  networkProfile: {
    networkPlugin: 'azure'
    serviceCidr: '10.1.0.0/16'
    dnsServiceIP: '10.1.0.10'
  }
  // Habilitamos el monitoreo con Azure Monitor for Containers
  addonProfiles: {
    omsagent: {
      enabled: true
      config: {
        // CORRECCIÓN: Usamos el ID del workspace que nos pasan como
parámetro
        logAnalyticsWorkspaceResourceID: logAnalyticsWorkspaceId
      }
    }
```

```
    }
  }
}

// Exponemos el Principal ID de la identidad del clúster.
// Lo necesitamos para asignarle roles.
output aksPrincipalId string = aksCluster.identity.principalId
```

5. Crear el módulo loganalytics.bicep

Este archivo es el almacén de logs

Contenido en infra/loganalytics.bicep:

```
// Define el nombre para el Log Analytics Workspace
param workspaceName string

// Define la ubicación
param location string

// Creación del recurso Log Analytics Workspace
resource logAnalytics
'Microsoft.OperationalInsights/workspaces@2022-10-01' = {
  name: workspaceName
  location: location
  properties: {
    sku: {
```

```
    name: 'PerGB2018' // SKU estándar y de pago por uso
  }
  retentionInDays: 30 // Retenemos los logs por 30 días (el mínimo y más barato)
 }
}

// Exponemos el ID del recurso para que otros módulos lo puedan usar
output workspaceId string = logAnalytics.id
```

6. El orquestador principal (main.bicep)

Este archivo une todo.

Contenido en infra/main.bicep:

```
// PARÁMETROS: Entradas para nuestro despliegue.

param resourceNamePrefix string
param location string
param acrName string

// VARIABLES: Nombres construidos que usaremos en los recursos.

var vnetName = '${resourceNamePrefix}-vnet'
```

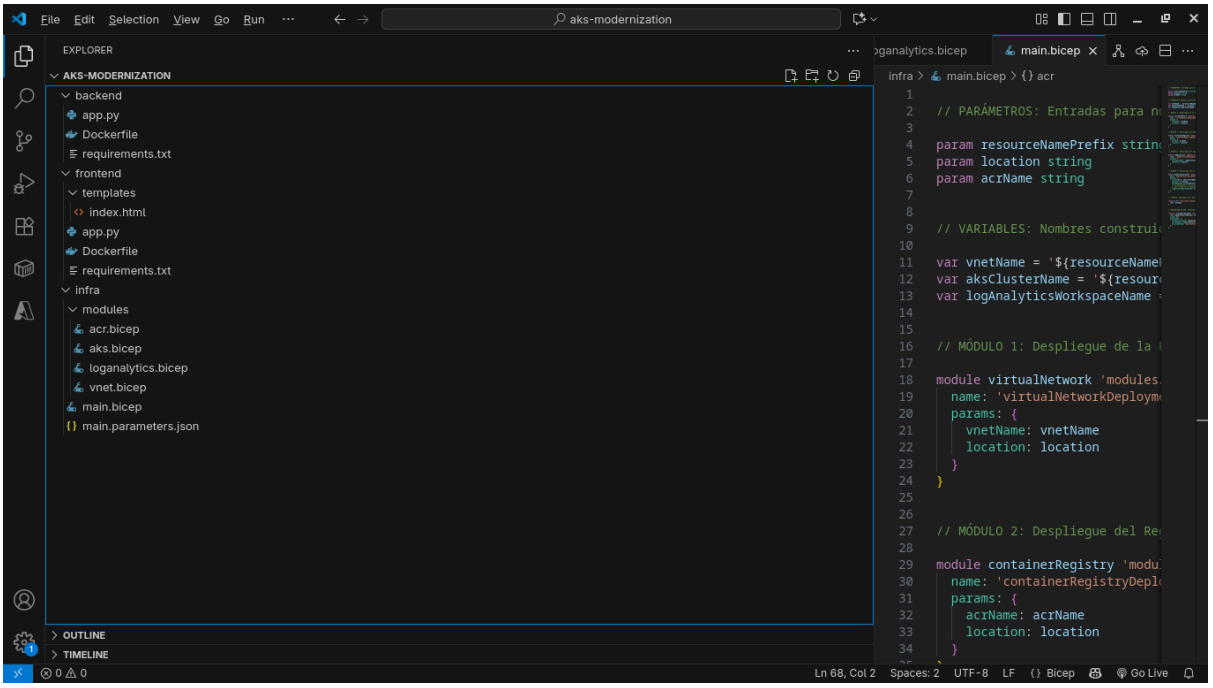
```
var aksClusterName = '${resourceNamePrefix}-aks'

var logAnalyticsWorkspaceName = '${resourceNamePrefix}-logs'



// MÓDULOS: Despliegue de los componentes de la infraestructura.


module virtualNetwork 'modules/vnet.bicep' = {
 name: 'virtualNetworkDeployment'
 params: {
   vnetName: vnetName
   location: location
 }
}


module containerRegistry 'modules/acr.bicep' = {
 name: 'containerRegistryDeployment'
 params: {
   acrName: acrName
   location: location
 }
}


module logAnalytics 'modules/loganalytics.bicep' = {
 name: 'logAnalyticsDeployment'
 params: {
```

```
    workspaceName: logAnalyticsWorkspaceName
    location: location
  }
}

module kubernetesCluster 'modules/aks.bicep' = {
  name: 'kubernetesClusterDeployment'
  params: {
    clusterName: aksClusterName
    location: location
    aksSubnetId: virtualNetwork.outputs.aksSubnetId
    logAnalyticsWorkspaceId: logAnalytics.outputs.workspaceId
  }
}



// RECURSO: Obtenemos una referencia al ACR para usarlo en el 'scope'.

resource acr 'Microsoft.ContainerRegistry/registries@2023-07-01'
existing = {
  name: acrName
}



// ASIGNACIÓN DE ROL: Conexión final entre AKS y ACR.
```

```
resource assignAcrPullRole
'Microsoft.Authorization/roleAssignments@2022-04-01' = {

 name: guid(resourceGroup().id, acrName, aksClusterName, 'AcrPull')

 scope: acr

 properties: {

   // Usando el ID del rol 'AcrPull' de la suscripción.

   roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions',
'7f951dda-4ed3-4680-a7ca-43fe172d538d')

   principalId: kubernetesCluster.outputs.aksPrincipalId

   principalType: 'ServicePrincipal'

 }

}
```

Se ha escrito todo el código IaC. **Revisa** que cada archivo esté en su lugar y tenga el contenido correcto, especialmente el nombre único de tu ACR en main.parameters.json.

Quedaría así:



Punto 2.3: Desplegar la infraestructura en Azure

Objetivo: Ejecutar un único comando que leerá nuestros archivos Bicep y construirá toda la arquitectura definida en nuestra suscripción de Azure.

Paso 1: Abrir la terminal en la ubicación correcta

1. Abre la terminal de Fedora.

Asegúrate de que estás en el directorio infra de tu proyecto. La ruta debería ser similar a ~/aks-modernization/infra. Si no estás ahí, navega a esa carpeta:

```
cd ~/aks-modernization/infra
```

Paso 2: Autenticarse y establecer la suscripción (Si es que es necesario)

Confirma que has iniciado sesión en Azure:

```
az account show
```

1. Si no te muestra la suscripción correcta, ejecuta az login de nuevo.
2. Si tienes varias suscripciones, asegúrate de que la que tiene el crédito gratuito está activa. Puedes listarlas con az account list y establecer la correcta con az account set --subscription "NOMBRE_O_ID_DE_LA_SUSCRIPCION".

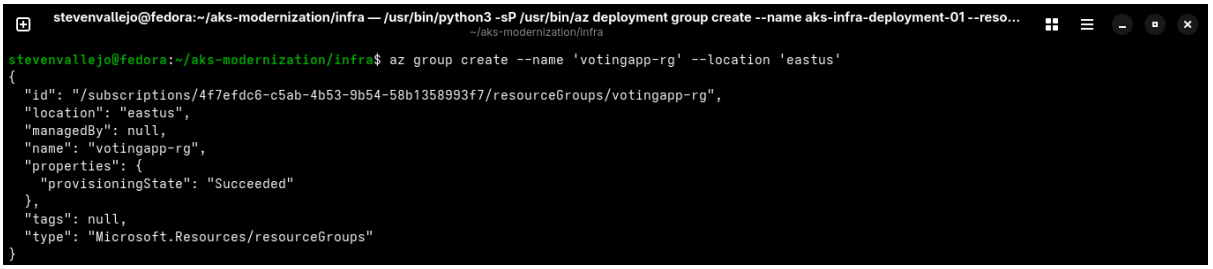
Paso 3: Ejecutar el comando de despliegue

1. Crear el Grupo de Recursos

Este comando creará el "contenedor" vacío en la región correcta. Debería completarse en pocos segundos.

```
az group create --name 'votingapp-rg' --location 'eastus'
```

Verás una salida JSON con "provisioningState": "Succeeded" confirmando que el grupo se ha creado.



Importante: Registrar los proveedores de Azure

1. Abre tu terminal.

   Ejecuta el siguiente comando para registrar el proveedor de servicios de Kubernetes. **Este comando puede tardar varios minutos (2 a 5 min) en completarse.** La terminal parecerá que no hace nada, pero está trabajando.

   ```
   az provider register --namespace Microsoft.ContainerService
   ```

2. Ahora, registra el proveedor para Log Analytics (Azure Monitor). Este también puede tardar unos minutos.

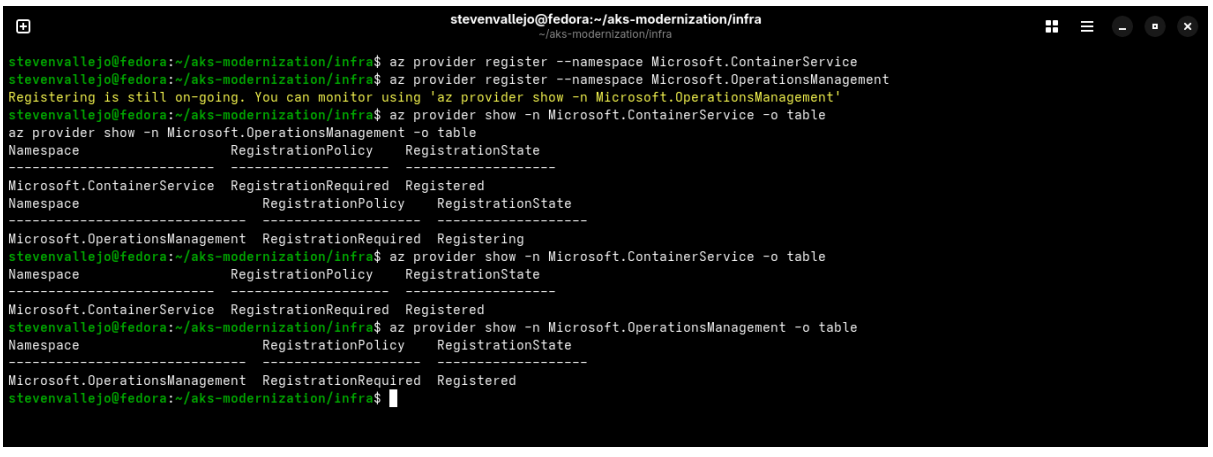
   ```
   az provider register --namespace Microsoft.OperationsManagement
   ```

3. **Verifica el estado del registro.** Puedes comprobar si ya han terminado con los siguientes comandos. Sigue ejecutándolos cada minuto hasta que el "registrationState" cambie de "Registering" a "Registered".

   ```
   az provider show -n Microsoft.ContainerService -o table
   ```

   ```
   az provider show -n Microsoft.OperationsManagement -o table
   ```

4. **No continúes** hasta que ambos comandos muestran "Registered" en la columna State**.**



2. Desplegar la infraestructura dentro del grupo

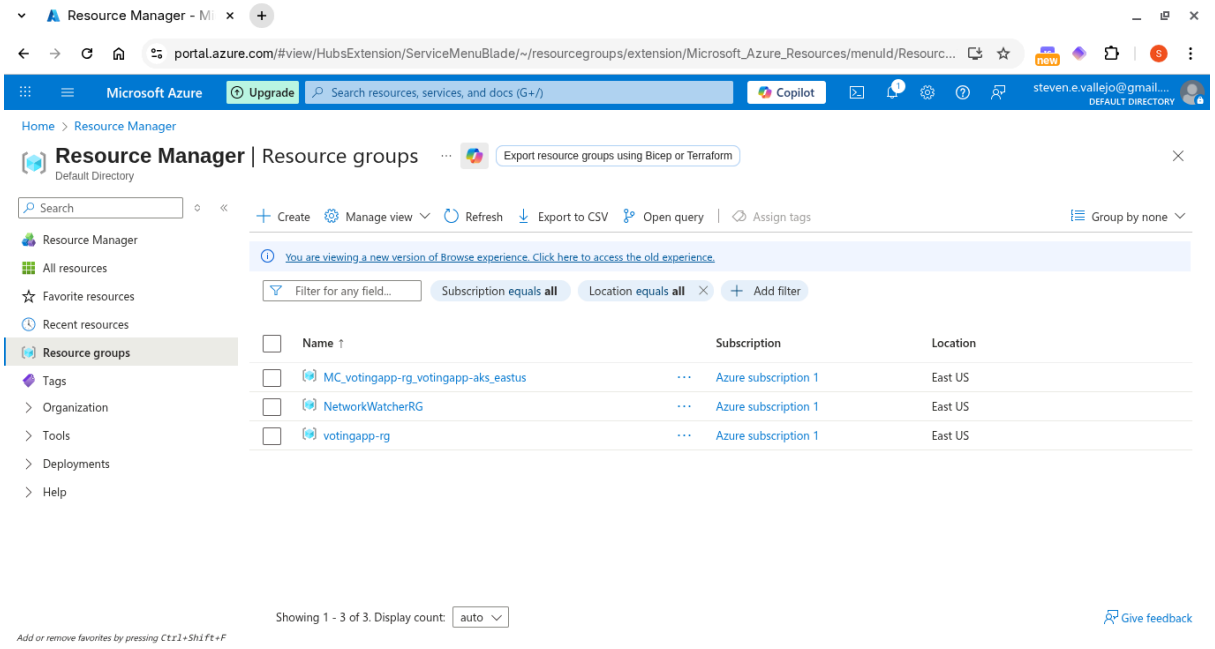
Ahora que el grupo de recursos existe, ejecutamos nuestro comando de despliegue

```
az deployment group create \
--name 'aks-infra-deployment-01' \
--resource-group 'votingapp-rg' \
--template-file 'main.bicep' \
--parameters 'main.parameters.json'
```

Empezará el proceso de 10-15 minutos para crear los recursos.

Paso 4: Observar y esperar

1. **No interrumpas el proceso.** La terminal mostrará un mensaje - Running ....
2. **Tiempo de ejecución:** Este proceso **tardará entre 10 y 15 minutos**.
3. **Opcional:**
    - Abre tu navegador y ve a https://portal.azure.com.
    - En la barra de búsqueda, escribe "Grupos de recursos".
    - Verás aparecer un nuevo grupo de recursos llamado votingapp-rg.
    - Si haces clic en él, verás cómo los recursos (VNet, ACR, AKS, Log Analytics) van apareciendo uno por uno a medida que el despliegue avanza.



El **despliegue** será **exitoso** cuando el comando en tu terminal termine y muestre una gran salida de texto en formato JSON. Lo más importante que debes buscar al final de esa salida es esta línea:

"provisioningState": "Succeeded",

Si es así, ha sido **exitoso.**

Resumen:

En el grupo de recursos votingapp-rg ahora mismo existen:

1. Una **Red Virtual** (votingapp-vnet) que aísla el clúster.
2. Un **Log Analytics Workspace** (votingapp-logs) para recolectar toda la telemetría.
3. Un **Azure Container Registry** (votingappacrsv2510) que servirá como almacén privado de imágenes de Docker.
4. Un **Clúster de Kubernetes** (votingapp-aks) con un nodo, conectado a la red, configurado para enviar logs.
5. **La conexión:** El permiso (roleAssignment) que permite al clúster AKS extraer imágenes del ACR de forma segura.

Fase 3: Despliegue de la aplicación en AKS

Objetivo: Publicar nuestras imágenes de contenedor en nuestro registro privado en la nube (ACR) y luego instruir a Kubernetes para que las descargue, las ejecute y las exponga a Internet.

Punto 3.1: Publicar las imágenes de Docker en Azure Container Registry (ACR)

Nuestras imágenes voting-app-frontend y voting-app-backend solo existen en nuestra máquina Fedora. Necesitamos subirlas a nuestro ACR para que el clúster de AKS pueda acceder a ellas.

Paso 1: Iniciar sesión en nuestro ACR desde la terminal

Primero, debemos autenticar nuestro Docker local con nuestro registro en la nube.

1. Abre tu terminal.

Necesitamos el nombre de nuestro ACR. **Reemplaza <TU_ACR_UNICO> por el nombre que le diste** (ej. votingappacrsv2510). O usa este comando para obtenerlo automáticamente:

```
ACR_NAME=$(az acr list --resource-group votingapp-rg --query "[0].name" --output tsv)

echo "Iniciando sesión en: $ACR_NAME"

az acr login --name $ACR_NAME
```

2. El comando az acr login debería devolver Login Succeeded. Esto configura tu Docker local para que pueda "empujar" (push) imágenes a tu registro.

Paso 2: "Etiquetar" (Tag) las imágenes para el registro

Docker necesita saber a qué registro debe subir cada imagen. Para ello, creamos una nueva "etiqueta" (un alias) para nuestras imágenes locales con el formato: <nombre_del_acr>.azurecr.io/<nombre_de_la_imagen>:<version>.

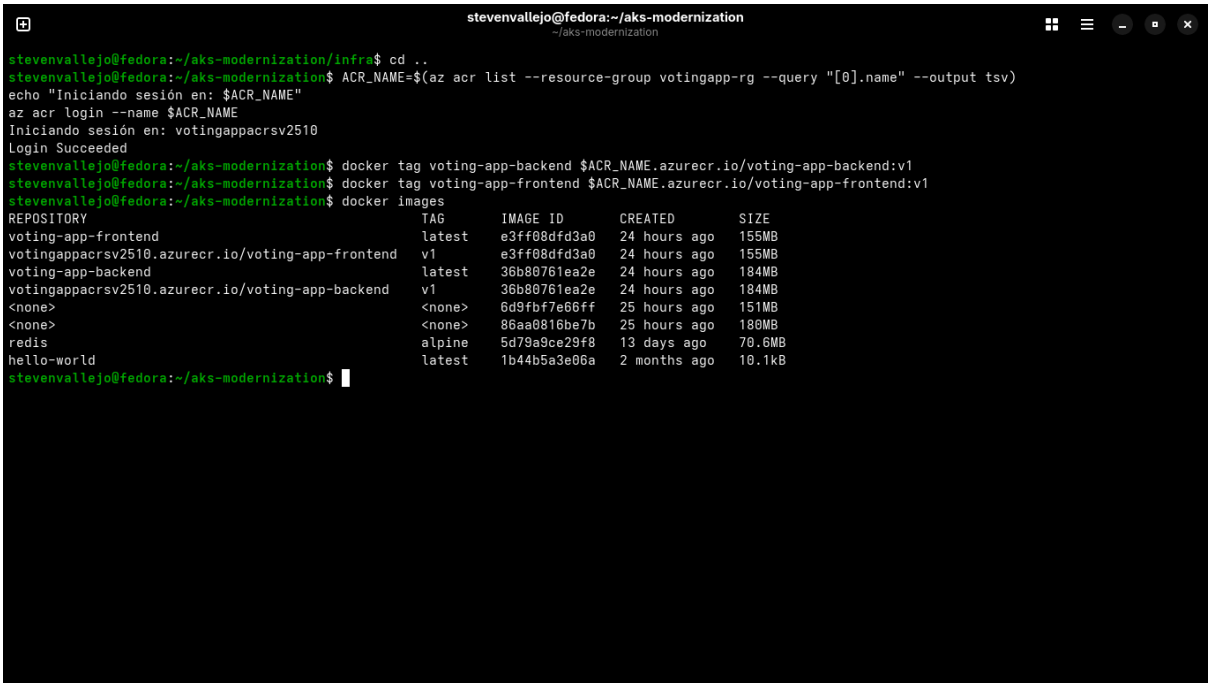
1. **Etiquetar la imagen del backend:**

   docker tag voting-app-backend $ACR_NAME.azurecr.io/voting-app-backend:v1

2. **Etiquetar la imagen del frontend:**

   docker tag voting-app-frontend $ACR_NAME.azurecr.io/voting-app-frontend:v1

3. **Verificar las nuevas etiquetas:** Ejecuta **docker images**. Ahora verás las imágenes originales y sus nuevos alias, que apuntan al mismo IMAGE ID.



Paso 3: Empujar (Push) las imágenes a la nube

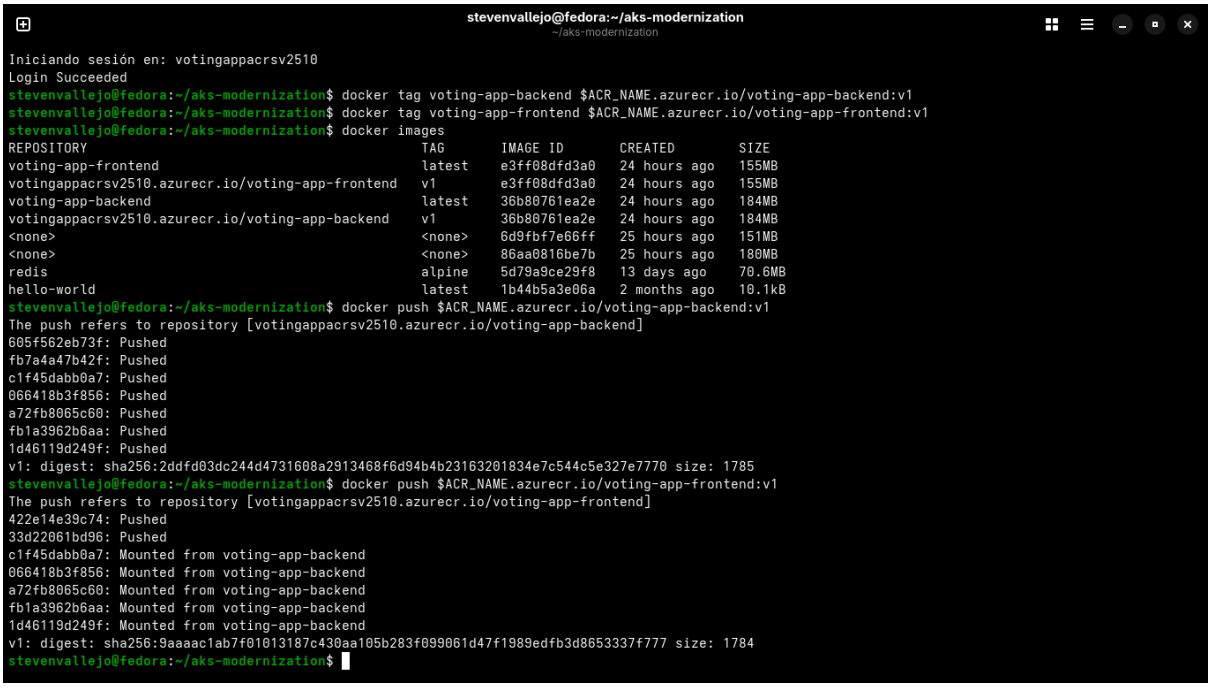
Es el equivalente a subir un archivo, pero para imágenes de Docker.

1. **Subir la imagen del backend:**

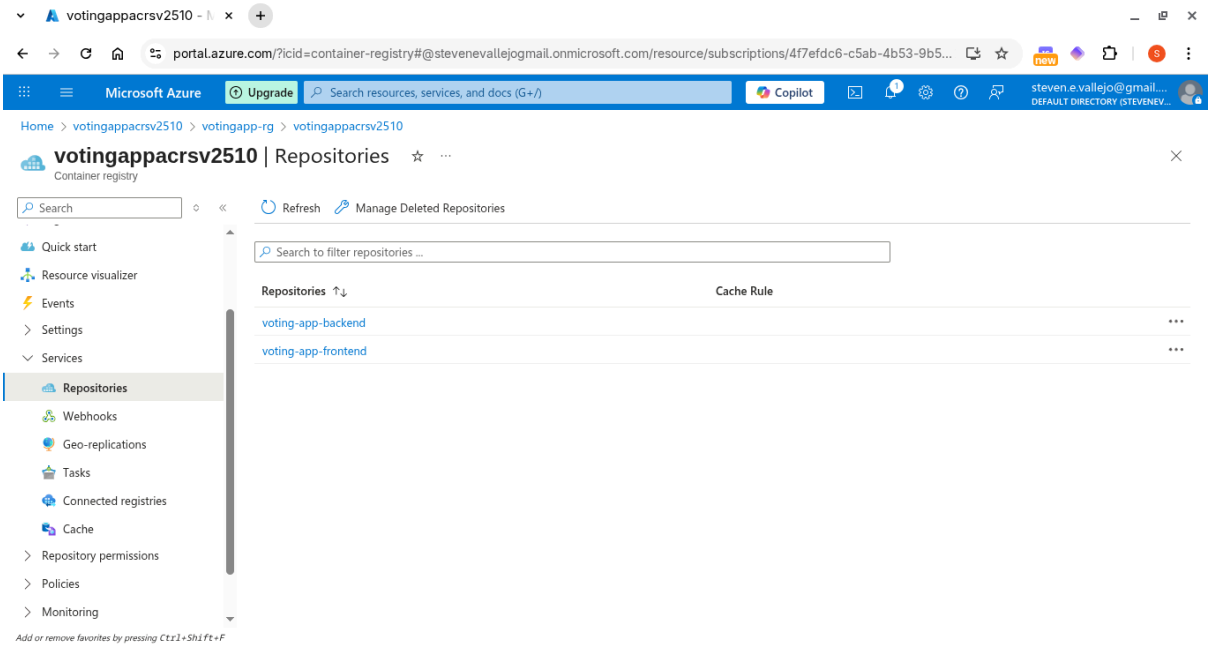
   docker push $ACR_NAME.azurecr.io/voting-app-backend:v1

   Verás una barra de progreso mientras las capas de la imagen se suben a tu ACR.
2. **Subir la imagen del frontend:**

   docker push $ACR_NAME.azurecr.io/voting-app-frontend:v1



Para **verificar** que las **imágenes** se han subido correctamente, ve al **Portal de Azure**, busca tu **ACR**, ir a la sección "**Repositorios**" y verás voting-app-backend y voting-app-frontend listados.



Punto 3.2: Escribir los manifiestos de Kubernetes

Objetivo: Crear los archivos de configuración (.yaml) que le dicen a Kubernetes qué contenedores ejecutar, cómo conectarlos entre sí y cómo exponer la aplicación al mundo exterior.

Paso 1: Crear la carpeta para los manifiestos

Para mantener nuestro proyecto organizado, crearemos una nueva carpeta para estos archivos.

1. En tu terminal, asegúrate de estar en la raíz del proyecto (~/aks-modernization).

   mkdir manifests

   cd manifests

Paso 2: Crear el manifiesto de despliegue (deployment.yaml)

Este archivo contendrá las instrucciones para los tres componentes de nuestra aplicación: **Redis, Backend y Frontend**.

1. Dentro de la carpeta manifests, crea un archivo llamado deployment.yaml.
   Contenido completo en el archivo.

```
# manifests/deployment.yaml
```

```yaml
# Definición del Deployment para REDIS
# Un Deployment se encarga de mantener un número deseado de
copias (pods) de nuestra aplicación funcionando.
apiVersion: apps/v1
kind: Deployment
metadata:
 name: redis-deployment # Nombre del deployment
spec:
 replicas: 1 # Una sola copia de Redis
 selector:
   matchLabels:
```

```yaml
      app: redis # Este selector conecta el Deployment con los Pods que gestiona
  template:
    metadata:
      labels:
        app: redis # Los Pods creados tendrán esta etiqueta
    spec:
      containers:
      - name: redis
        image: redis:alpine # La imagen de Docker a usar (de Docker Hub)
        ports:
        - containerPort: 6379 # El puerto que expone el contenedor de Redis

---

# Definición del Service para REDIS
# Un Service proporciona una dirección de red estable (un nombre DNS interno) para acceder a los Pods.
apiVersion: v1
kind: Service
metadata:
  name: redis-service # Nombre del service. El backend usará este nombre para conectarse.
spec:
  ports:
  - port: 6379 # El puerto que expone el Service
  selector:
    app: redis # Redirige el tráfico a cualquier Pod con la etiqueta app=redis

---

# Deployment para el BACKEND
apiVersion: apps/v1
kind: Deployment
```

```yaml
metadata:
 name: backend-deployment
spec:
 replicas: 1 # Empezamos con una copia
 selector:
   matchLabels:
     app: backend
 template:
   metadata:
     labels:
       app: backend
   spec:
     containers:
     - name: backend
       image: votingappacrsv2510.azurecr.io/voting-app-backend:v1
       ports:
       - containerPort: 80
       env:
       - name: REDIS_HOST
         value: "redis-service" # Le decimos al backend que Redis se encuentra en 'redis-service'

---

# Definición del Service para el BACKEND
apiVersion: v1
kind: Service
metadata:
 name: backend-service
spec:
 ports:
 - port: 80
 selector:
   app: backend

---
```

```yaml

# Deployment para el FRONTEND
apiVersion: apps/v1
kind: Deployment
metadata:
 name: frontend-deployment
spec:
 replicas: 1
 selector:
   matchLabels:
     app: frontend
 template:
   metadata:
     labels:
       app: frontend
   spec:
     containers:
     - name: frontend
       image:
votingappacrsv2510.azurecr.io/voting-app-frontend:v1
       ports:
       - containerPort: 8080
          env:
          - name: BACKEND_API_URL
            value: "http://backend-service" # Le decimos al
frontend que la API está en 'http://backend-service'

   ---

   # Service para el FRONTEND
   # Se expone nuestra aplicación a Internet.
   apiVersion: v1
   kind: Service
   metadata:
    name: frontend-service
   spec:
```

```
    # 'LoadBalancer' es el tipo de Service que automáticamente
crea un balanceador de carga
    # en Azure con una dirección IP pública para recibir tráfico
externo.
    type: LoadBalancer
    ports:
    - port: 80 # El Load Balancer recibirá tráfico en el puerto 80
(HTTP estándar)
      targetPort: 8080 # Y lo redirigirá al puerto 8080 de
nuestros contenedores frontend
    selector:
      app: frontend # Redirige el tráfico a cualquier Pod con la
etiqueta app=frontend

```

Se ha creado un **archivo deployment.yaml** completo que **describe** toda tu **aplicación** en el lenguaje de Kubernetes y lo has **personalizado** con la ubicación de tus imágenes en tu **ACR**.

Punto 3.3: Conectarse al clúster y desplegar la aplicación

Objetivo: Establecer una conexión segura desde la terminal de Fedora a tu clúster de AKS en Azure y aplicar nuestro manifiesto para que Kubernetes construya y lance la aplicación.

Paso 1: Obtener las credenciales de acceso al clúster

Necesitamos descargar un archivo de configuración especial (kubeconfig) que contiene las credenciales y la dirección del clúster.

1. Abre tu terminal.

   Ejecuta el siguiente comando. Reemplaza votingapp-rg y votingapp-aks si usaste nombres diferentes.

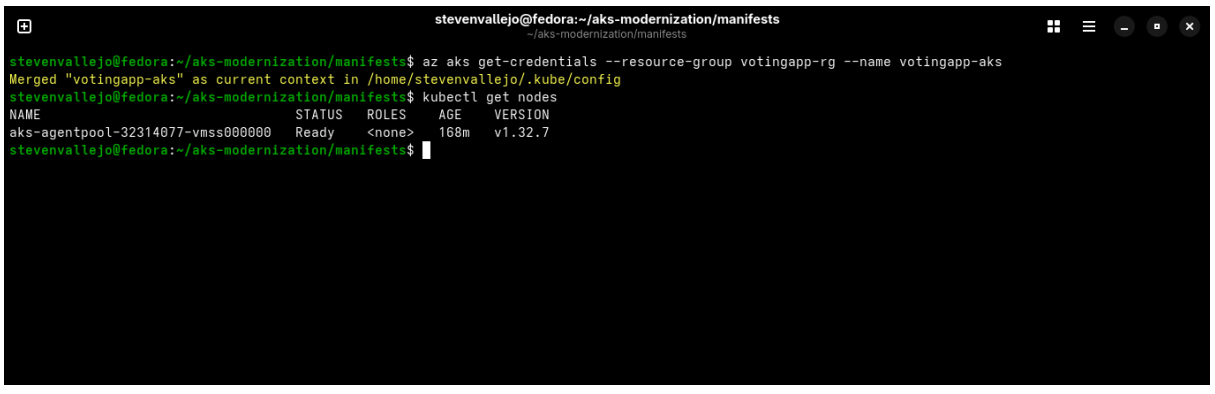
   ```
   az aks get-credentials --resource-group votingapp-rg --name votingapp-aks
   ```

2. Este comando contactará a Azure, descargará las credenciales y las fusionará automáticamente con tu configuración de kubectl local en el archivo ~/.kube/config. Deberías ver un mensaje como: Merged "votingapp-aks" as current context in /home/stevenvallejo/.kube/config.

Para verificar que kubectl ahora puede comunicarse con tu clúster, ejecuta este comando:

```
kubectl get nodes
```

Deberías ver **un nodo** en la lista, con el estado Ready.



Paso 2: Desplegar la aplicación usando el manifiesto

Ahora que tenemos una conexión, podemos enviarle instrucciones.

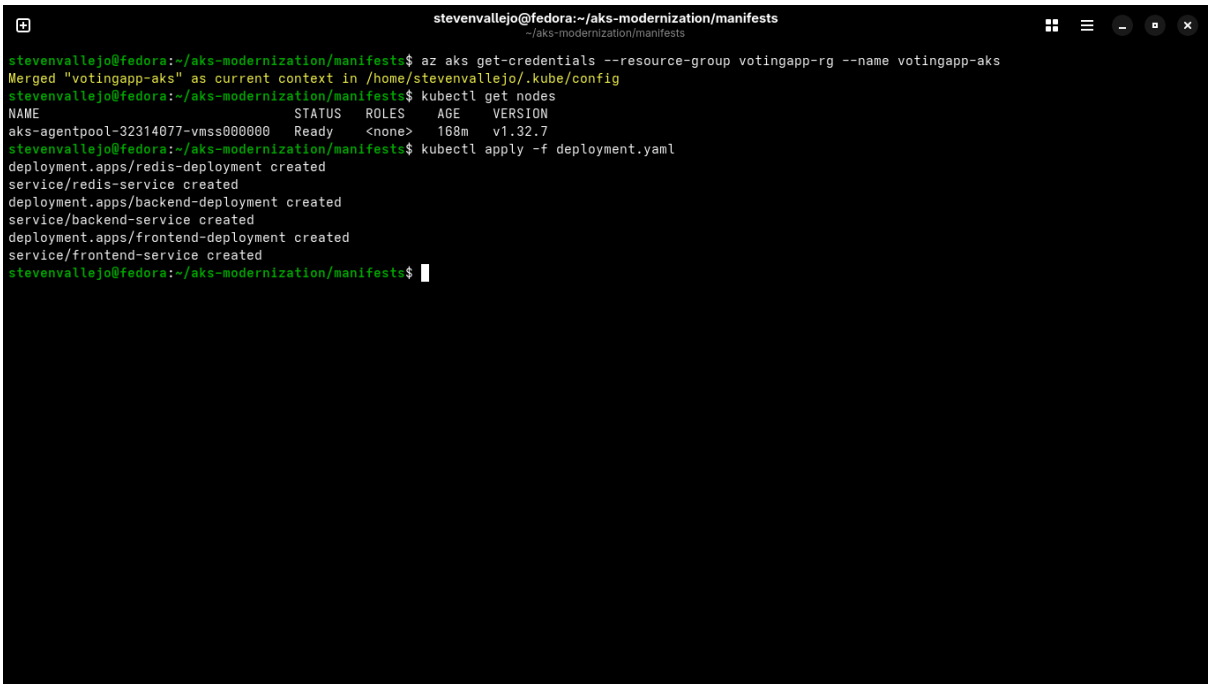
1. Navega a la carpeta donde guardaste tu archivo de manifiesto:

   cd ~/aks-modernization/manifests

2. Ejecuta el comando apply de kubectl. Este comando lee el archivo y le dice a Kubernetes que coincida con lo que está en el manifiesto.

   kubectl apply -f deployment.yaml

kubectl te **confirmará** que has **creado** todos los **recursos** definidos en el archivo.
Saldrá así:



Paso 3: Verificar el despliegue

Le hemos dado las órdenes a Kubernetes. Ahora, está trabajando para hacerlas realidad. Kubernetes tiene que:

1. Programar los "Pods" (contenedores) para que se ejecuten en nuestro nodo.
2. El nodo tiene que descargar las imágenes de tu ACR (esto puede tardar 1-2 minutos).
3. Iniciar los contenedores.
4. Azure tiene que aprovisionar un balanceador de carga y asignarle una IP pública (esto puede tardar otros 2-3 minutos).

Usa los siguientes comandos para monitorizar el proceso..

1. **Ver el estado de los Pods:**

   ```
   kubectl get pods
   ```

   Al principio, puede que veas el estado ContainerCreating. Después de un minuto o dos, todos deberían pasar al estado Running.

   Si backend y frontend no están corriendo, se debe a que Azure intentó descargarlos pero falló o no tiene permiso y los puso en espera. Para solucionar esto ejecutaremos el siguiente código:

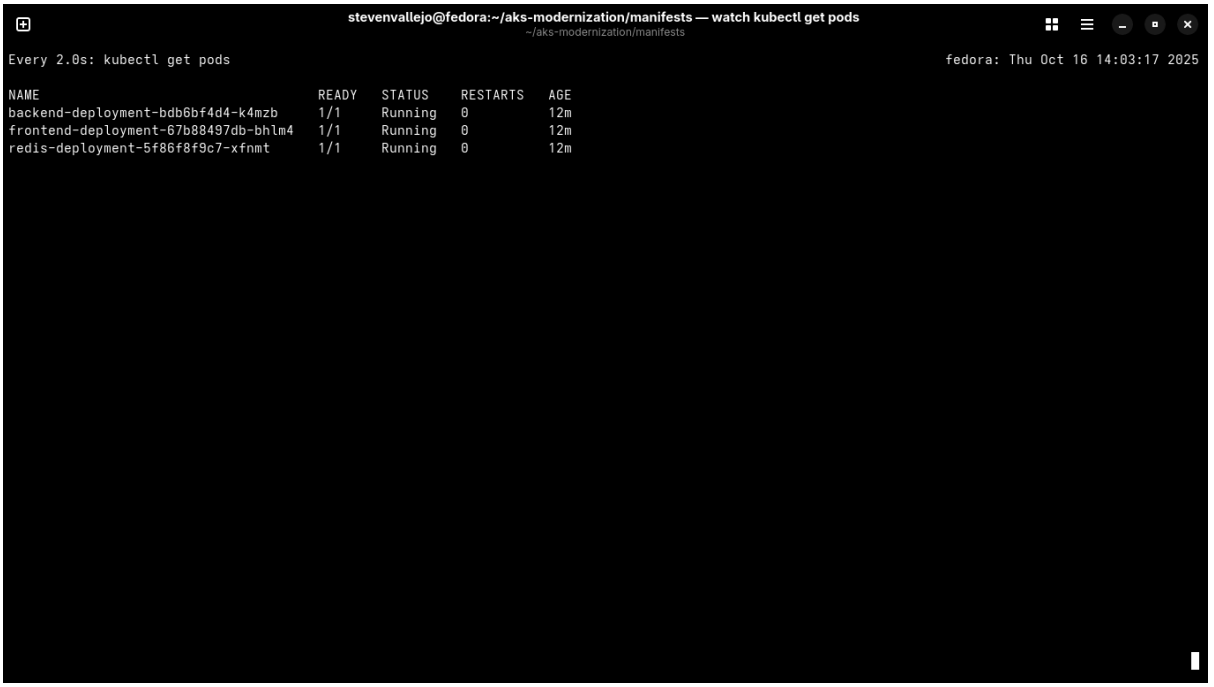
   Primero, obtenemos el nombre del ACR para no tener que escribirlo a mano

   ```
   ACR_NAME=$(az acr list --resource-group votingapp-rg --query "[0].name" --output tsv)
   ```

   Este es el comando que **arregla la conexión**

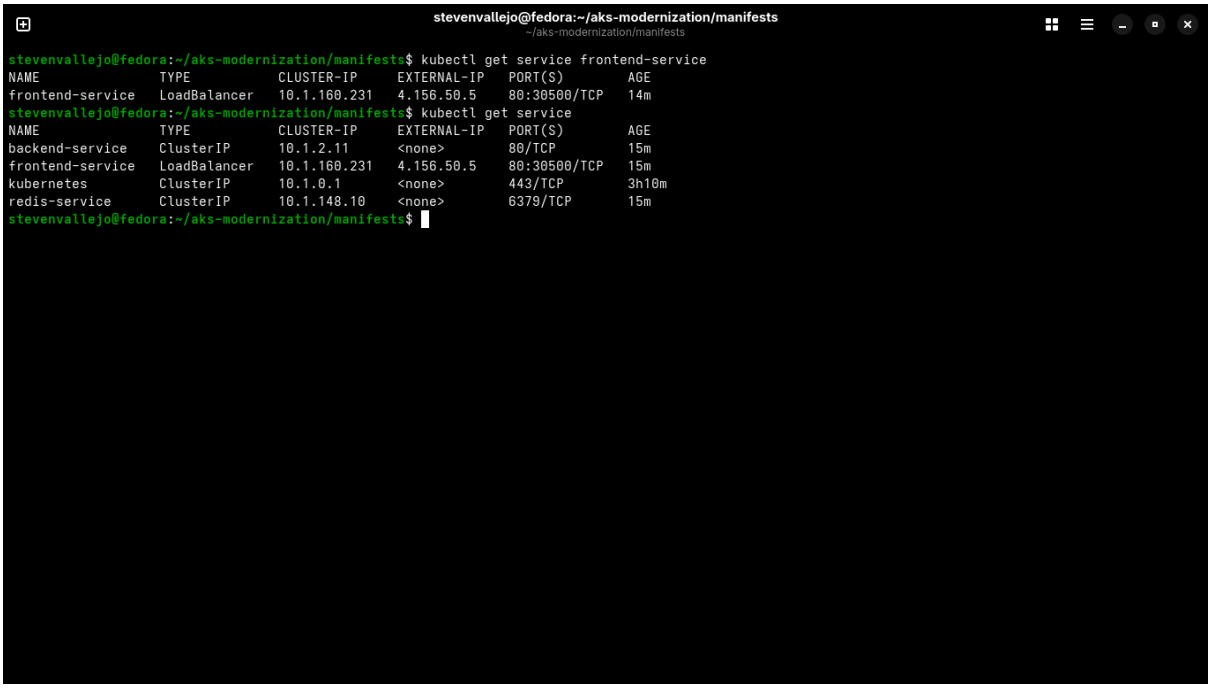
   ```
   az aks update --name votingapp-aks --resource-group votingapp-rg --attach-acr $ACR_NAME
   ```

Verificamos y nos saldrá esto:

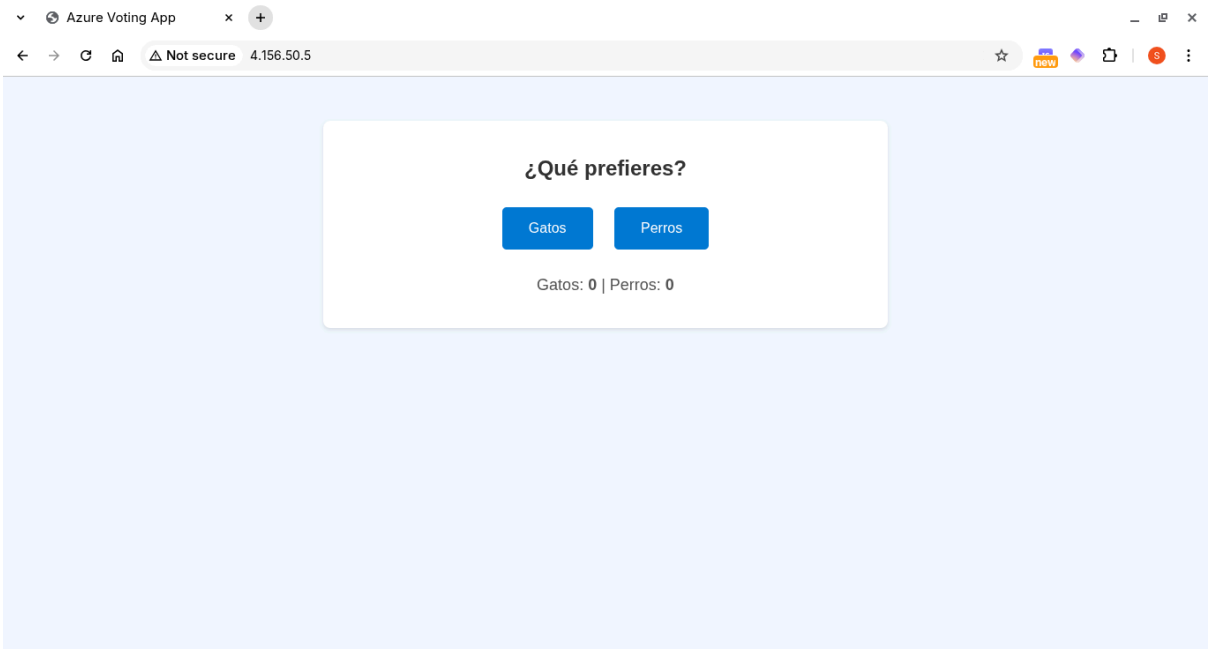


2. **Ver el estado de los services y obtener la IP Pública:**

kubectl get service



Y ahora con la **IP pública**, **pegamos** en el **navegador** y saldrá la aplicación web



Si deseas **apagarlo** antes de pasar a la siguiente fase para **evitar consumos extras**, utiliza el siguiente código en la terminal:

```
az aks stop --name votingapp-aks --resource-group votingapp-rg
```

Fase 4: Validación, Monitoreo y Resiliencia

Objetivo: Comprender cómo observar el estado de nuestra aplicación, demostrar la capacidad de auto-reparación de Kubernetes y ver cómo escalar la aplicación para manejar más carga.

Punto 4.1: Reanudar el clúster de Kubernetes

1. Abre tu terminal.
2. Ejecuta el comando start. Este proceso tardará entre 2 y 4 minutos.

   ```
   az aks start --name votingapp-aks --resource-group votingapp-rg
   ```
3. Una vez que el comando termine, espera un minuto adicional para que todos los componentes internos se estabilice.

Revisa que los pods vuelvan al estado Running y que el frontend-service obtenga una nueva IP pública (o la misma de antes, puede variar).

```
kubectl get pods

kubectl get service frontend-service
```

4. Copia la nueva EXTERNAL-IP y pégala en tu navegador para confirmar que la aplicación está de nuevo en línea.

Punto 4.2: Demostración de alta disponibilidad (Auto-Reparación)

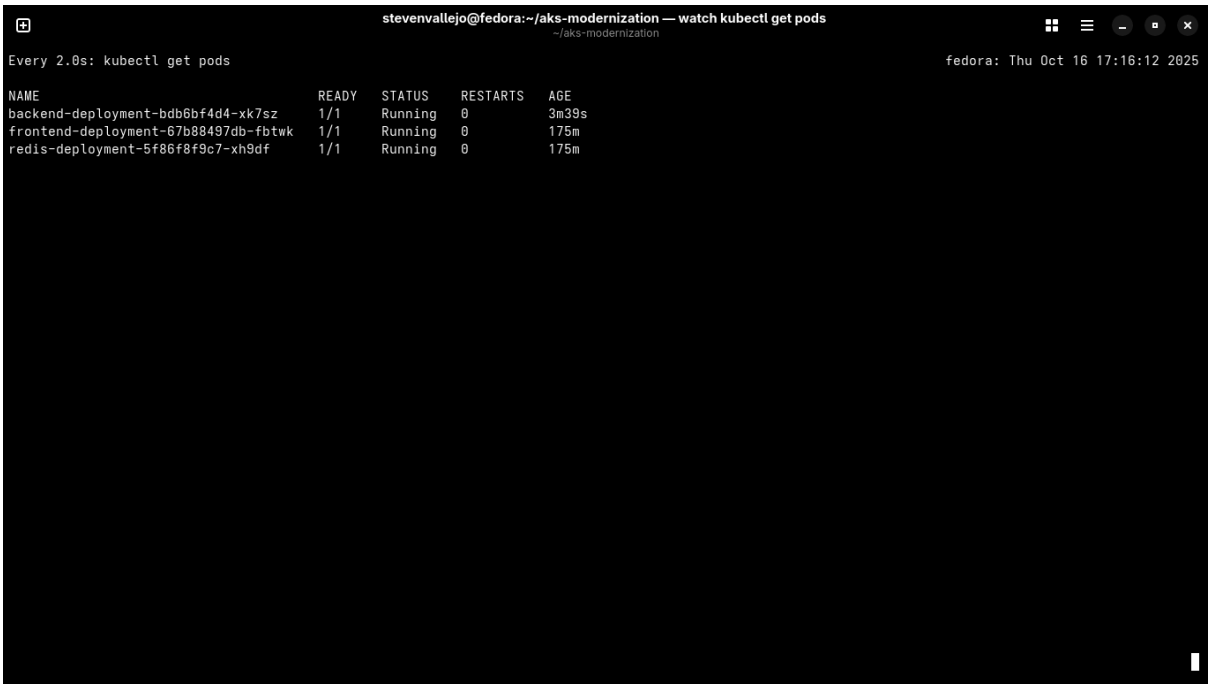
Esta es una de las características más poderosas de Kubernetes. Vamos a simular una falla catastrófica en nuestro backend y el sistema se recupera solo.

1. **Abre dos terminales.**

   En la Terminal 1: Configura un "observador" para los pods.

   ```
   watch kubectl get pods
   ```

   Verás la lista de tus los pods en estado Running.



En la Terminal 2: Identifica el nombre exacto de tu pod de backend. Parecido algo como backend-deployment-bdb6bf4d4-k4mzb.

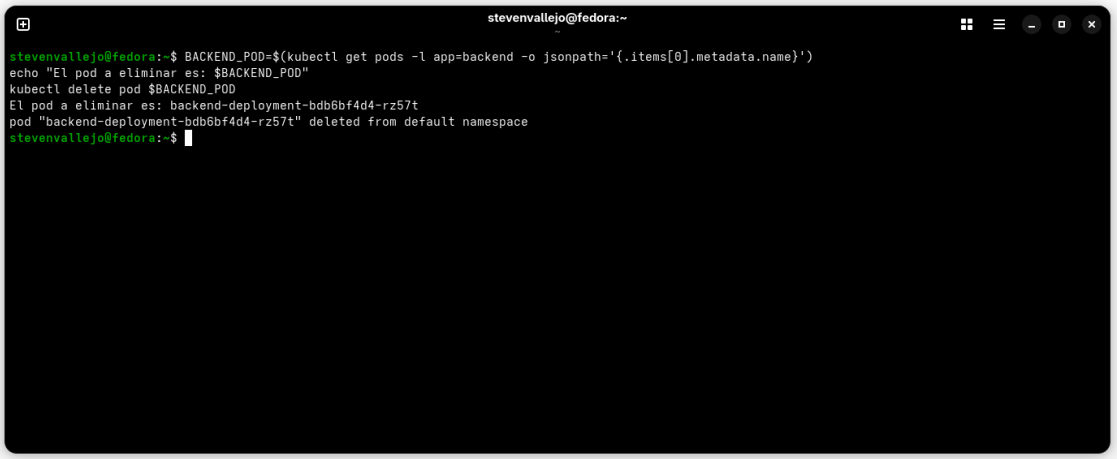
Primero obtén el **nombre exacto** del pod

```
BACKEND_POD=$(kubectl get pods -l app=backend -o jsonpath='{.items[0].metadata.name}')
```

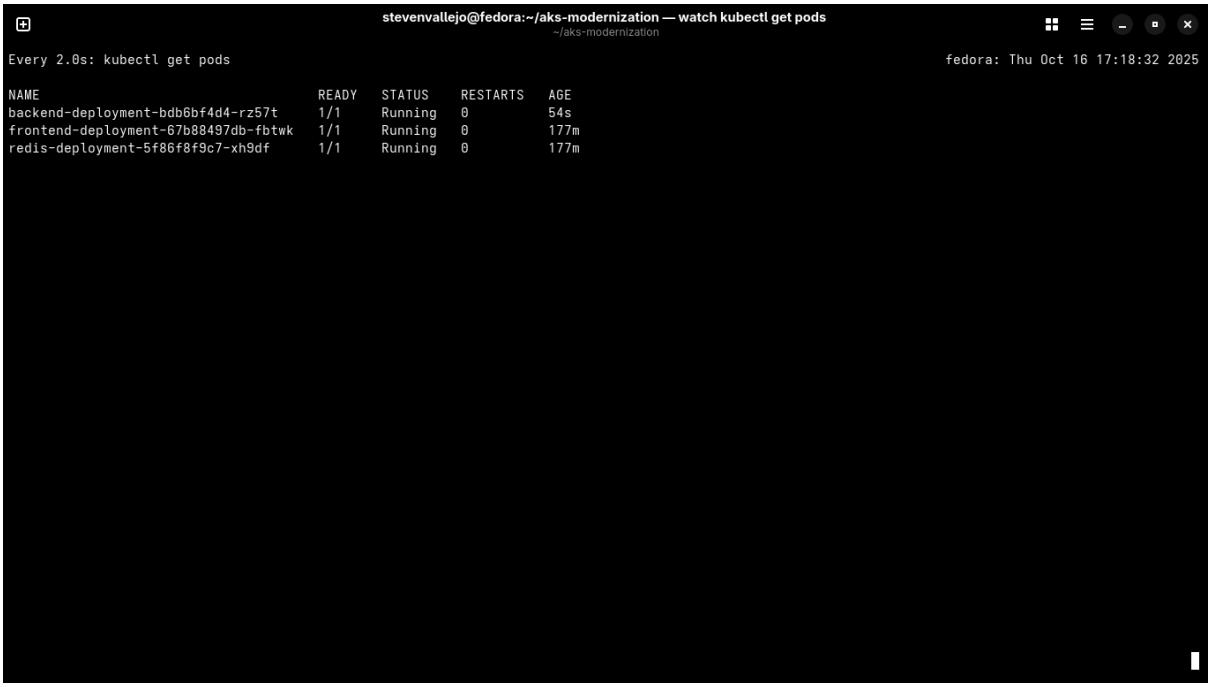
```
echo "El pod a eliminar es: $BACKEND_POD"
```

Ahora se **simula** la falla

```
kubectl delete pod $BACKEND_POD
```



2. **Observa la Terminal 1 (la del watch)**
    - El pod backend-... cambiará inmediatamente al estado Terminating.
    - Casi al mismo tiempo, aparecerá un **nuevo** pod backend-... con un nombre diferente, y su estado será Pending, luego ContainerCreating, y finalmente Running.



Conclusión: El Deployment de Kubernetes detectó que el número de réplicas deseadas (1) no se cumplía y automáticamente creó un nuevo pod para reemplazar al que falló. **Esto es la auto-reparación en acción.** La aplicación probablemente no tuvo ninguna interrupción perceptible para el usuario.

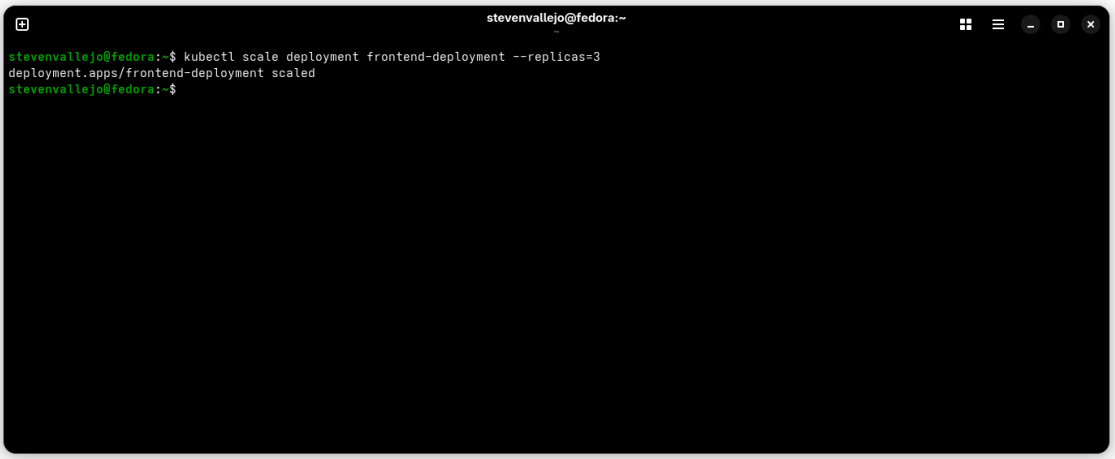
Punto 4.3: Demostrar la escalabilidad (Escalado Horizontal)

Digamos que la aplicación de votación se vuelve viral. Necesitamos más capacidad para manejar el tráfico.

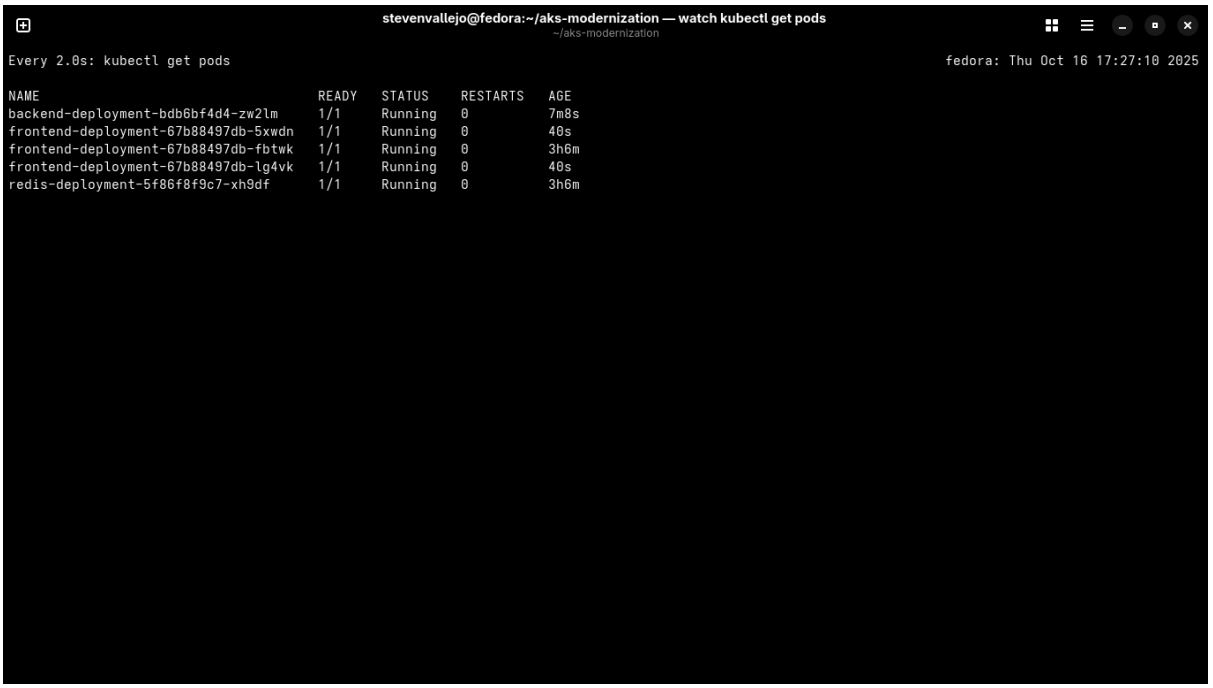
1. **Mantén la Terminal 1 con watch kubectl get pods**.

   En la Terminal 2: Ejecuta el comando scale para decirle a Kubernetes que ahora quieres **tres** copias (réplicas) de tu frontend.

   ```
   kubectl scale deployment frontend-deployment --replicas=3
   ```



2. **Observa la Terminal 1:** Inmediatamente, verás aparecer **dos nuevos pods** del frontend (frontend-deployment-...). Pasarán por los estados Pending, ContainerCreating y Running.



En menos de un minuto, tendrás tres pods del frontend funcionando. El Service de tipo LoadBalancer que creamos automáticamente distribuirá el tráfico de los usuarios entre estos tres pods, triplicando la capacidad de tu aplicación.

Para volver a la normalidad y ahorrar recursos, ejecuta:

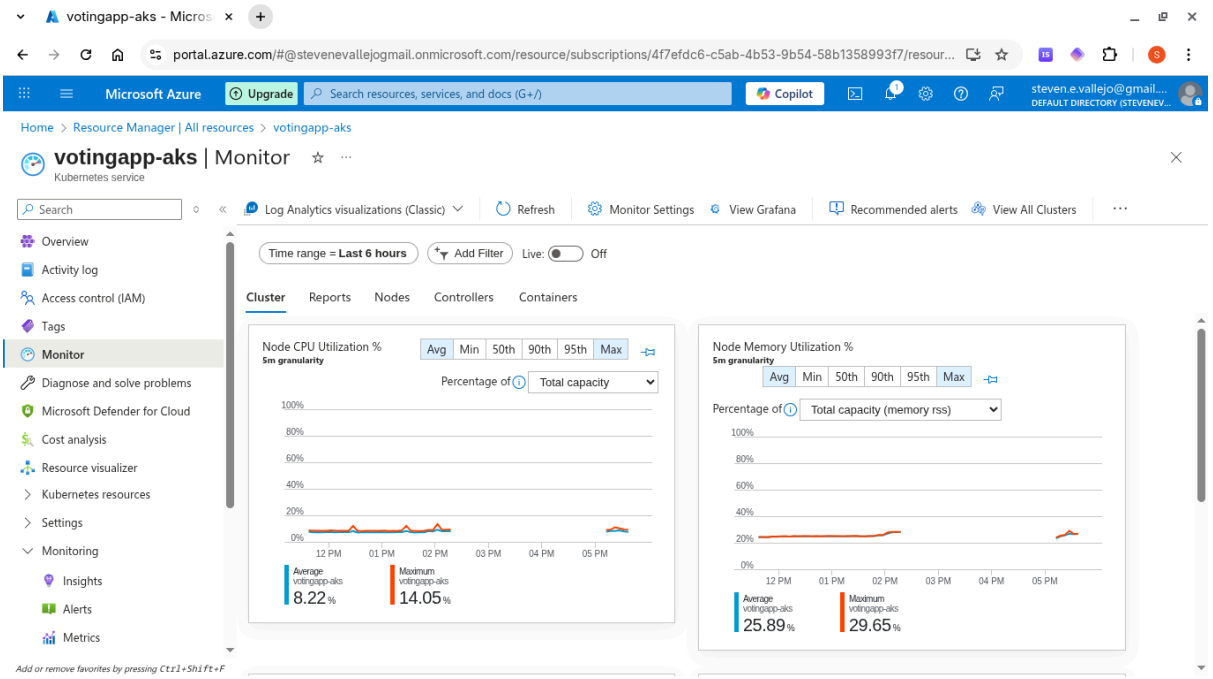
```
kubectl scale deployment frontend-deployment --replicas=1
```

Verás cómo los dos pods extra pasan al estado Terminating hasta que solo quede uno.

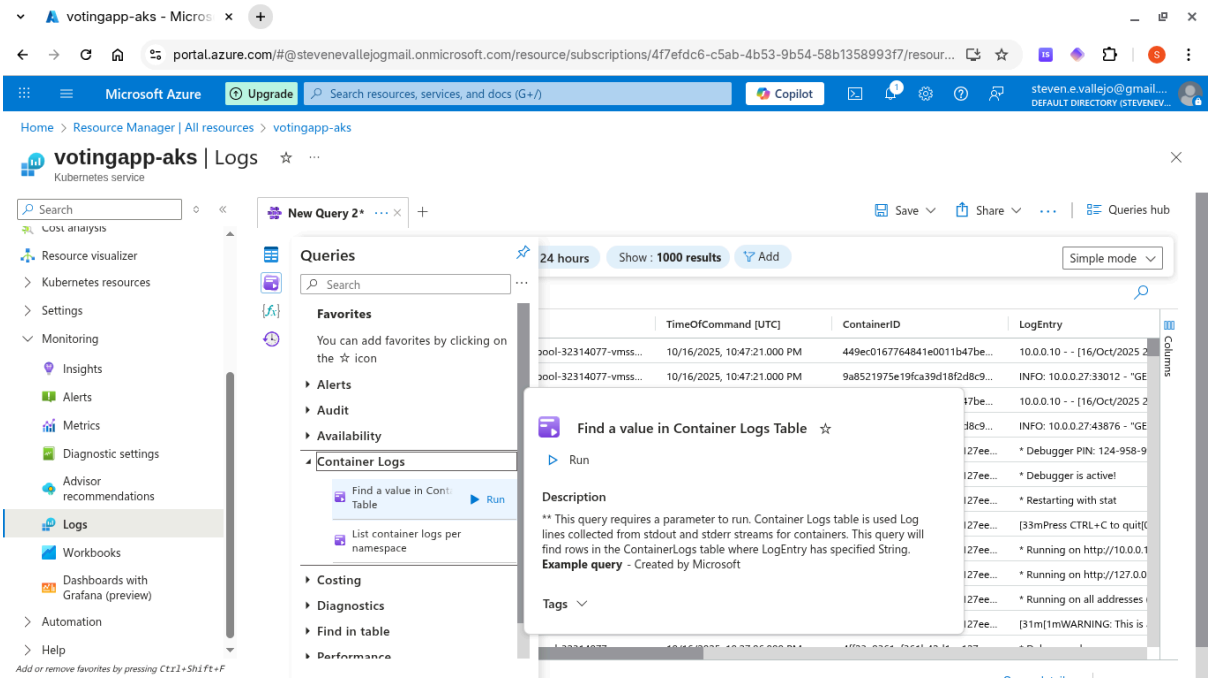
Punto 4.4: Explorar Azure Monitor for Containers (Observabilidad)

Ahora veamos el "diario de operaciones" que construimos en la Fase 2.

1. Ve al **Portal de Azure**.
2. Busca el clúster de Kubernetes, **votingapp-aks**.
3. En el menú de la izquierda, en la sección **"Monitoring"** (Monitoreo), haz clic en **"Insights"** (Información).
4. **Dale un minuto para que cargue los datos.** Verás un dashboard:
    - **Pestaña "Cluster":** Verás el uso promedio de CPU y memoria de tus nodos.
    - **Pestaña "Nodes":** Un desglose del rendimiento por cada nodo (en nuestro caso, solo uno).
    - **Pestaña "Controllers" y "Containers":** Aquí puedes profundizar y ver el uso de CPU y memoria de cada uno de tus pods (frontend, backend, redis).



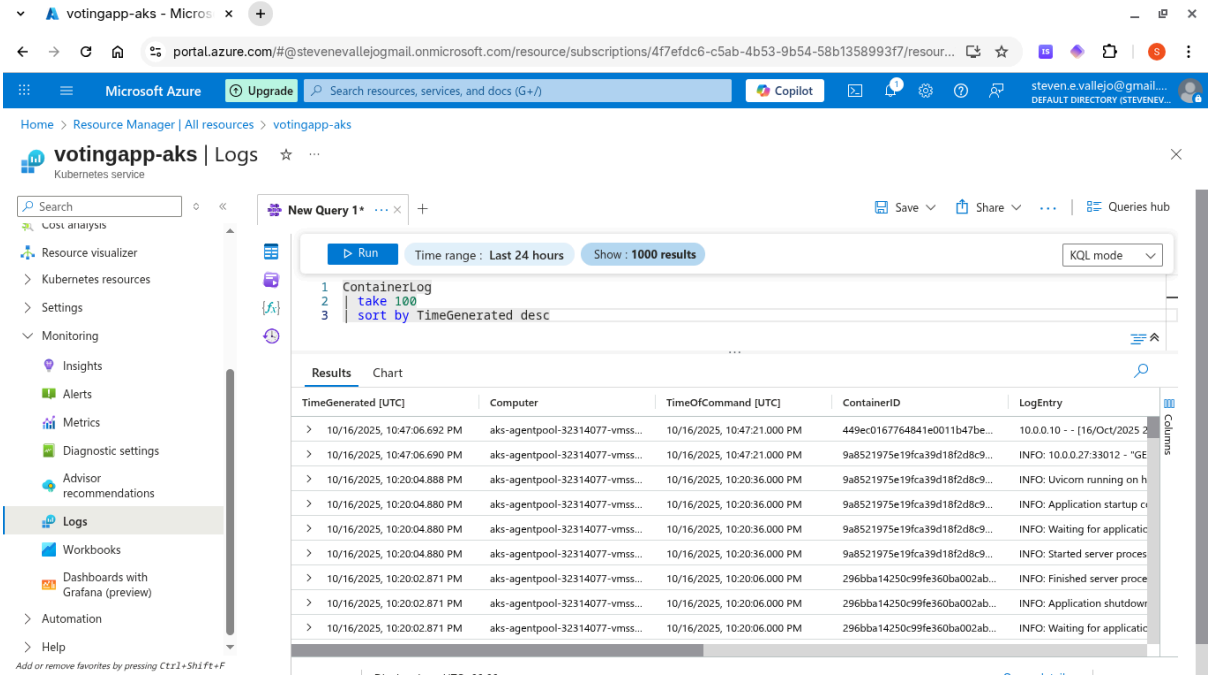
5. **Para ver los logs:**
    - Haz clic en "View Logs" o busca la pestaña "Logs".
    - Se abrirá una ventana de consulta. En el panel izquierdo, puedes navegar por las tablas. La más importante es ContainerLog.



Pega la siguiente consulta en la ventana y haz clic en "Run". Esto te mostrará los últimos 100 logs de tus contenedores:

```
ContainerLog

| take 100

| sort by TimeGenerated desc
```

    - Puedes ver los print() que pusimos en el código de Python, lo que es invaluable para la depuración.



Ejecuta el siguiente comando para **eliminar** todos los **recursos** que hemos utilizado para este proyecto para evitar el pago innecesario.

```
az group delete --name votingapp-rg --yes --no-wait
```

- --name votingapp-rg: Especifica el grupo de recursos que contiene todo nuestro trabajo.
- --yes: Confirma la eliminación sin pedirte una segunda confirmación.
- --no-wait: Devuelve el control de la terminal inmediatamente y la eliminación se ejecuta en segundo plano.

ADVERTENCIA: Asegúrate que se haya eliminado visualizando en el Portal de Azure, de no ser así, **elimina manualmente**.

Conclusión del proyecto.-

Se modernizó exitosamente una aplicación a una arquitectura de microservicios en Azure Kubernetes Service (AKS). Usando Bicep (IaC), se desplegó toda la infraestructura de forma automatizada, creando una solución funcional, escalable y con capacidad de auto-reparación. El proyecto demostró los principios clave de la nube nativa y se completó con un costo operativo mínimo, validando la eficiencia de la plataforma.