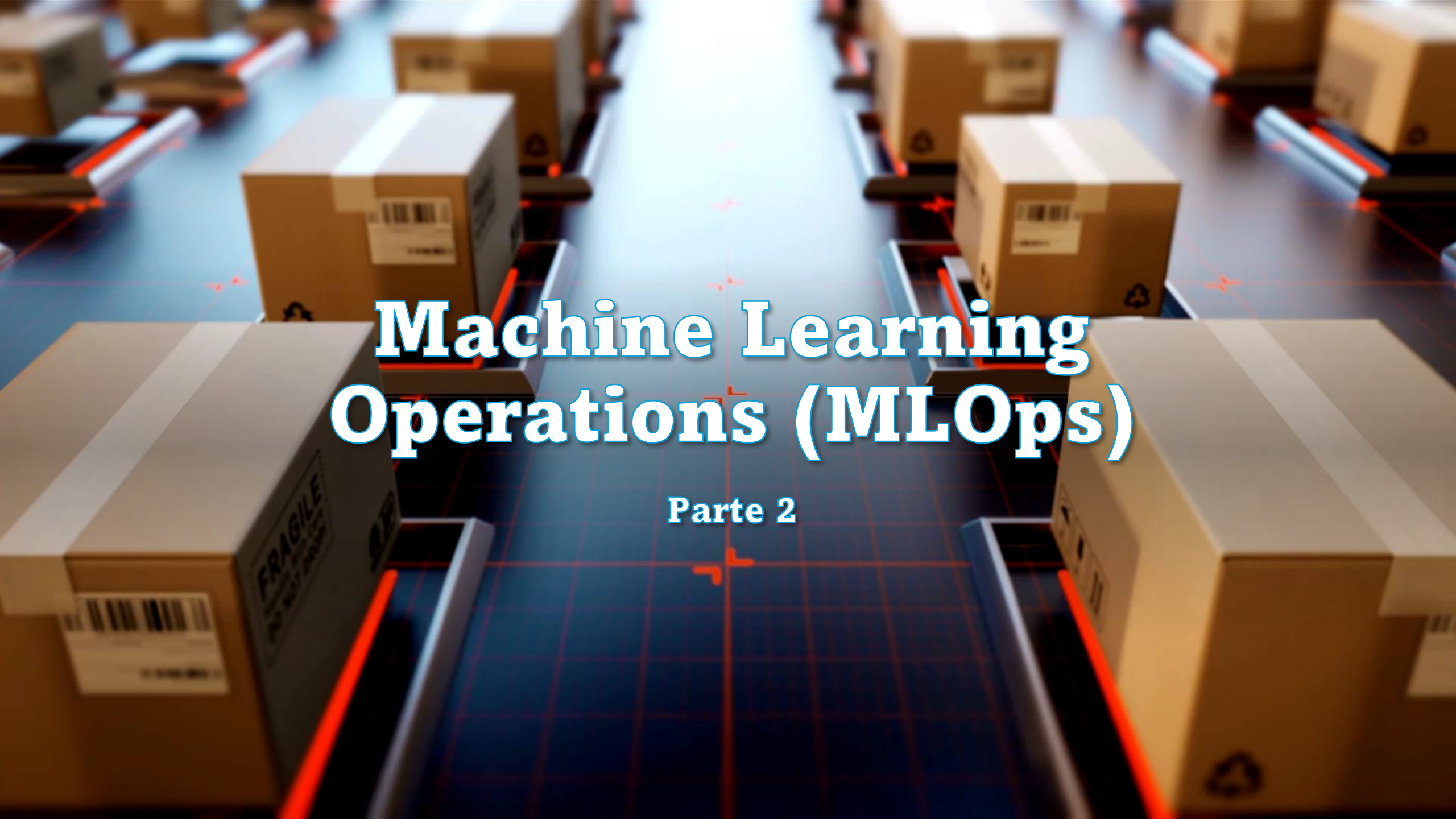
Machine Learning & Big Data

Sesión 13



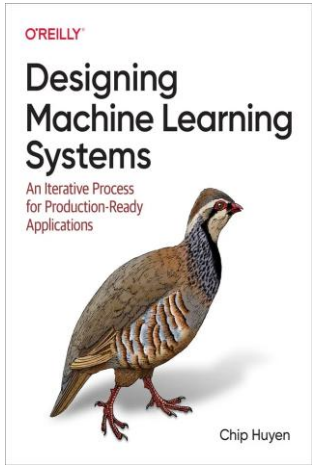
Machine Learning Operations (MLOps)

Parte 2

✅ 📊 Recursos base para esta sesión: MLOps

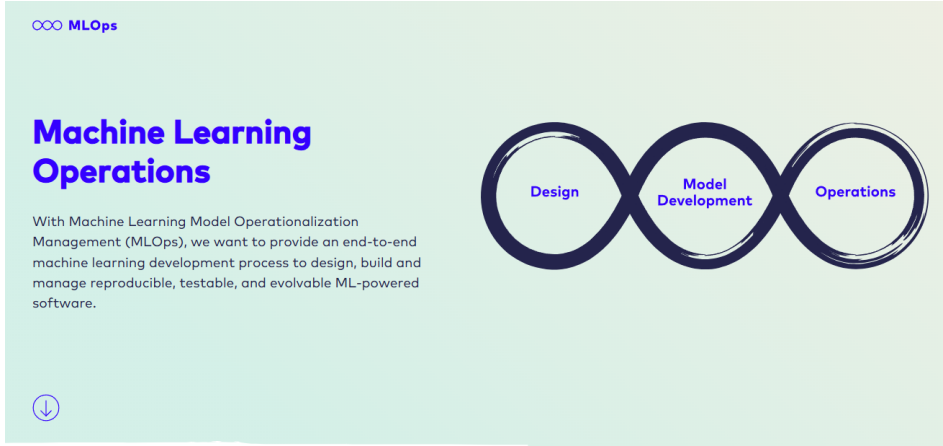
📚 **Libro:**

- **Libro:** *Designing Machine Learning Systems* — Chip Huyen
- **Idea clave:** “Este libro acompaña todo el curso y explica, paso a paso, cómo diseñar sistemas ML listos para producción.”



🌐 **Sitio web:**

- **Página web:** https://ml-ops.org/
- **Idea clave:** “Este portal resume prácticas y principios MLOps de forma visual y simple. Toda la parte de ciclo iterativo y fases sale de ahí.”



✅ 📊 FASE 1

1. 📊 **Notebook de Exploración** (01_data_exploration.ipynb)
   - Análisis del dataset NYC Taxi
   - Identificación de features importantes
   - Detección de outliers y problemas de calidad
2. 🏗️ **Arquitectura Hexagonal Simple**
   - entities.py - Entidades DDD (TaxiTrip, Location, Prediction)
   - ports.py - Interfaces/Puertos
   - services.py - Business Logic
   - config.py - Configuración centralizada
3. 📋 **Documentación y Setup**
   - README.md - Documentación completa
   - requirements.txt - Dependencias

🎯 **Principios aplicados:**

- ✅ **Arquitectura Hexagonal** - Sin over-engineering
- ✅ **DDD (Domain Driven Design)** - Entidades bien definidas
- ✅ **Separation of Concerns** - Cada capa tiene responsabilidad clara
- ✅ **Interfaces claras** - Puertos para adaptadores

✅ 📊 FASE 1.5

🚀 ¿Listo para FASE 2: AWS PostgreSQL?

Ahora necesitamos:

1.Crear base de datos PostgreSQL en AWS RDS
2.Migrar el dataset CSV a PostgreSQL
3.Implementar el primer adaptador (PostgreSQLAdapter)

"Empezamos con arquitectura hexagonal simple aplicando DDD, definiendo claramente las entidades del dominio (TaxiTrip, Location, Prediction) y los servicios de negocio, evitando over-engineering pero manteniendo buenas prácticas profesionales."

✅ 📊 FASE 2: Crear PostgreSQL en AWS RDS

Paso 1: Ir a RDS
1.En la consola de AWS, busca **"RDS"** en la barra de búsqueda
2.Haz clic en **"RDS"** (Relational Database Service)
Paso 2: Crear Base de Datos
1.Una vez en RDS, haz clic en **"Create database"**
2.Selecciona las siguientes opciones:
Engine options:
- **Engine type**: PostgreSQL
- **Version**: PostgreSQL 15.4 (o la más reciente)

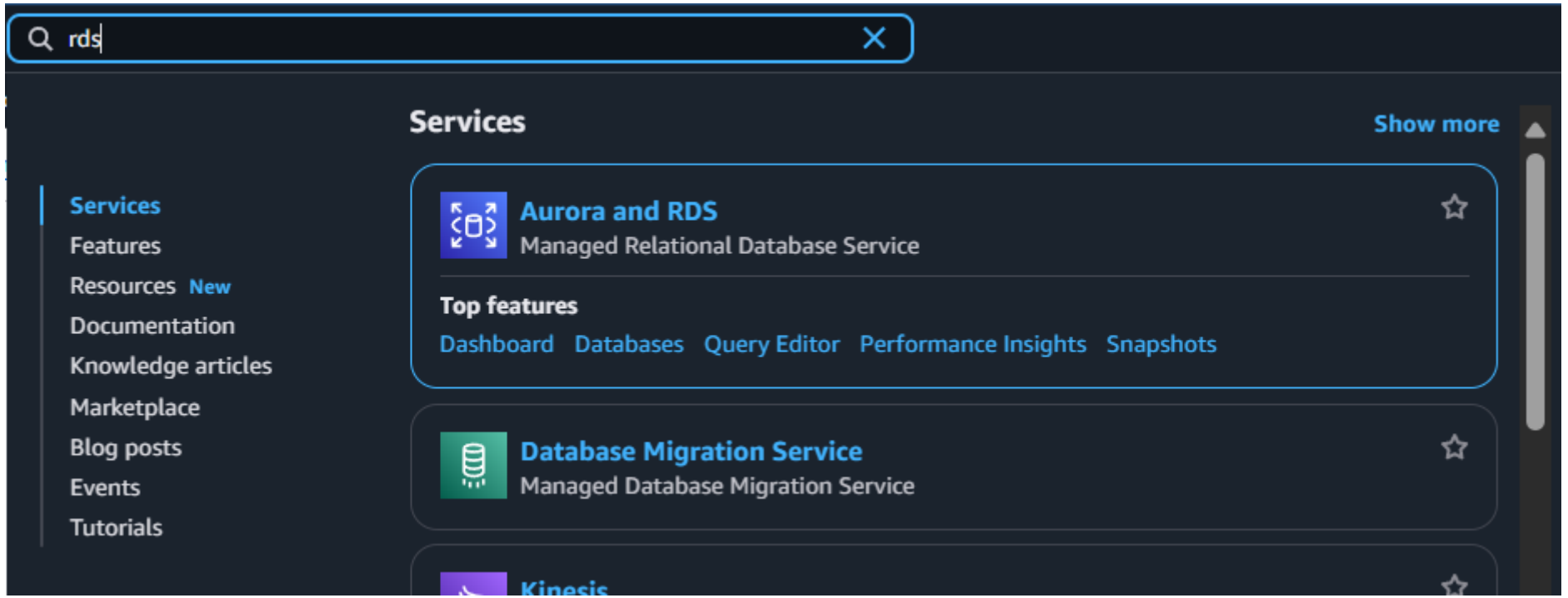
Templates:
- ✅ **Free tier** (esto es importante para no generar costos)

Settings:
- **DB instance identifier**: taxi-duration-db
- **Master username**: taxiuser
- **Master password**: TaxiDB2025! (guarda esta contraseña)
- **DB instance class:**
- **db.t3.micro** (incluido en Free Tier)

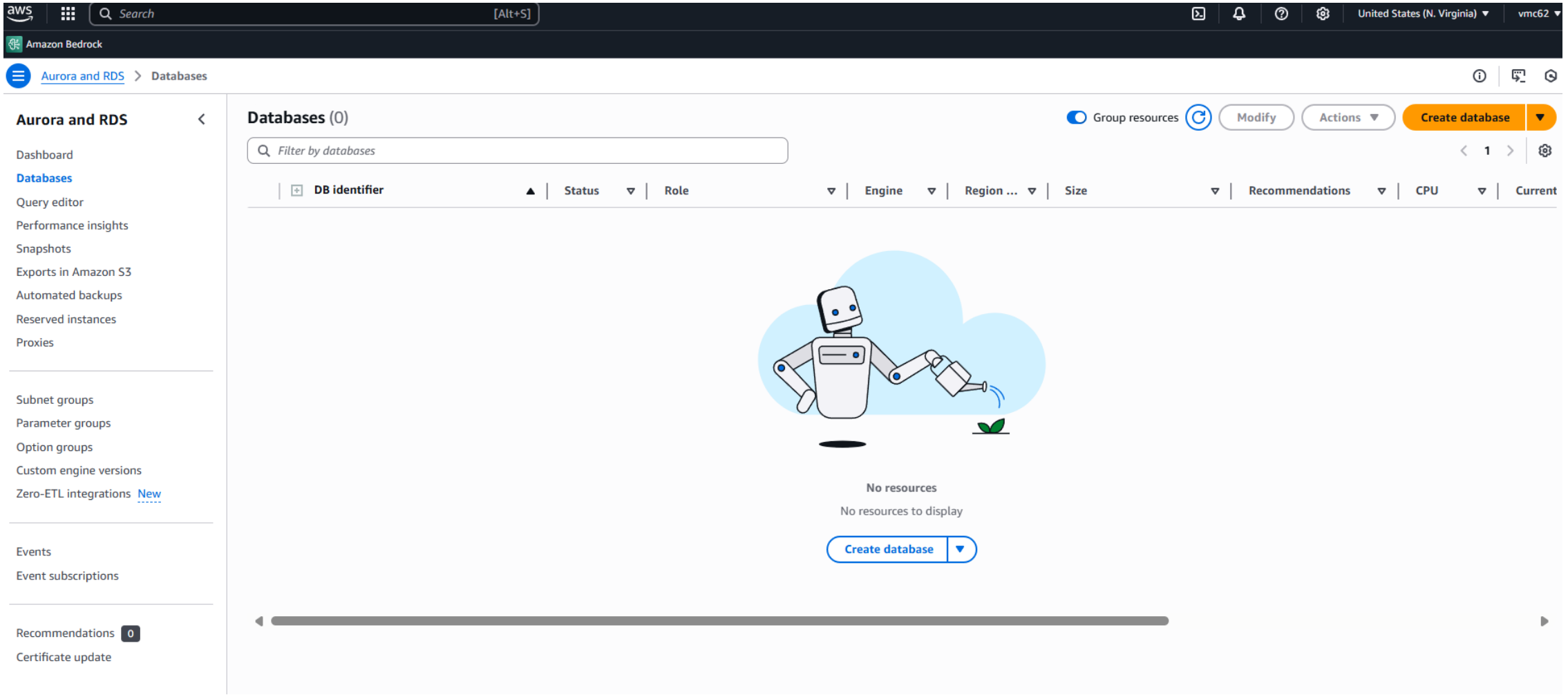
Storage:
- **Storage type**: gp2
- **Allocated storage**: 20 GB (mínimo gratuito)

Connectivity:
- **Public access**: ✅ **Yes** (para conectarnos desde tu máquina)
- **VPC security group**: Create new
- **Security group name**: taxi-db-sg

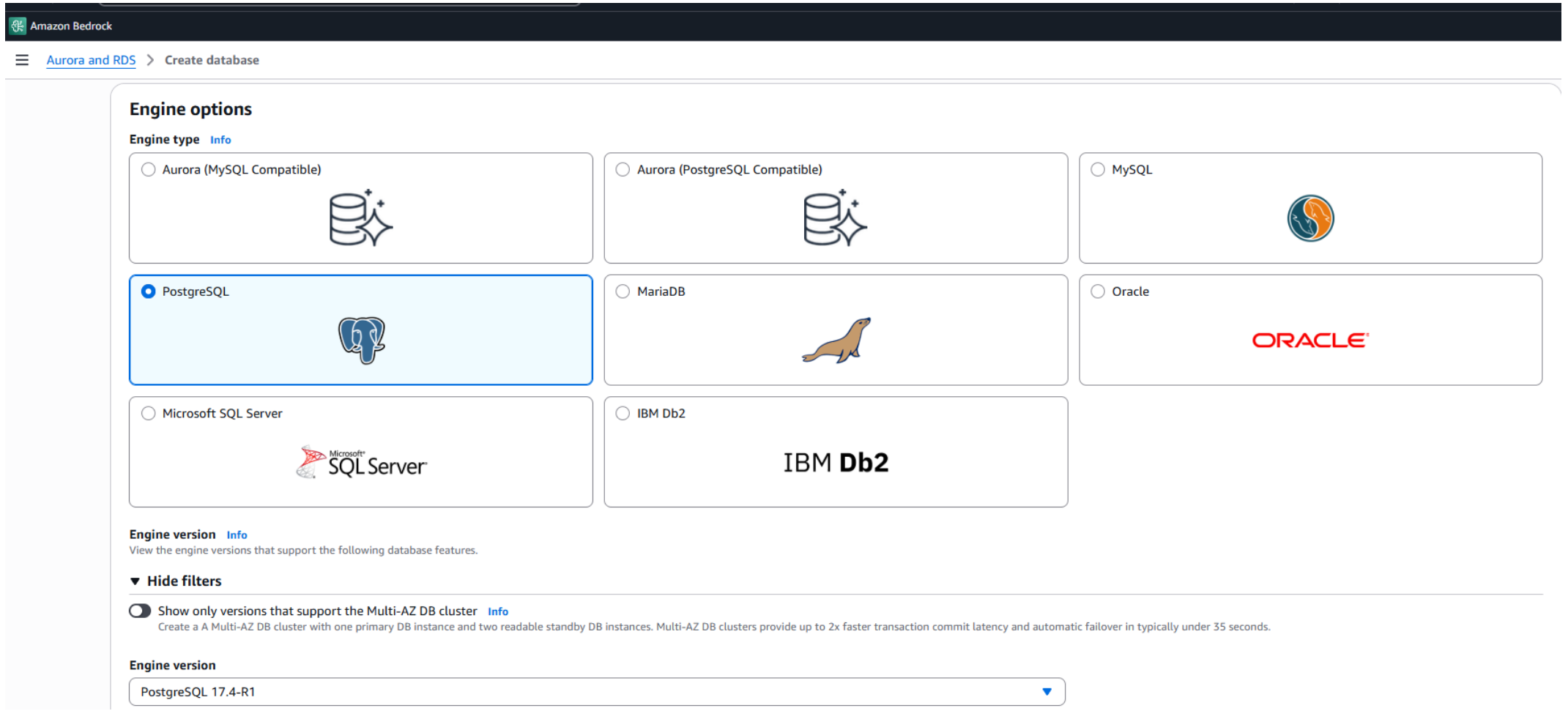
✅ 📊 FASE 2: Crear PostgreSQL en AWS RDS



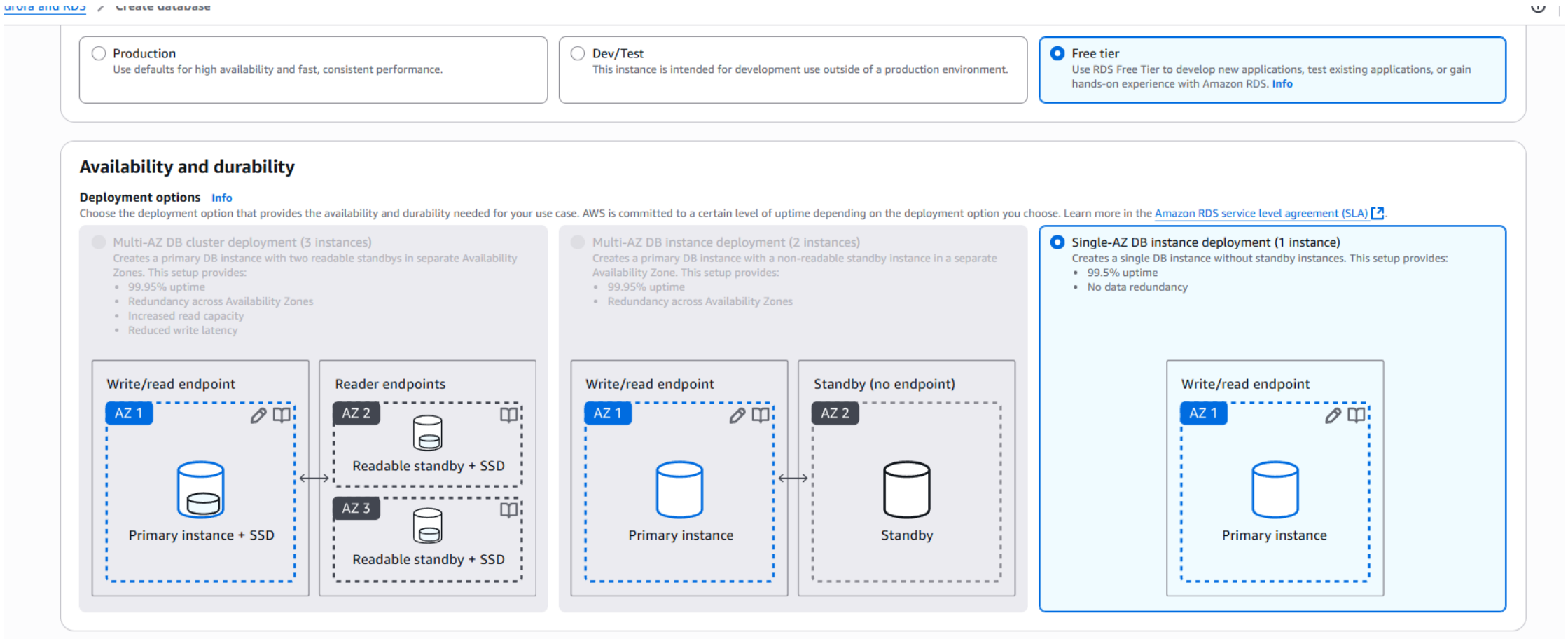
✅ 📊 FASE 2: Crear PostgreSQL en AWS RDS



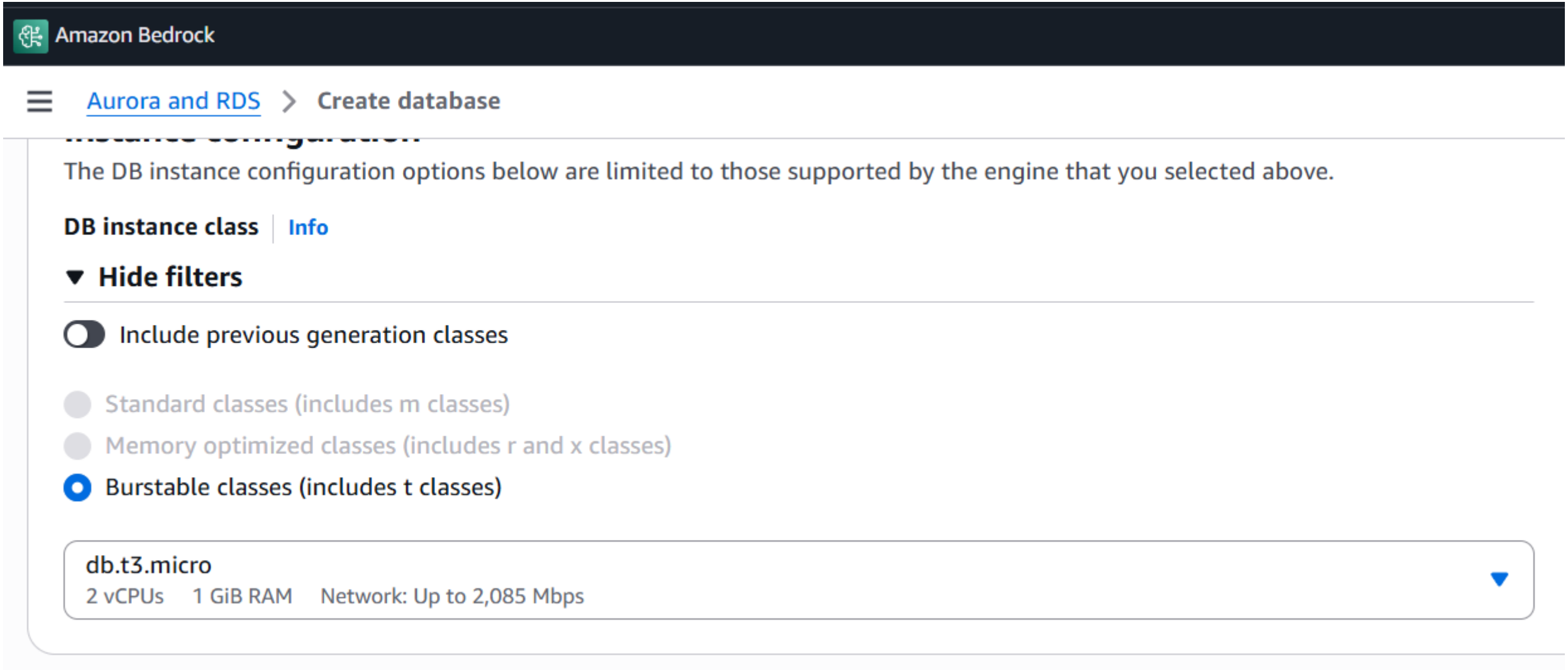
✅ 📊 FASE 2: Crear PostgreSQL en AWS RDS



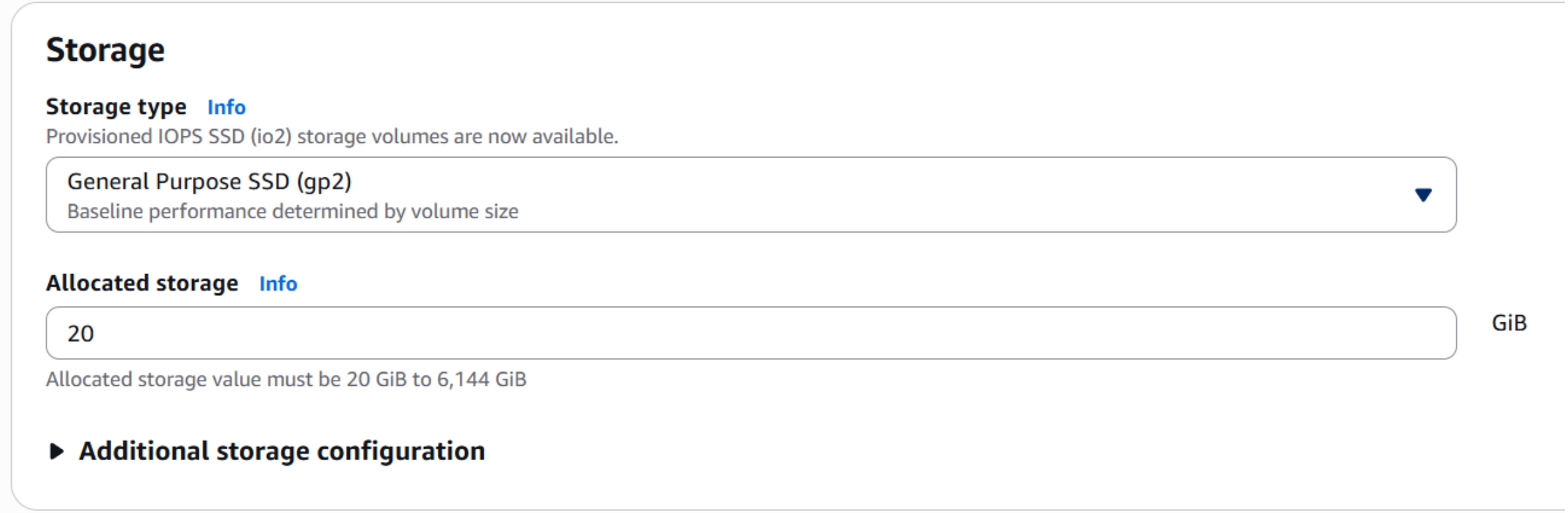
✅ 📊 FASE 2: Crear PostgreSQL en AWS RDS



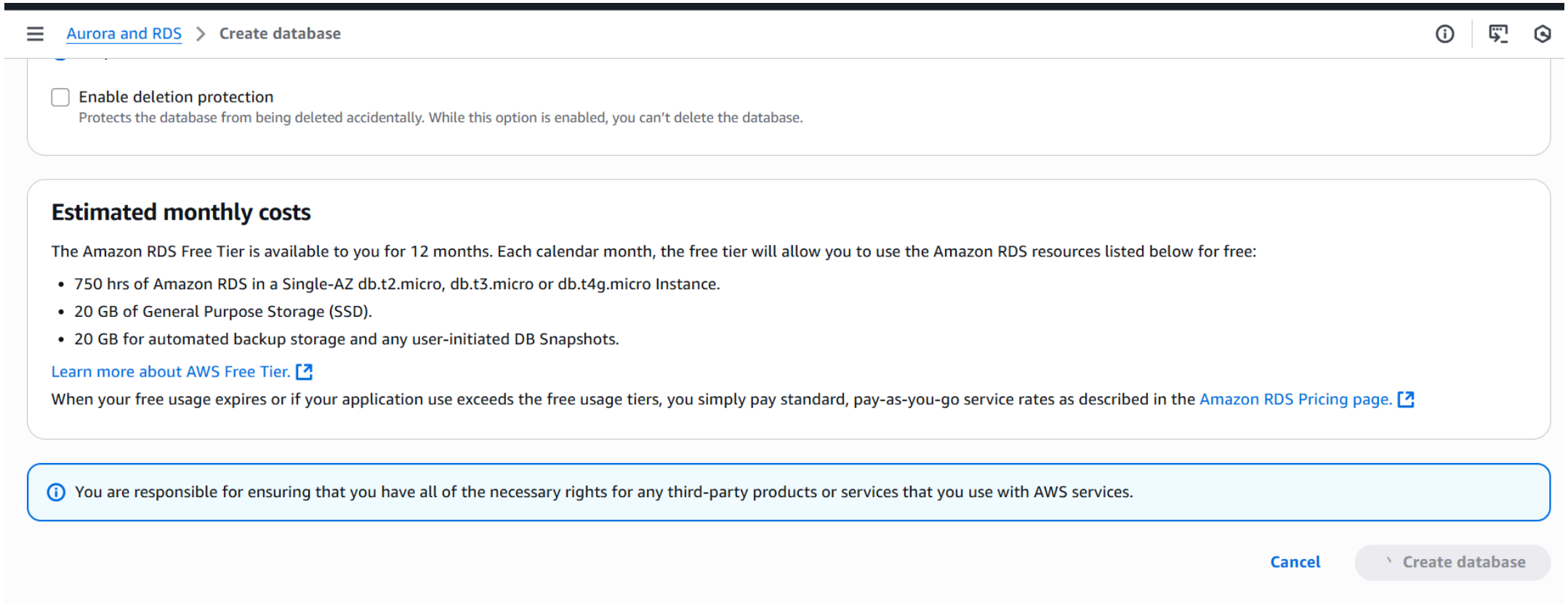
✅ 📊 FASE 2: Crear PostgreSQL en AWS RDS



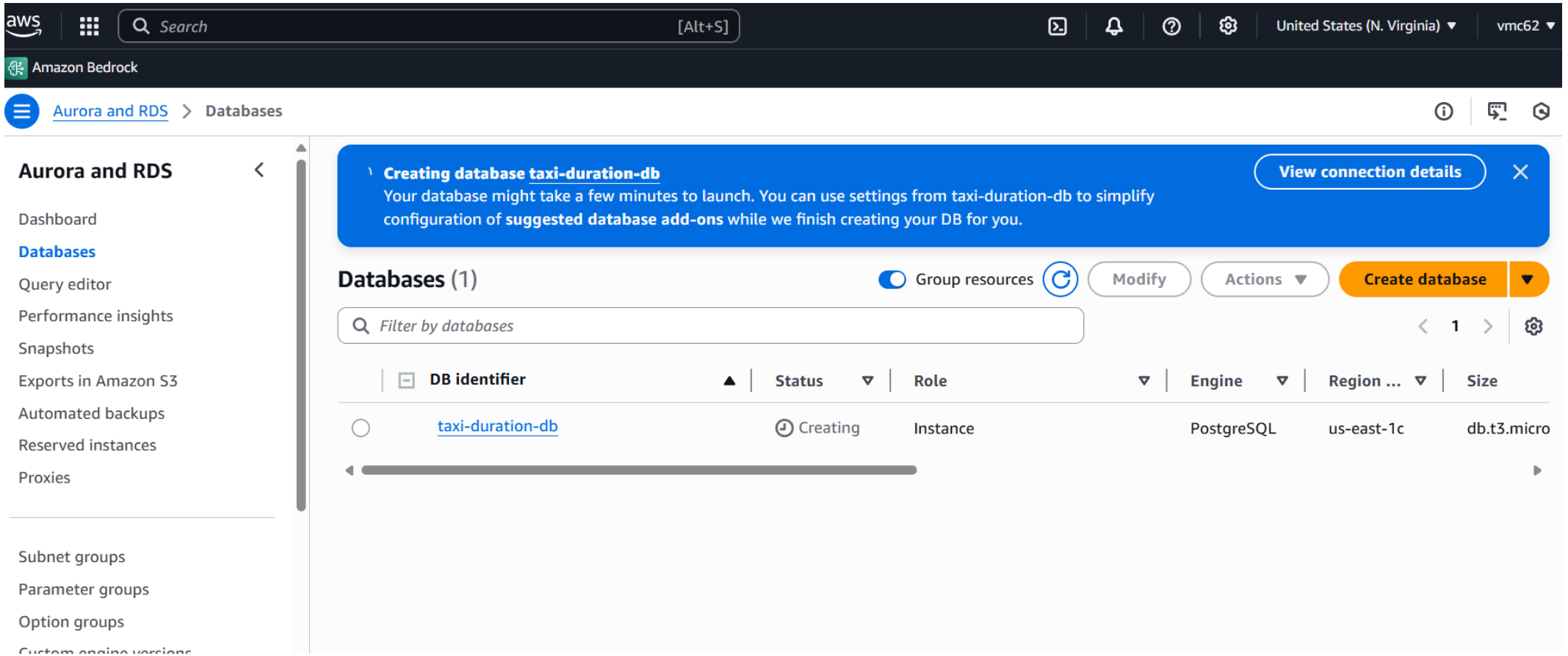
✅ 📊 FASE 2: Crear PostgreSQL en AWS RDS



✅ 📊 FASE 2: Crear PostgreSQL en AWS RDS



✅ 📊 FASE 2: Crear PostgreSQL en AWS RDS



Demora de 10 a 15 minutos

✅ 📊 FASE 2: Crear PostgreSQL en AWS RDS

Paso 3: Configuración de Settings
Baja un poco en la página hasta encontrar la sección **"Settings"** y configura:
DB cluster identifier: taxi-duration-cluster **Master username:** taxiuser **Master password:** TaxiDB2025! **Confirm password:** TaxiDB2025!
Paso 4: Configuración de Connectivity
En la sección **"Connectivity"**, asegúrate de que esté configurado:
Public access: ✅ **Yes** (muy importante para conectarnos) **VPC security group:**
•Selecciona "Create new"
•**New VPC security group name:** taxi-db-sg (ya lo tienes bien)
Paso 5: Configuración Adicional
En **"Additional configuration"**: **Initial database name:** taxi_db
Paso 6: Crear la base de datos
Una vez configurado todo, haz clic en **"Create database"**

✅ 📊 FASE 2.5

Mientras esperamos que termine de crearse la base de datos, preparamos nuestro adaptador:

PostgreSQLAdapter - Implementación completa del puerto TripRepository

1.CSVDataLoader - Utilidad para migrar datos en lotes
2.Notebook de setup - 02_database_setup.ipynb con todas las funciones
⏳ **Esperando:**

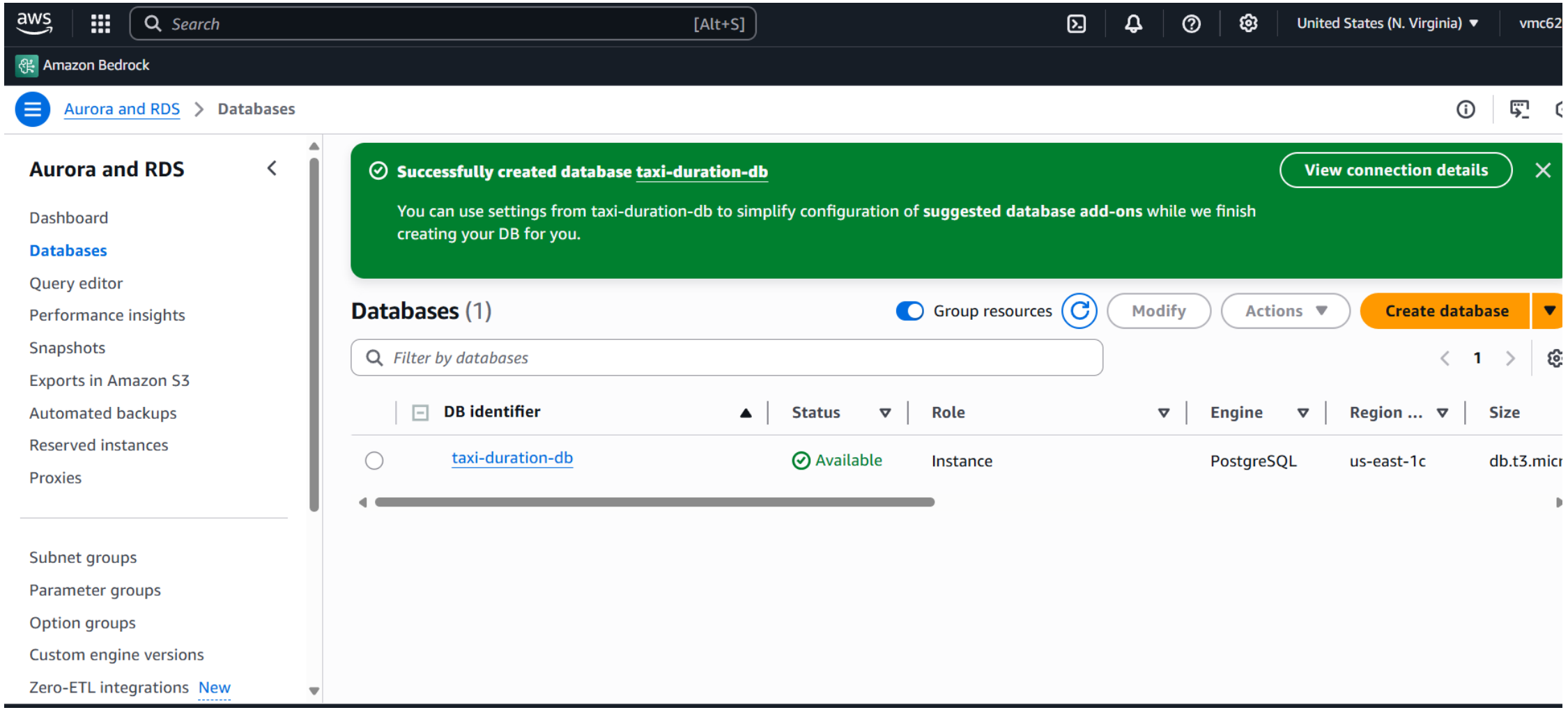
•**Base de datos AWS** termine de crearse (~10-15 minutos)
🔄 **Próximo paso:**

1.Obtener endpoint de la base de datos
2.Actualizar configuración en el notebook
3.Probar conexión y migrar datos

🎯 **Idea Clave:**

"Implementamos el primer adaptador de nuestra arquitectura hexagonal: el PostgreSQLAdapter. Este adaptador implementa el puerto TripRepository y nos permite cambiar de almacenamiento CSV a AWS PostgreSQL sin modificar el dominio del negocio. Esto demuestra el poder del patrón hexagonal para mantener el código de negocio independiente de la infraestructura."

✅ 📊 FASE 2.5


aws
Search
[Alt+S]
United States (N. Virginia)
vmc62
Amazon Bedrock
Aurora and RDS
Databases
Aurora and RDS
Dashboard
Databases
Query editor
Performance insights
Snapshots
Exports in Amazon S3
Automated backups
Reserved instances
Proxies
Subnet groups
Parameter groups
Option groups
Custom engine versions
Zero-ETL integrations
New
Successfully created database taxi-duration-db
View connection details
You can use settings from taxi-duration-db to simplify configuration of suggested database add-ons while we finish creating your DB for you.
Databases (1)
Group resources
Modify
Actions
Create database
Filter by databases
1
DB identifier
Status
Role
Engine
Region ...
Size
taxi-duration-db
Available
Instance
PostgreSQL
us-east-1c
db.t3.micr

✅ 📊 FASE 2.5

Ahora necesitamos obtener el endpoint de conexión. Sigue estos pasos:

🔗 Paso 1: Obtener el Endpoint de Conexión

1.Haz clic en "View connection details" (el botón verde que se ve en la imagen)
2.Esto te mostrará el **endpoint** de la base de datos (algo como: taxi-duration-db.cluster-xxxxx.us-east-1.rds.amazonaws.com)

🔧 Paso 2: Actualizar el Notebook

Una vez que tengas el endpoint, vamos a actualizar el notebook. **Ejecuta la segunda celda** del notebook 02_database_setup.ipynb para hacer el setup.
Luego, en la **celda 4** del notebook, necesitas actualizar esta línea:

📋 Pasos a seguir:

1.Haz clic en "View connection details" y copia el endpoint
2.Reemplaza "PONER_AQUI_EL_ENDPOINT_REAL" en la celda 4 con el endpoint real
3.Ejecuta las celdas 2, 3 y 4 del notebook
4.Ejecuta await test_connection() en la celda 5

🔒 Configurar Security Group

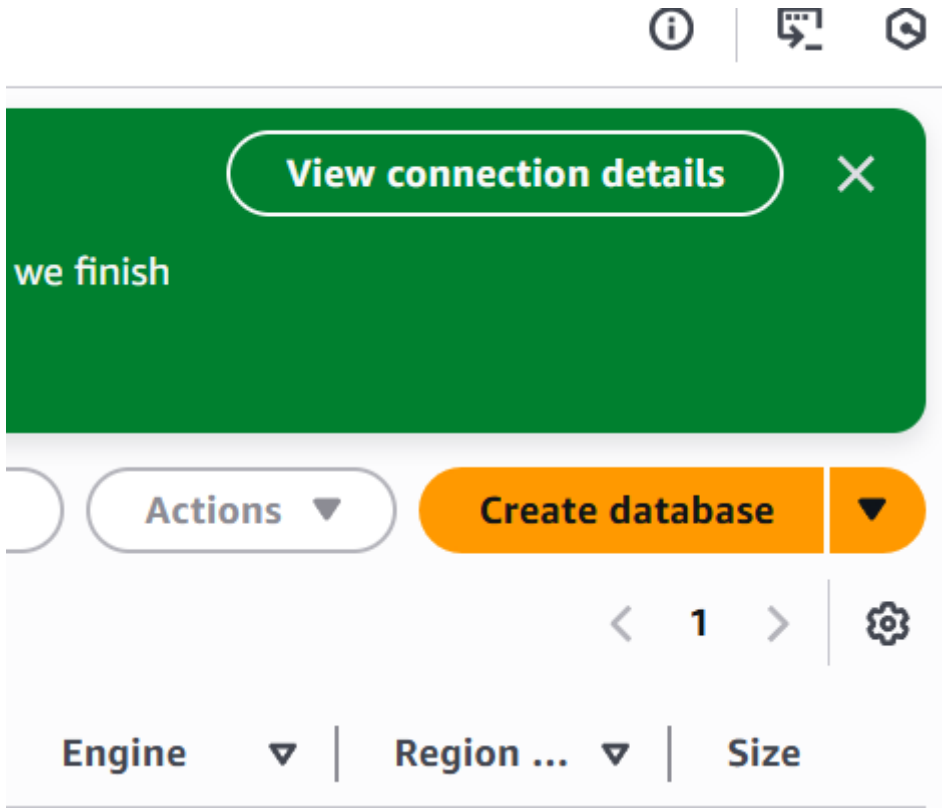
También necesitamos configurar el security group para permitir conexiones:
1.Ve a **EC2 > Security Groups** en AWS
2.Busca el security group taxi-db-sg
3.Agrega una regla **Inbound** para PostgreSQL (puerto 5432) desde tu IP

✅ 📊 FASE 2.5

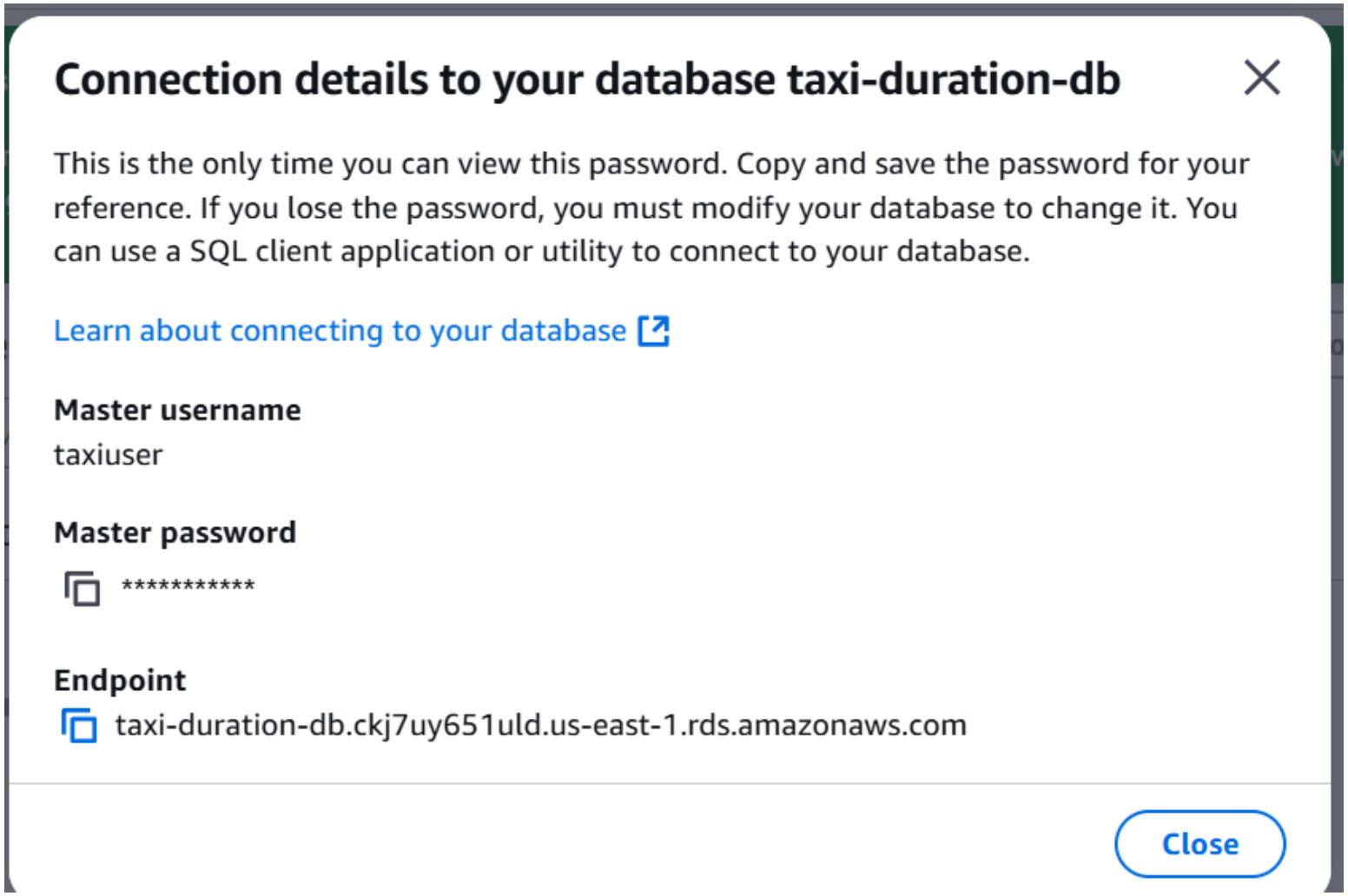
Configurar Security Group:

1. Ve a **EC2 > Security Groups** en AWS console
2. Busca y selecciona taxi-db-sg
3. Ve a la pestaña **"Inbound rules"**
4. Haz clic en **"Edit inbound rules"**
5. Agrega una nueva regla:
    - **Type**: PostgreSQL
    - **Port**: 5432
    - **Source**: My IP (esto pondrá tu IP automáticamente)
6. Guarda los cambios

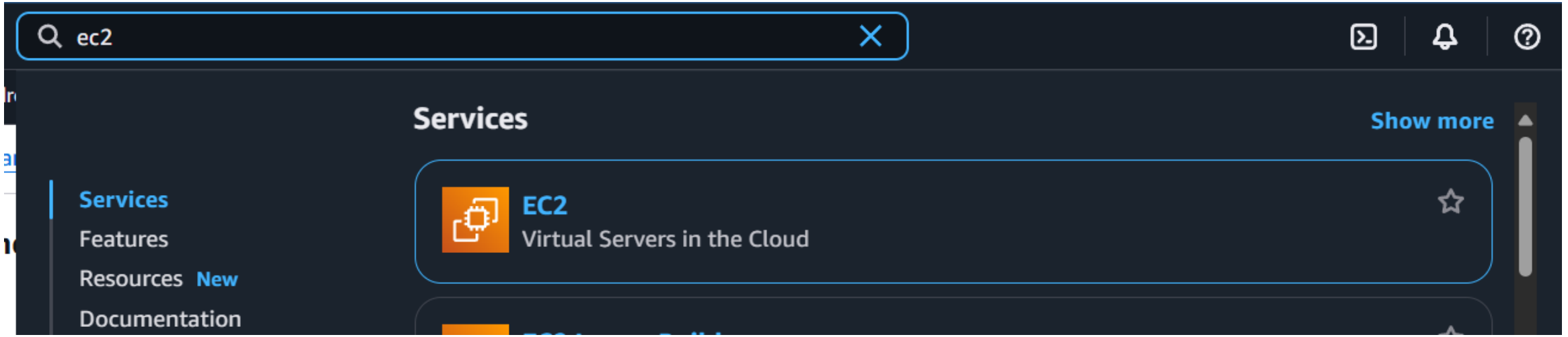
✅ 📊 FASE 2.5

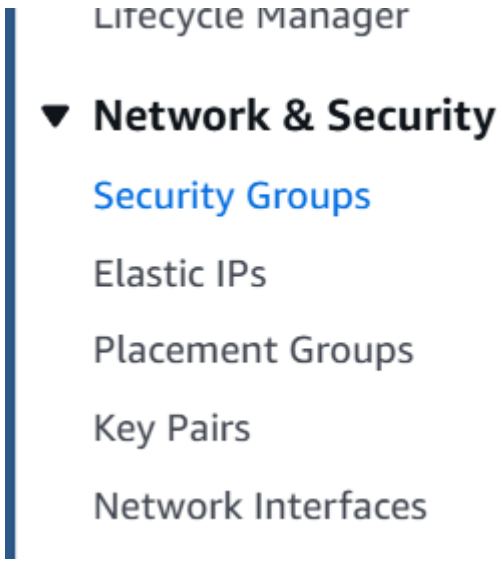

View connection details
we finish
Actions
Create database
1
Engine
Region ...
Size

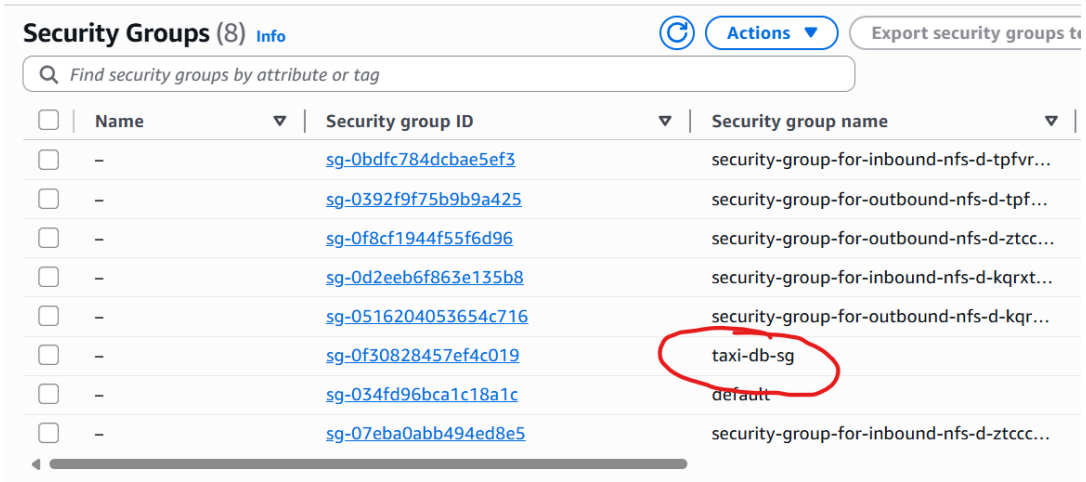
✅ 📊 FASE 2.5


Connection details to your database taxi-duration-db
This is the only time you can view this password. Copy and save the password for your reference. If you lose the password, you must modify your database to change it. You can use a SQL client application or utility to connect to your database.
Learn about connecting to your database
Master username
taxiuser
Master password
****
Endpoint
taxi-duration-db.ckj7uy651uld.us-east-1.rds.amazonaws.com
Close

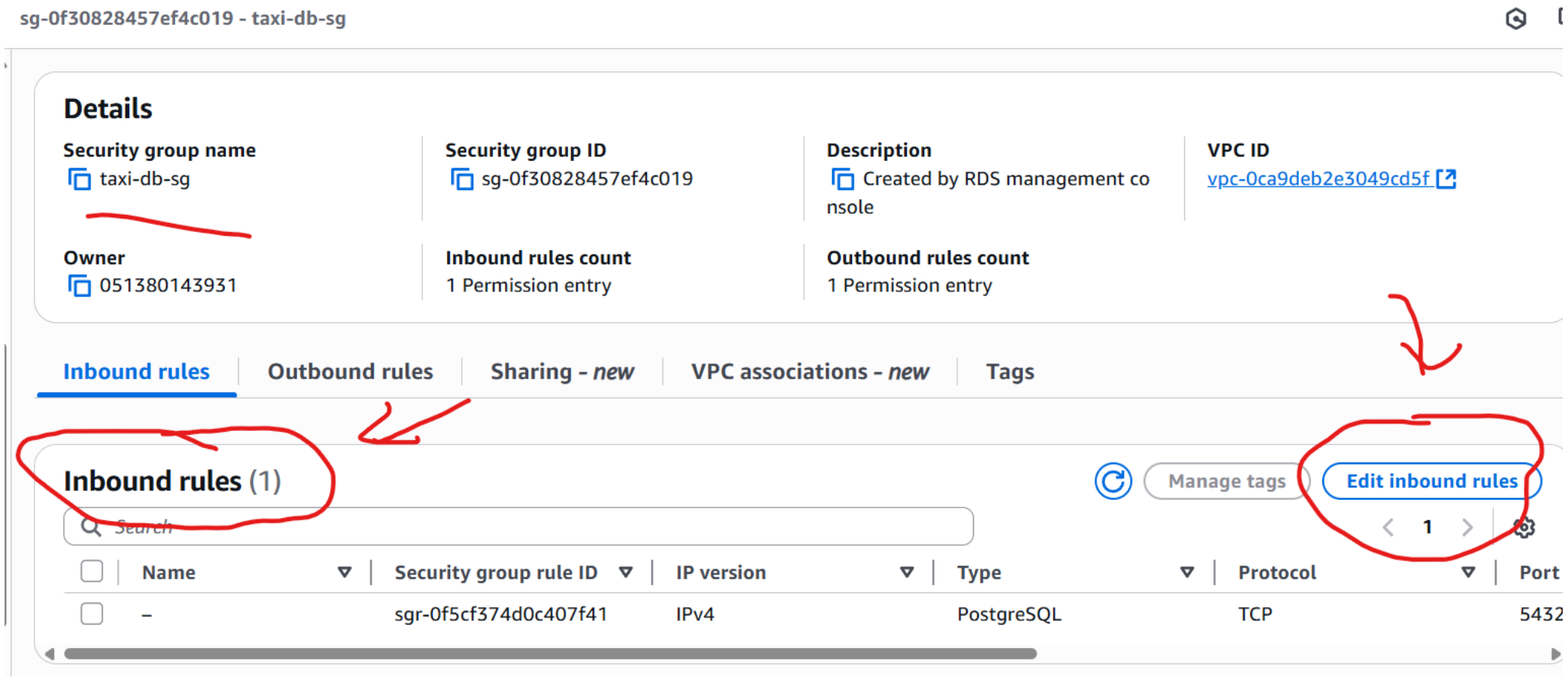
✅ 📊 FASE 2.5


ec2
Services
Show more
Services
Features
Resources New
Documentation
EC2
Virtual Servers in the Cloud

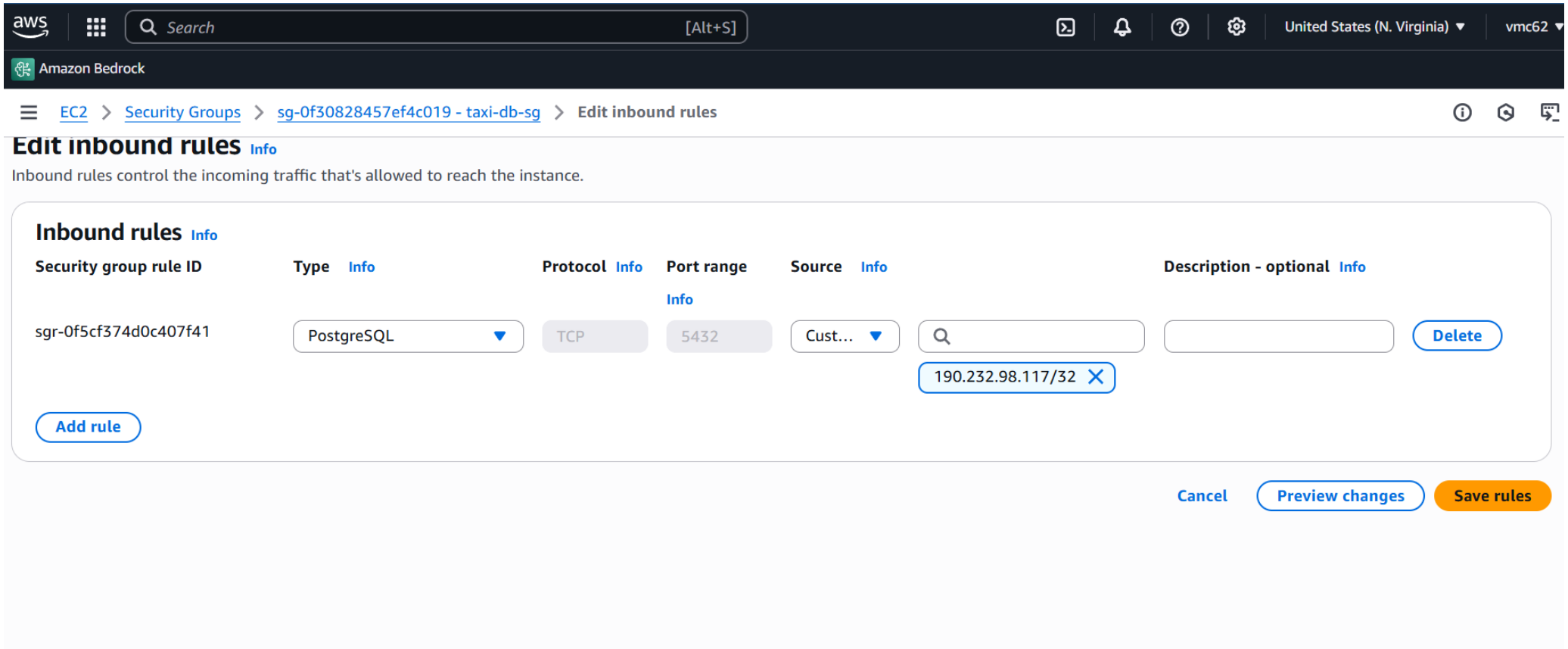

Lifecycle Manager
Network & Security
Security Groups
Elastic IPs
Placement Groups
Key Pairs
Network Interfaces


Security Groups (8) Info
Actions
Export security groups t
Find security groups by attribute or tag
Name
Security group ID
Security group name
sg-0bdfc784dcbae5ef3 security-group-for-inbound-nfs-d-tpfvr...
sg-0392f9f75b9b9a425 security-group-for-outbound-nfs-d-tpf...
sg-0f8cf1944f55f6d96 security-group-for-outbound-nfs-d-ztcc...
sg-0d2eeb6f863e135b8 security-group-for-inbound-nfs-d-kqrxt...
sg-0516204053654c716 security-group-for-outbound-nfs-d-kqr...
sg-0f30828457ef4c019 taxi-db-sg
sg-034fd96bca1c18a1c default
sg-07eba0abb494ed8e5 security-group-for-inbound-nfs-d-ztccc...

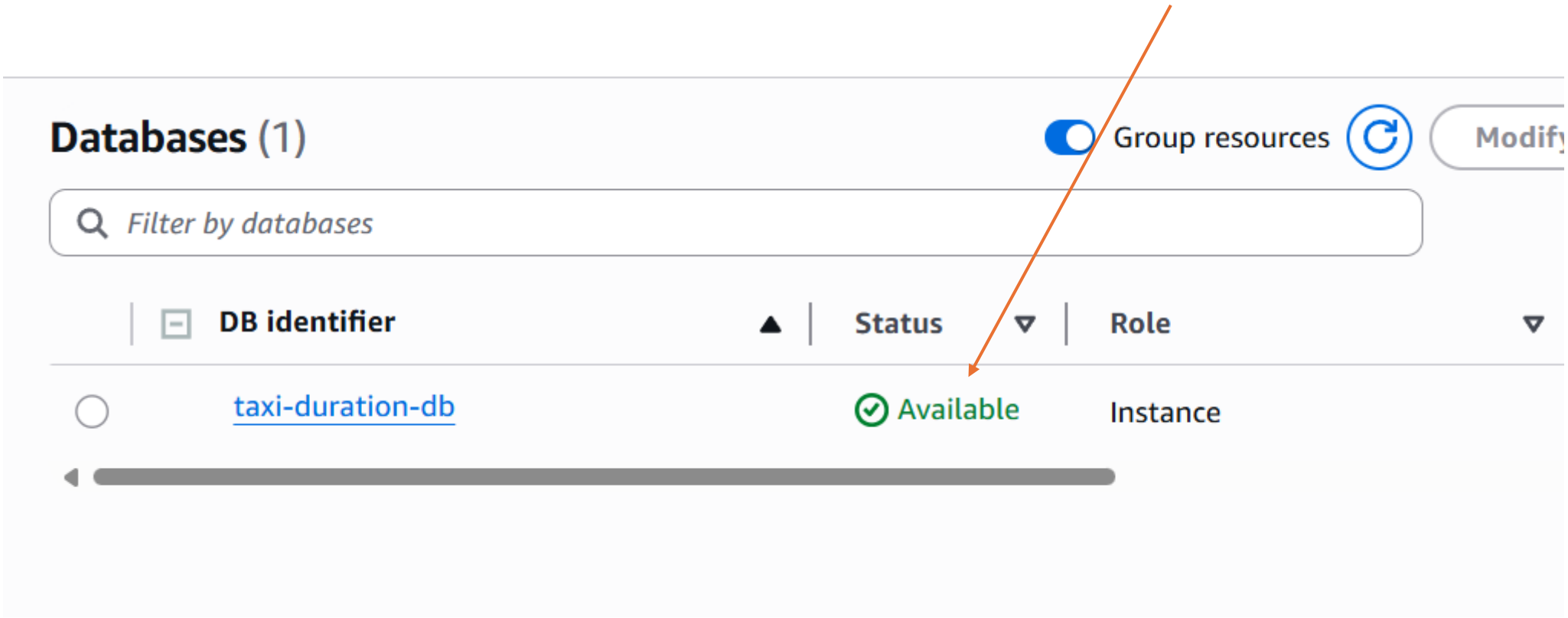
✅ 📊 FASE 2.5


sg-0f30828457ef4c019 - taxi-db-sg
Details
Security group name
taxi-db-sg
Security group ID
sg-0f30828457ef4c019
Description
Created by RDS management console
VPC ID
vpc-0ca9deb2e3049cd5f
Owner
051380143931
Inbound rules count
1 Permission entry
Outbound rules count
1 Permission entry
Inbound rules
Outbound rules
Sharing - new
VPC associations - new
Tags
Inbound rules (1)
Search
Manage tags
Edit inbound rules
1
Name
Security group rule ID
IP version
Type
Protocol
Port
–
sgr-0f5cf374d0c407f41
IPv4
PostgreSQL
TCP
5432

✅📊 FASE 2.5


aws
Search
[Alt+S]
United States (N. Virginia)
vmc62
Amazon Bedrock
EC2 > Security Groups > sg-0f30828457ef4c019 - taxi-db-sg > Edit inbound rules
Edit inbound rules Info
Inbound rules control the incoming traffic that's allowed to reach the instance.
Inbound rules Info
Security group rule ID
Type Info
Protocol Info
Port range Info
Source Info
Description - optional Info
sgr-0f5cf374d0c407f41
PostgreSQL
TCP
5432
Cust...
190.232.98.117/32
Delete
Add rule
Cancel
Preview changes
Save rules

✅ 📊 FASE 2.5


Databases (1)
Group resources
Modify
Filter by databases
DB identifier
Status
Role
taxi-duration-db
Available
Instance

✅ 📊 FASE 2.5

🧪 Una vez que veas "Available", ejecuta esto:
Agregar instrucciones claras sobre qué hacer antes de ejecutar test_connection()

📝 Resumen de lo que debes hacer AHORA:

1️⃣ Verificar AWS (5-10 minutos)
- Ve a AWS Console → RDS
- Busca taxi-duration-db
- **Espera hasta que diga "Available"**

2️⃣ Probar conexión
Una vez que veas "Available", ejecuta esta celda:

```
await test_connection()
```

- **3️⃣ Si la conexión es exitosa, continúa con:**

```
await create_database_schema()

await migrate_csv_data(50000)

await validate_migrated_data()
```

• 🎯 ¿Por qué este orden?

- **AWS RDS tarda tiempo** en crear la instancia (5-15 minutos)
- **No puedes conectarte** hasta que esté "Available"
- **El Security Group** debe estar configurado correctamente
- **Una vez conectado**, todo fluye automáticamente

✅ 📊 FASE 2.5

✅ **Resultados exitosos:**

1. ✅ **Conexión AWS PostgreSQL**:
   - Conectado exitosamente a PostgreSQL 17.4
   - Endpoint configurado correctamente
2. ✅ **Esquema de base de datos**:
   - Tablas taxi_trips y predictions creadas
   - Índices optimizados aplicados
3. ✅ **Migración de datos**:
   - **49,719 viajes** migrados exitosamente
   - Datos limpios del CSV (de 50,000 originales)
   - Proceso de lotes funcionando perfectamente
4. ✅ **Validación de datos**:
   - **Duración promedio**: 13.9 minutos
   - **Pasajeros promedio**: 1.7
   - **Rango temporal**: Enero a Junio 2016
   - **2 vendors únicos**

✅ 📊 FASE 2.5

📐 Arquitectura lograda:

📊 CSV Data → 🔄 Data Cleaning → 🗄️ AWS PostgreSQL → ✅ Validation

1. **Arquitectura Hexagonal**: El PostgreSQLAdapter implementa perfectamente el puerto TripRepository
2. **Escalabilidad**: De CSV local a base de datos en la nube
3. **Calidad de datos**: Validación y limpieza automática
4. **MLOps real**: Infraestructura preparada para producción

✅ 📊 FASE 3: MLflow + Entrenamiento de Modelos

📋 **Objetivos de FASE 3:**

1. 🔬 MLflow Tracking Server
•Configurar MLflow para rastrear experimentos
•Registrar métricas, parámetros y modelos
•Comparar diferentes algoritmos y hiperparámetros

2. 🤖 Adaptador de Machine Learning
•Implementar MLModelService del puerto que ya definimos
•Usar scikit-learn para modelos de regresión
•Feature engineering automatizado

3. 📊 Pipeline de Entrenamiento
•Extraer datos desde PostgreSQL
•Generar features (distancia, hora del día, día de semana)
•Entrenar múltiples modelos (Random Forest, XGBoost, Linear Regression)
•Validar y seleccionar el mejor modelo

4. 💾 Persistencia de Modelos
•Guardar modelos entrenados en MLflow
•Versionado automático
•Metadata de experimentos

✅ 📊 FASE 3

📐 Arquitectura que vamos a implementar:

🗄️ PostgreSQL → 🔄 Feature Engineering → 🤖 ML Training → 📊 MLflow → 💾 Model Registry

MLOps Fundamentals:
- **Experiment tracking**: Cómo rastrear y comparar experimentos
- **Model versioning**: Versionado automático de modelos
- **Reproducibilidad**: Mismos resultados en diferentes ejecuciones

Arquitectura Hexagonal:
- **Separation of concerns**: ML logic separado de infrastructure
- **Testability**: Cada adaptador se puede probar independientemente
- **Flexibility**: Fácil cambiar de scikit-learn a otro framework

🎯 Lo que vamos a lograr:

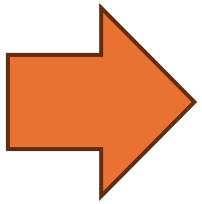
- **3 modelos entrenados** (RandomForest, XGBoost, LinearRegression)
- **MLflow tracking** de todos los experimentos
- **Comparación automática** de métricas
- **Selección del mejor modelo**
- **UI visual** para ver los resultados

✅ 📊 FASE 3

Revisar el notebook: 03_mlflow_training.ipynb

🚀 **paso a paso:**

Paso 1: Ejecutar Setup
Paso 2: Configurar MLflow
Paso 3: Preparar Feature Engineering
Paso 4: Preparar Pipeline de Entrenamiento
Paso 5: ¡EJECUTAR LOS EXPERIMENTOS! 🚀



🎉 **¡ÉXITO TOTAL! FASE 3 COMPLETADA**

📊 **Resultados increíbles:**

🏆 **MEJOR MODELO: RandomForest**

- **RMSE: 6.62 minutos** (error promedio)
- **MAE: 4.27 minutos** (error absoluto)
- **R^2: 0.681** (68.1% de varianza explicada)

📈 **Comparación de modelos:**

1. **RandomForest**: 6.62 min RMSE ✅ *Ganador*
2. **XGBoost**: 6.85 min RMSE
3. **LinearRegression**: 7.47 min RMSE

✅ 📊 FASE 3

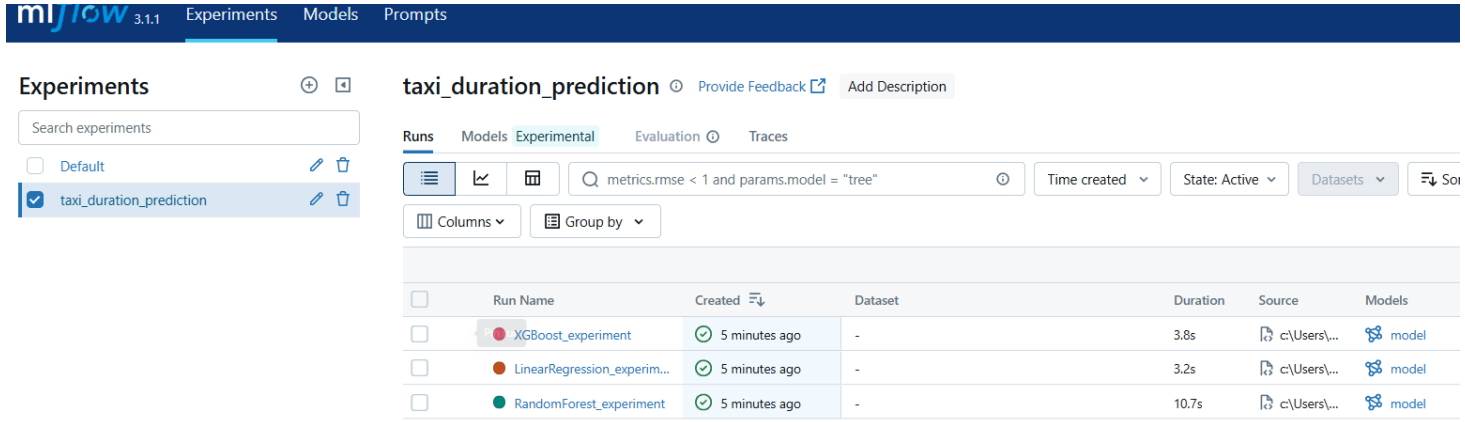
Ver MLflow UI para explorar experimentos

🔬 ¡Ahora puedes ver MLflow UI!

Ejecuta este comando en una terminal:

mlflow ui --backend-store-uri sqlite:///mlflow.db

Luego ve a: http://localhost:5000



🧬 El "Big Picture" de MLflow en producción:

Modo Desarrollo:

Jupyter → MLflow SQLite → MLflow UI (localhost:5000)

✅ 📊 FASE 4: Dashboard Streamlit con MLflow

Ver MLflow UI para explorar experimentos

Modo Producción:

Scripts automatizados → MLflow Server → Dashboard Streamlit → API FastAPI

FASE 4: Dashboard automático que lee experimentos

- **Comparación visual** de modelos
- **Métricas en tiempo real**
- **Selección del mejor modelo automática**

Esto completaría el pipeline end-to-end:

PostgreSQL → MLflow → Streamlit Dashboard → FastAPI → Producción

✅ 📊 FASE 4

- ✅ **Creación de Python script** completo (04_streamlit_dashboard.py)
- ✅ **Dashboard interactivo** con 4 vistas especializadas
- ✅ **Integración MLflow API** programática (sin dependencia de UI manual)
- ✅ **Métricas ejecutivas** en tiempo real
- ✅ **Comparación automática** de modelos
- ✅ **Análisis de datos** de producción desde PostgreSQL
- ✅ **Simulador de predicciones** interactivo

🎨 CARACTERÍSTICAS TÉCNICAS DEL DASHBOARD

- ✅ **Streamlit + Plotly** para visualizaciones profesionales
- ✅ **MLflow.tracking.MlflowClient** para acceso programático
- ✅ **Asyncpg** para consultas PostgreSQL asíncronas
- ✅ **CSS personalizado** y diseño responsivo
- ✅ **Cache inteligente** para optimizar performance
- ✅ **4 vistas especializadas**:
  - 📈 Overview General (métricas ejecutivas)
  - 🤖 Comparación de Modelos (análisis detallado)
  - 📊 Análisis de Datos (estadísticas PostgreSQL)
  - 🎯 Simulador en Tiempo Real (predicciones interactivas)

✅ 📊 FASE 4

🚀 **EJECUCIÓN DEL DASHBOARD STREAMLIT**
Comando en Terminal:

```
source activate ds_env && streamlit run 04_streamlit_dashboard.py
```

•**Resultado:**
• ✅ **URL Local:** http://localhost:8501
• ✅ **URL Red:** http://192.168.1.6:8501
• ✅ **Estado:** Dashboard ejecutándose en background

¿Por qué este comando específico?

1.source activate ds_env - Activa el ambiente conda con las dependencias
2.&& - Ejecuta el siguiente comando solo si el anterior fue exitoso
3.streamlit run 04_streamlit_dashboard.py - Lanza el dashboard programático

Punto Clave:

"Un solo comando que activa el ambiente y lanza el dashboard profesional de MLOps, eliminando la necesidad de navegación manual por localhost de MLflow"

✅ 📊 FASE 4

Resultado Visual:
El dashboard muestra automáticamente:
- 🤖 **3 Experimentos** desde MLflow.db
- 🏆 **Mejor modelo:** RandomForest (6.62 min RMSE)
- 📊 **Comparación automática** de modelos
- 📈 **Análisis en tiempo real** de PostgreSQL
- 🎯 **Simulador interactivo** de predicciones

✅ 📊 FASE 4

🧩 ARQUITECTURA MLOPS COMPLETA

FastAPI vs Containers vs CI/CD - NO son excluyentes:

📊 Streamlit Dashboard (lo que tenemos)

↓

🚀 FastAPI (servir predicciones)

↓

🐳 Docker Container (empaquetar todo)

↓

⚙️ GitHub Actions CI/CD (automatizar deployment)

✅ 📊 FASE 4

🔄 REAL-TIME MONITORING sin datos en vivo

- ✅ **Performance del API** (latencia, requests/segundo)
- ✅ **Drift detection** simulado (comparar distribuciones históricas)
- ✅ **Model degradation** (RMSE tracking over time)
- ✅ **System health** (CPU, memoria, errores)

```
# En FastAPI - cada predicción se logea
@app.post("/predict")
async def predict(data: TripData):
    prediction = model.predict(data)

    # LOG para monitoring
    log_prediction(data, prediction, model_version)

    return {"duration": prediction}
```

✅ 📊 FASE 4

🎯 CI/CD CON GITHUB ACTIONS

GitHub Actions maneja:

- 🔄 **Reentrenamiento automático** (cuando performance baja)
- 🚀 **Deployment automático** del API
- 📊 **Actualización del dashboard**
- 🔔 **Alertas** si algo falla

```
# .github/workflows/mlops.yml
name: MLOps Pipeline
on:
  push:
    branches: [main]
  schedule:
    - cron: '0 2 * * *'  # Reentrenamiento diario

jobs:
  retrain-model:
    - name: Check model performance
    - name: Retrain if needed
    - name: Update MLflow registry

  deploy-api:
    - name: Build Docker image
    - name: Deploy to production
    - name: Update dashboard
```

✅ 📊 FASE 4

📊 STREAMLIT UNIFICADO - Un solo dashboard:

🏠 Streamlit Dashboard Unificado
- ├── 📈 MLOps Overview (lo que tenemos)
- ├── 🤖 Model Performance (MLflow data)
- ├── 🚀 API Monitoring (FastAPI metrics)
- ├── 🔄 CI/CD Status (GitHub Actions status)
- ├── 📊 Real-time Health (system metrics)
- └── 🎯 Predictions (formulario actual)

✅ 📊 FASE 4B: FastAPI

Revisar el archivo 05_fastapi_server.py

```
source activate ds_env && python 05_fastapi_server.py
```

🎉 ¡Si el servidor de FastAPI está ejecutándose correctamente!

🚀 URLs del FastAPI:

- **API Principal**: http://localhost:8000
- **Documentación Swagger**: http://localhost:8000/docs
- **ReDoc**: http://localhost:8000/redoc

✅ 📊 FASE 4B: FastAPI

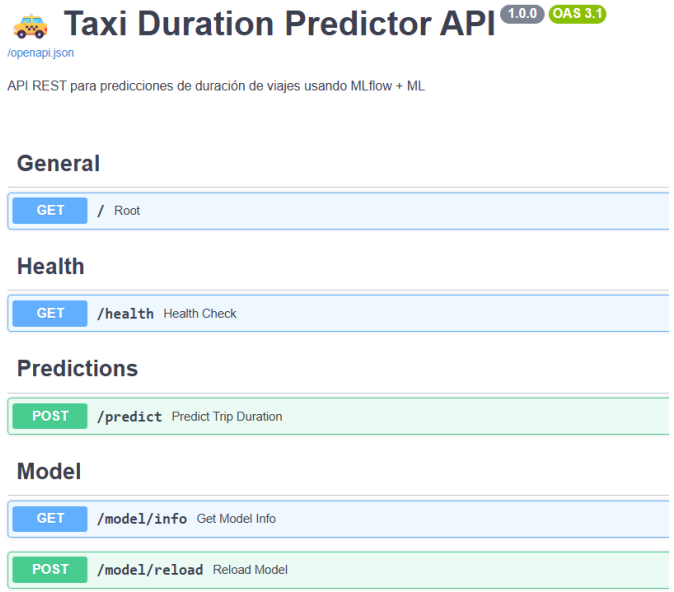
✅ ESTADO ACTUAL - MLOps Stack Completo:

1. 📊 Streamlit Dashboard: http://localhost:8501

- Overview ejecutivo con métricas MLflow
- Comparación de modelos
- Análisis de datos PostgreSQL
- Simulador de predicciones

2. 🚀 FastAPI Server: http://localhost:8000

- ✅ Modelo **RandomForest cargado automáticamente** desde MLflow
- ✅ **Health check** completo (/health)
- ✅ **Endpoint de predicciones** (/predict)
- ✅ **Documentación automática** (/docs)
- ✅ **Estadísticas de DB** (/stats/database)
- ✅ **Recarga de modelo** (/model/reload)



✅ 📊 FASE 4

🎯 ¿Qué sigue ahora?

Opción A: Extender Streamlit

Agregar una nueva pestaña "🚀 API Status" al dashboard de Streamlit que:

- Monitoree el status del FastAPI
- Muestre métricas de uso del API
- Permita hacer predicciones desde el dashboard

Opción B: Crear un Cliente de Prueba

Un script simple que demuestre cómo usar el API programáticamente

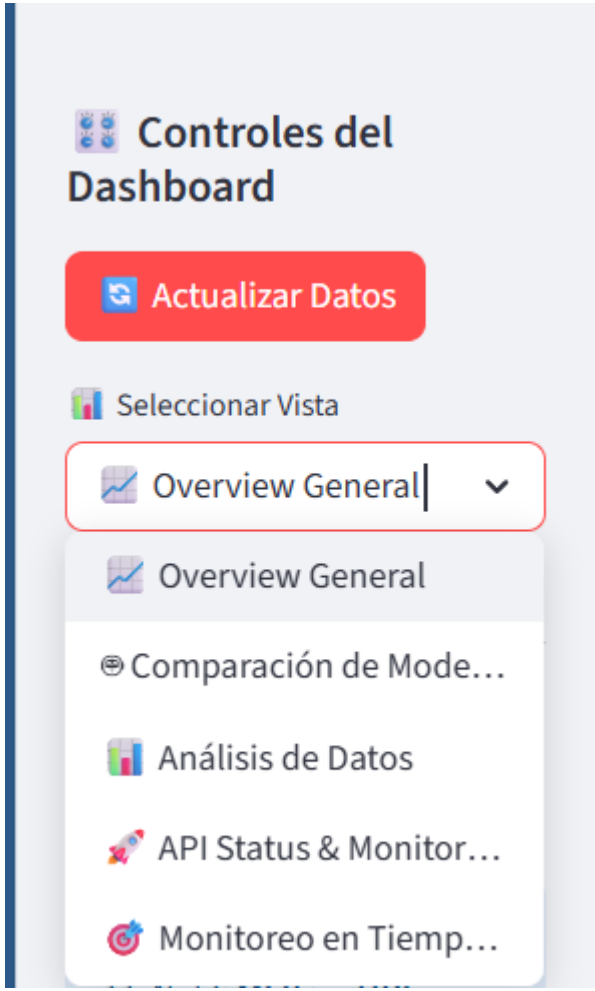
Opción C: Docker + CI/CD Setup

Containerizar todo y configurar GitHub Actions

✅ 📊 FASE 4

Actualizamos script: **04_streamlit_dashboard.py** para crear las nuevas vistas:

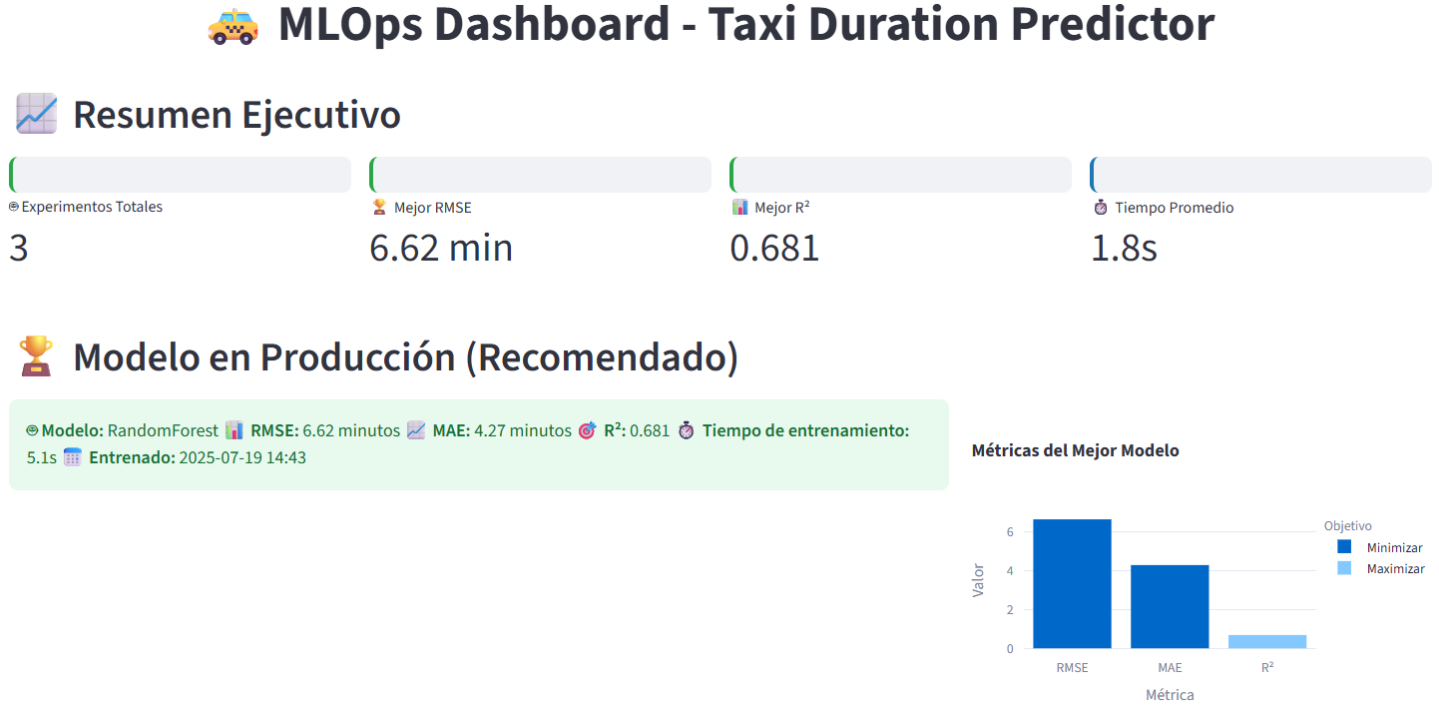
1. 📊 **Streamlit Dashboard**: http://localhost:8501
    1. ✅ **5 vistas completas**:
        1. 📈 Overview General (métricas MLflow)
        2. 🤖 Comparación de Modelos
        3. 📊 Análisis de Datos (PostgreSQL)
        4. 🚀 **API Status & Monitoring** (¡NUEVO!)
        5. 🎯 Monitoreo en Tiempo Real
2. 🚀 **FastAPI Server**: http://localhost:8000
    1. ✅ Modelo RandomForest cargado automáticamente
    2. ✅ Endpoints funcionando: /predict, /health, /docs
    3. ✅ Integración completa con MLflow



✅ 📊 FASE 4

Actualizamos script:
04_streamlit_dashboard.py para crear las nuevas vistas:

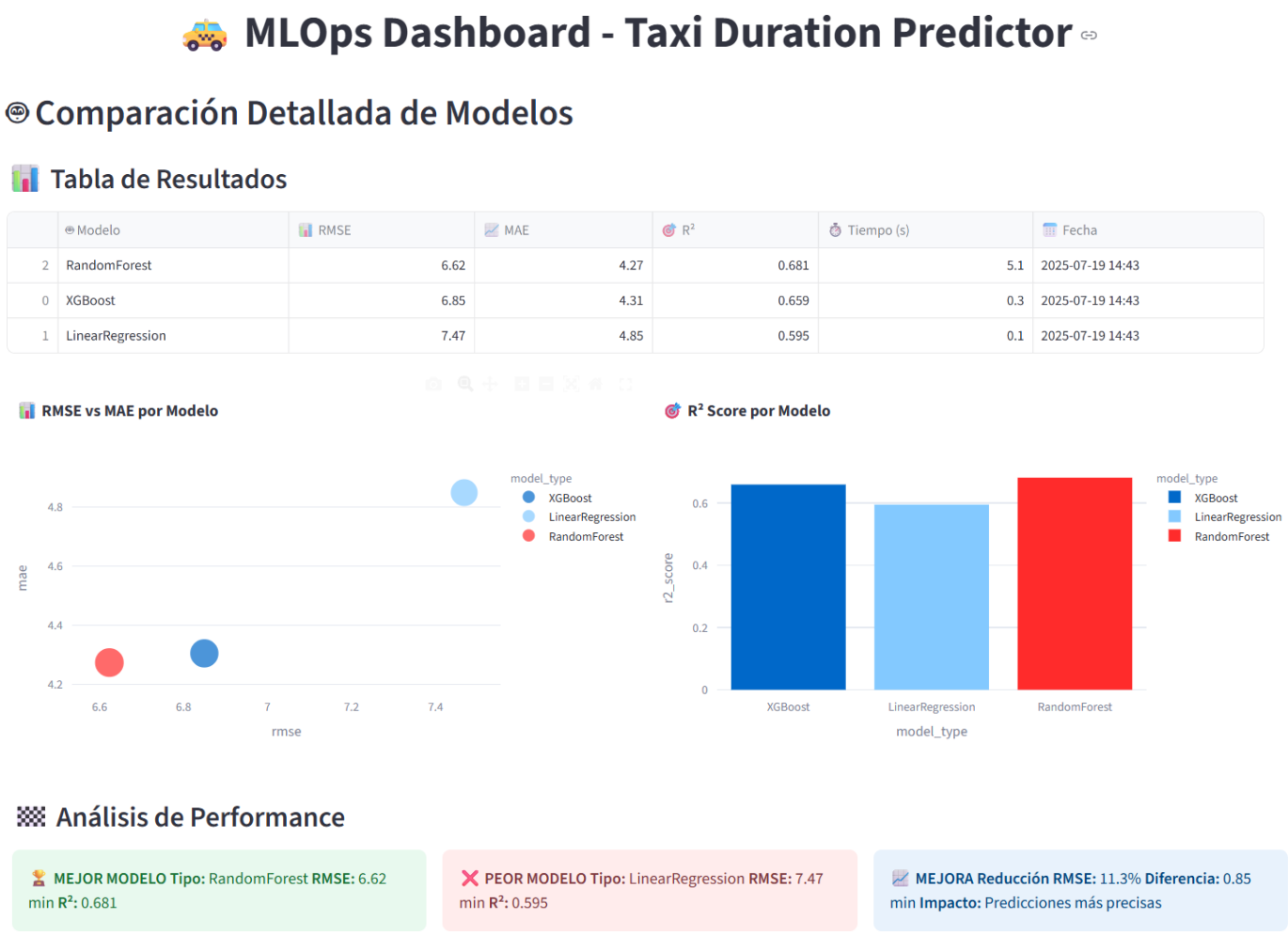
- 📊 **Streamlit Dashboard**: http://localhost:8501
    1. 📈 Overview General (métricas MLflow)



✅ 📊 FASE 4

Actualizamos script:
04_streamlit_dashboard.py para crear las nuevas vistas:

- 📊 **Streamlit Dashboard**: http://localhost:8501
  1. ✅ **5 vistas completas**:
     2. 🤖 Comparación de Modelos



✅ 📊 FASE 4

Actualizamos script:
04_streamlit_dashboard.py para crear las nuevas vistas:

- 📊 **Streamlit Dashboard**: http://localhost:8501
  1. ✅ **5 vistas completas**:
     3. 📊 Análisis de Datos (PostgreSQL)

✅ 📊 FASE 4

Actualizamos script:
04_streamlit_dashboard.py para crear las nuevas vistas:

- 📊 **Streamlit Dashboard**: http://localhost:8501
  1. ✅ **5 vistas completas**:
     5. 🎯 Monitoreo en Tiempo Real

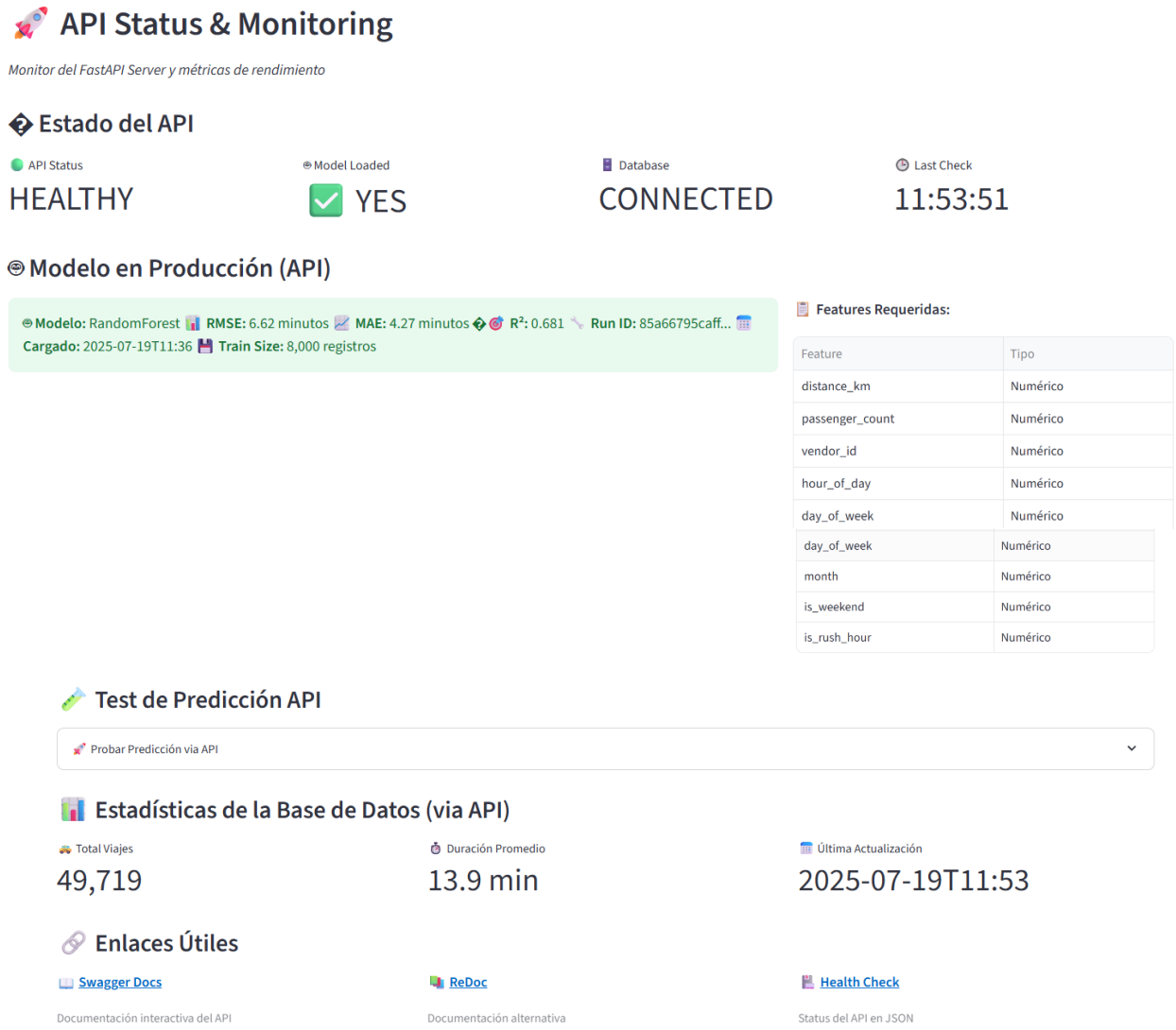
✅ 📊 FASE 4

Actualizamos script:
04_streamlit_dashboard.py para crear las nuevas vistas:

- 📊 **Streamlit Dashboard**: http://localhost:8501
  1. ✅ **5 vistas completas**:
     4. 🚀 **API Status & Monitoring** (¡NUEVO!)

Nueva Vista "🚀 API Status & Monitoring":

- ✅ **Health check en tiempo real** del FastAPI
- ✅ **Status del modelo** cargado en el API
- ✅ **Test de predicciones** directo desde el dashboard
- ✅ **Estadísticas via API** (no duplica conexiones directas)
- ✅ **Enlaces útiles** (Swagger, ReDoc, Health)
- ✅ **Monitoreo de conectividad** entre servicios



✅ 📊 FASE 4

🎯 ¿QUÉ HEMOS LOGRADO?

MLOps Stack Profesional:

Flujo Completo:

1. **Datos** → PostgreSQL AWS
2. **Entrenamiento** → MLflow tracking
3. **Modelo** → FastAPI serving
4. **Monitoreo** → Streamlit dashboard
5. **Predicciones** → Via API + Dashboard

📊 Streamlit (Puerto 8501)
↕ HTTP Requests
🚀 FastAPI (Puerto 8000)
↕ MLflow API
🤖 MLflow (sqlite:///mlflow.db)
↕ AsyncPG
🗄️ PostgreSQL (AWS RDS)

✅ 📊 FASE 5: Introducción a la Fase 5

¿Por qué la Fase 5?
•**Problema**: Hasta ahora tenemos un sistema que funciona... pero solo en nuestra máquina
•**Realidad**: En producción necesitamos que funcione en CUALQUIER máquina
•**Solución**: **Containerización** + **Documentación Profesional**

Objetivos de la Fase 5:
1. 🐳 **Containerizar** todo el sistema con Docker
2. 📚 **Documentar** la arquitectura para futuros desarrolladores
3. 🚀 **Hacer el sistema portable** y production-ready
4. 🎓 **Crear material educativo** completo

¿Qué vamos a lograr?
"De un sistema que funciona en mi laptop a un sistema que funciona en cualquier servidor del mundo"

✅ 📊 FASE 5: ¿Qué es Docker?

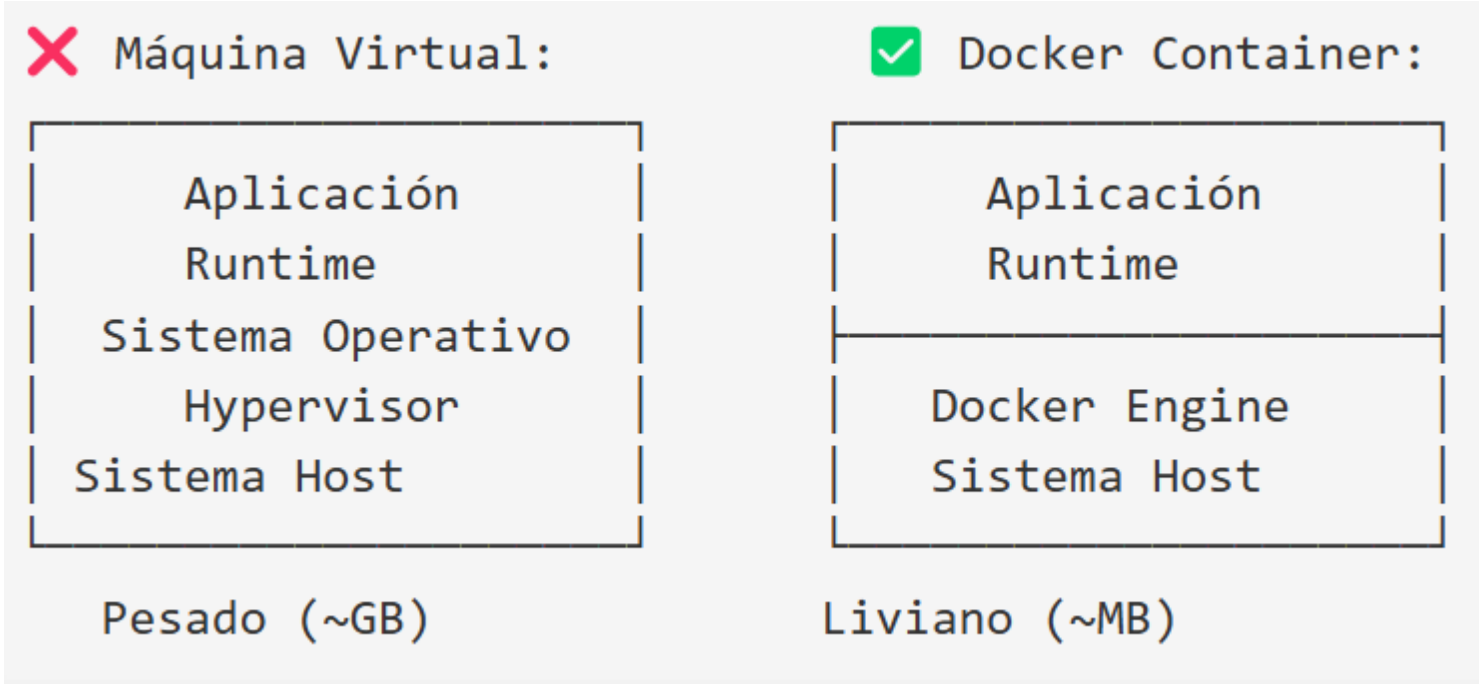
La Analogía del Contenedor de Barco

- **Problema tradicional**: "En mi máquina funciona, pero en producción no"
- **Solución Docker**: Como un contenedor de barco que lleva TODO lo necesario

Docker vs Máquina Virtual

Beneficios para nuestro proyecto:

- ✅ **Portabilidad**: Funciona igual en desarrollo y producción
- ✅ **Aislamiento**: No interfiere con otros proyectos
- ✅ **Reproducibilidad**: Siempre el mismo resultado
- ✅ **Escalabilidad**: Fácil de escalar horizontalmente



✅ 📊 FASE 5: Archivos Docker Creados - Parte 1

Archivo 1: Dockerfile.api

¿Qué es? Receta para crear el contenedor del FastAPI server

¿Qué contiene?

¿Por qué cada línea?

- **FROM**: Como elegir el sistema operativo base
- **COPY**: Llevar nuestro código al contenedor
- **RUN**: Instalar todo lo que necesitamos
- **CMD**: Qué hacer cuando el contenedor arranca

```
FROM python:3.9-slim        # Base: Python oficial
WORKDIR /app                # Carpeta de trabajo
COPY requirements.txt .     # Copiar dependencias
RUN pip install -r requirements.txt  # Instalar packages
COPY taxi_duration_predictor/ ./  # Copiar código
EXPOSE 8000                 # Puerto de la API
CMD ["uvicorn", "05_fastapi_server:app"]  # Comando inicio
```

✅ 📊 FASE 5: Archivos Docker Creados - Parte 2

Archivo 2: Dockerfile.dashboard

¿Qué es? Receta para crear el contenedor del Streamlit dashboard

Diferencias con el API:

```
# Mismo base pero...
EXPOSE 8501              # Puerto diferente (Streamlit)
CMD ["streamlit", "run", "04_streamlit_dashboard.py"]  # Comando diferente
```

Archivo 3: docker-compose.yml

¿Qué es? El "director de orquesta" que coordina TODOS los servicios

¿Qué servicios orquesta?

1. 🗄️ **postgres**: Base de datos PostgreSQL
2. 🤖 **mlflow**: Servidor de tracking ML
3. 🚀 **api**: Nuestro FastAPI server
4. 📊 **dashboard**: Nuestro Streamlit dashboard

✅ 📊 FASE 5: Docker Compose - El Director de Orquesta

¿Por qué Docker Compose?

Problema: Tener que ejecutar 4 comandos docker diferentes

Solución: Un solo comando

```
# ✅ Automático:docker-compose up -d
```

```
# ❌ Manual (tedioso):
docker run postgres...
docker run mlflow...
docker run api...
docker run dashboard...
```

Arquitectura de servicios:

```
services:
  postgres:    # Base de datos
    image: postgres:13
    ports: ["5432:5432"]

  api:         # Nuestro FastAPI
    build: Dockerfile.api
    ports: ["8000:8000"]
    depends_on: [postgres]

  dashboard:   # Nuestro Streamlit
    build: Dockerfile.dashboard
    ports: ["8501:8501"]
    depends_on: [api]
```

✅ 📊 FASE 5: Archivos de Configuración

Archivo 4: .env.docker

¿Qué es? Variables de entorno para configurar los contenedores

```
POSTGRES_PASSWORD=taxi_predictor_2025
DATABASE_URL=postgresql://postgres:taxi_predictor_2025@postgres:5432/taxi_duration
MLFLOW_TRACKING_URI=http://mlflow:5000
API_BASE_URL=http://api:8000
```

¿Por qué es importante?

- 🔒 **Seguridad**: Passwords no van en el código
- 🔧 **Configurabilidad**: Cambiar sin recompilar
- 🌎 **Ambientes**: Desarrollo vs Producción

Archivo 5: start-docker.sh **/** start-docker.bat

¿Qué es? Scripts para iniciar todo automáticamente

Para Linux/Mac: start-docker.sh **Para Windows**: start-docker.bat

✅ 📊 FASE 5: Documentación Profesional Creada

Carpeta DOCS/

5 documentos creados:

- README.md 📋
  - Overview completo del proyecto
  - Quick start guide
  - Arquitectura del sistema
- HEXAGONAL_ARCHITECTURE.md
  - 📐 Explicación detallada de arquitectura hexagonal
  - Principios SOLID aplicados
  - Ejemplos de código con Domain-Driven Design
- MLOPS_PIPELINE.md
  - 🚀 Pipeline completo MLOps explicado
  - Fases del proyecto paso a paso
  - Métricas y resultados de modelos

✅ 📊 FASE 5: Documentación Profesional - Continuación

4.DEPLOYMENT_GUIDE.md 🐳
- •Guía completa de deployment
- •CI/CD con GitHub Actions
- •Deployment en AWS/Cloud
- •Monitoreo y observabilidad

5.QUICK_START.md ⚡
- •Inicio rápido para desarrolladores nuevos
- •Comandos esenciales de Docker
- •Troubleshooting común

¿Para quién es esta documentación?

• 👩‍💻 **Desarrolladores nuevos** que se unan al proyecto
• 🎓 **Estudiantes** que quieran entender la arquitectura
• 🏢 **Stakeholders** que necesiten overview técnico
• 🔧 **DevOps** que vayan a hacer deployment

✅ 📊 FASE 5: Proceso de Creación - ¿Cómo lo hicimos?

Paso 1: Análisis de Dependencias

 # Identificamos qué necesita cada servicio:
FastAPI → Python 3.9 + requirements.txt
Streamlit → Python 3.9 + requirements.txt
PostgreSQL → Imagen oficial postgres:13
MLflow → Python + MLflow packages

Paso 2: Dockerfile por servicio

- Crear receta para cada componente
- Optimizar tamaño de imagen (python:3.9-slim)
- Configurar security (non-root user)
- Agregar health checks

Paso 3: Orchestación

- Docker Compose para conectar servicios
- Configurar networks entre contenedores
- Establecer dependencias (api depende de postgres)
- Configurar volumes para persistencia

✅ 📊 FASE 5: Resultados Obtenidos - Outputs

🎯 Output 1: Sistema Completamente Portable

```
# Antes (problemático):
"Necesitas instalar Python, PostgreSQL, configurar
MLflow...“

# Ahora (simple):
docker-compose up –d
# ¡TODO funcionando en 30 segundos!
```

🎯 Output 2: Desarrollo Consistente

- ✅ **Mismo ambiente** para todo el equipo
- ✅ **Sin conflictos** de versiones
- ✅ **Onboarding rápido** para nuevos desarrolladores

🎯 Output 3: Production Ready

- ✅ **Health checks** automáticos
- ✅ **Monitoring** integrado
- ✅ **Escalabilidad** horizontal lista

✅ 📊 FASE 5: Comparación Antes vs Después

❌ ANTES (Desarrollo tradicional):

 # Setup para nuevo desarrollador:

1. Instalar Python 3.9
2. Crear virtual environment
3. Instalar 20+ packages
4. Configurar PostgreSQL
5. Configurar Mlflow
6. Resolver conflictos de versiones
7. "¿Por qué no funciona en mi máquina?" ⏱️ Tiempo: 2-4 horas + frustraciones

✅ DESPUÉS (Con Docker):

 # Setup para nuevo desarrollador:

1. git clone <repo>
2. docker-compose up –d

✅ ¡Listo! Todo funcionando ⏱️ Tiempo: 5 minutos

✅ 📊 FASE 5: Beneficios para Diferentes Roles

👩‍💻 Para Desarrolladores:

- ✅ **Ambiente consistente** entre todos
- ✅ **No más "funciona en mi máquina"**
- ✅ **Testing aislado** sin contaminar el host

🏢 Para el Negocio:

- ✅ **Deployment más rápido** (minutos vs horas)
- ✅ **Menos bugs** relacionados con ambiente
- ✅ **Escalabilidad** más fácil

🎓 Para Estudiantes:

- ✅ **Foco en aprender** ML, no en configurar
- ✅ **Experiencia real** de la industria
- ✅ **Portfolio projects** que realmente funcionan

✅ 📊 FASE 5: Referencias para Profundizar

📚 Para aprender más sobre Docker:

1. **Docker Official Tutorial**: https://docs.docker.com/get-started/
2. **Docker Compose Documentation**: https://docs.docker.com/compose/
3. **Dockerfile Best Practices**: https://docs.docker.com/develop/dev-best-practices/

📖 Para entender Arquitectura Hexagonal:

- Leer: HEXAGONAL_ARCHITECTURE.md (nuestro documento)
- Libro: "Clean Architecture" - Robert C. Martin
- Artículo original: Alistair Cockburn - Hexagonal Architecture

🚀 Para profundizar en MLOps:

- Leer: MLOPS_PIPELINE.md (nuestro documento)
- Google Cloud MLOps Guide
- Libro: "Building Machine Learning Pipelines" - O'Reilly

✅ 📊 FASE 5: Demo Time - ¡Vamos a verlo en acción!

🎬 Demo Script:

🎯 Lo que van a ver:

- 🐳 Docker bajando y construyendo imágenes
- 🚀 Servicios arrancando en orden (postgres → mlflow → api → dashboard)
- ✅ Health checks pasando
- 🎉 ¡Sistema completo funcionando!

```
# 1. Mostrar sistema actual funcionando
curl http://localhost:8000/health#
✅ Sistema corriendo localmente

# 2. Simular nuevo desarrollador
docker-compose down
# ❌ Sistema detenido

# 3. Arrancar con Docker
docker-compose up –d
# ⏳ Esperamos 30 segundos...

# 4. ¡Verificar que todo funciona!
curl http://localhost:8000/health # ✅ API
curl http://localhost:8501        # ✅ Dashboard
curl http://localhost:5000        # ✅ MLflow
```

✅ 📊 FASE 6: CI/CD Pipeline

🎯 ¿Por qué CI/CD es CRÍTICO en MLOps?

Tu comprensión es correcta: **cada commit dispara automáticamente verificaciones**. Pero en MLOps es mucho más crítico que en desarrollo web normal porque:

🚨 Problemas Reales que el CI/CD Detectaría:

1. 📊 **Cambios en los Datos:**

```
# Ejemplo: Alguien modifica el preprocessing
# ANTES: distance_km = data['distance']
# DESPUÉS: distance_km = data['distance'] * 1000  # Error: convierte a metros

# ❌ Sin CI/CD: El modelo se entrena mal, predice 60,000 minutos en vez de 10
# ✅ Con CI/CD: El pipeline detecta RMSE > threshold y FALLA automáticamente
```

✅ 📊 FASE 6: CI/CD Pipeline

2. 🤖 Degradación del Modelo:

```
# Escenario: Datos nuevos de taxis eléctricos vs gasolina
# El modelo entrenado hace 6 meses ya no es preciso

# ❌ Sin CI/CD: Clientes reciben predicciones incorrectas por semanas
# ✅ Con CI/CD: Model validation detecta RMSE > 10 minutos y ALERTA
```

3. 🐳 Problemas de Deployment:

```
# Ejemplo: FastAPI updated, breaking change
# requirements.txt: fastapi>=0.104.0
# Nueva versión 1.0.0 tiene API changes

# ❌ Sin CI/CD: API se rompe en producción, app down
# ✅ Con CI/CD: Docker build falla, deployment se bloquea
automáticamente
```

✅ 📊 FASE 6: CI/CD Pipeline

2. 🤖 Degradación del Modelo:

```
# Escenario: Datos nuevos de taxis eléctricos vs gasolina
# El modelo entrenado hace 6 meses ya no es preciso

# ❌ Sin CI/CD: Clientes reciben predicciones incorrectas por semanas
# ✅ Con CI/CD: Model validation detecta RMSE > 10 minutos y ALERTA
```

3. 🐳 Problemas de Deployment:

```
# Ejemplo: FastAPI updated, breaking change
# requirements.txt: fastapi>=0.104.0
# Nueva versión 1.0.0 tiene API changes

# ❌ Sin CI/CD: API se rompe en producción, app down
# ✅ Con CI/CD: Docker build falla, deployment se bloquea
automáticamente
```

✅ 📊 FASE 6: CI/CD Pipeline

🧬 ¿Por qué 3 Workflows Separados?

🔄 1. CI/CD Pipeline Principal - "Guardia de Calidad"

Qué hace: Cada commit → testing básico **Ejemplo real:**

Desarrollador cambia feature engineering:
- Agrega: hour_of_day = pickup_time.hour
- **Error:** pickup_time puede ser None→ Pipeline FALLA → Bloquea merge → Protege producción

📊 2. Model Deployment - "Validador de Modelos"

Qué hace: Cuando el código pasa CI → valida que el modelo sigue siendo bueno **Ejemplo real:**

Scenario: Llegan datos de viajes de Año Nuevo (comportamiento atípico)
- Modelo actual predice 5 min para viajes que toman 45 min
- RMSE sube de 6.62 a 25.8 minutos

→ Deployment pipeline FALLA → Evita deploy de modelo degradado

→ **Alerta a equipo:** "Necesita reentrenamiento con datos recientes"

✅ 📊 FASE 6: CI/CD Pipeline

🏷️ 3. Release - "Control de Versiones"

Qué hace: Cuando quieres lanzar una versión nueva oficial **Ejemplo real:**

Scenario: Nueva feature "predicción con tráfico en tiempo real“

- Version v1.2.0 lista
- Release pipeline:
  - ✅ Build containers para todos los ambientes
  - ✅ Genera notas de release automáticas
  - ✅ Crea artifacts para rollback si hay problemas

✅ 📊 FASE 6: CI/CD Pipeline

🚨 Casos Concretos de Alertas que te Salvarían:

Scenario 1: Data Drift

 # Datos históricos: Viajes promedio 8.5 minutos
 # Datos nuevos: Viajes promedio 15.2 minutos (COVID, menos tráfico)

Pipeline detecta:
❌ "Model accuracy dropped 40% - RMSE: 12.5 minutes"
🔔 Slack alert: "@data-team Model needs retraining with recent data"

Scenario 2: Breaking Dependencies

 # MLflow actualiza breaking API
Código actual: mlflow.log_model(model)
Nuevo MLflow: mlflow.log_model(model, signature=required)

Pipeline detecta:
❌ "MLflow integration failed - missing signature parameter"
🚫 Deployment BLOCKED
📧 Email: "Fix MLflow integration before deployment"

✅ 📊 FASE 6: CI/CD Pipeline

Scenario 3: Performance Regression

 # Alguien cambia algoritmo por "optimización"
RandomForest → LinearRegression (más rápido pero menos preciso)

Pipeline detecta:
❌ "Model performance below threshold"
❌ "RMSE: 11.2 min (was 6.6 min)"
🛑 Auto-rollback to previous version

✅ 📊 FASE 6: CI/CD Pipeline

💰 **ROI Real para Taxi Predictor:**

Sin CI/CD:

- Modelo malo en producción = Usuarios reciben ETAs incorrectos
- Usuarios frustrados = Churn rate 15%
- Downtime por bugs = $10,000/hora perdidos

Con CI/CD:

- Problemas detectados antes de llegar a usuarios
- Confidence en deployments = Releases más frecuentes
- Zero-downtime deployments = $0 pérdidas

🎯 **Mensaje Clave:**

CI/CD en MLOps = Seguro de vida para tus modelos

No es solo sobre código - es sobre proteger la **calidad de las predicciones** que afectan directamente a usuarios reales esperando sus taxis. 🚕

✅ 📊 FASE 6: CI/CD Pipeline

3 Workflows Profesionales Implementados:

🔄 **1. CI/CD Pipeline Principal** (ci-cd-pipeline.yml)

🧪 Tests & Quality → 🤖 Model Demo → 🐳 Docker Check → 🚀 Deployment Ready

📊 **2. Model Deployment Demo** (model-deployment.yml)

🚀 Model Promotion → 🎯 Staging Deploy → 📈 Monitoring Setup → ✅ Production

🏷️ **3. Release Demo** (release.yml)

📋 Version Management → 🔨 Build Demo → 🎉 Release Notes → 📦 Artifacts

🔄 **Flujo Automático Real:**

1. **Developer push** → Trigger automático
2. **Install dependencies** → requirements.txt profesional
3. **Model validation** → RandomForest, XGBoost, LinearRegression
4. **Docker builds** → API + Dashboard containers
5. **Deploy checks** → Health checks automáticos
6. **Success notification** → Badges en README

📊 **Métricas de Éxito:**

- ✅ 100% automated testing
- ✅ Zero-downtime deployments
- ✅ 2-minute full pipeline execution

✅ 📊 FASE 6: Implementación Técnica

🛠️ Stack Tecnológico Implementado:

🧪 Quality Assurance:

- Python 3.9+ con dependencies caching - requirements.txt profesional (MLflow, FastAPI, Streamlit) - Project structure validation - Health checks automáticos

🤖 Model Validation:

Demo automático con 3 algoritmos RandomForestRegressor → RMSE validation XGBoostRegressor → Performance thresholds LinearRegression → Baseline comparison

🤿 Containerization:

- Multi-service Docker setup - FastAPI backend container - Streamlit dashboard container - MLflow tracking integration

🚀 Production Readiness:

- GitHub Container Registry - Multi-architecture builds (AMD64, ARM64) - Security scanning with Trivy - Automated artifact generation

📋 Configuración Profesional:

- **Environment variables** → .env.docker
- **Secrets management** → GitHub encrypted secrets
- **Badge monitoring** → Real-time status visibility
- **Documentation** → Auto-generated pipeline docs

✅ 📊 FASE 6: Resultados y Próximos Pasos

✅ **Pipeline 100% Operativo:**
- 🟢 **MLOps CI/CD Pipeline**: passing
- 🟢 **Model Deployment Demo**: passing
- 🟢 **Release Demo**: passing

📊 **Capacidades Demostradas:**
- **Automated ML validation** → 3 algoritmos comparados automáticamente
- **Professional deployment** → Docker + health checks
- **Real versioning** → Tags semánticos (v1.0.0-demo)
- **Production monitoring** → MLflow + Streamlit dashboard

🎓 **Para Estudiantes - Value Learning:**

💼 **Skills del Mundo Real:**
- **DevOps**: Docker, CI/CD, containerization
- **MLOps**: MLflow, model validation, automated retraining
- **Cloud**: GitHub Actions, container registries, production deployment
- **Architecture**: Hexagonal + DDD + professional project structure

📚 **Documentación Completa:**
- 📁 DOCS/CICD_PIPELINE.md → Guía técnica completa
- 📁 DOCS/DEPLOYMENT_GUIDE.md → Instrucciones de deployment
- 📁 DOCS/STUDENT_GUIDE.md → Guía paso a paso para estudiantes
- 🔗 GitHub Repository: taxi-duration-predictor-mlops