

EKS KUBERNETES

Sergio Morales
Armanjot Singh
Diego Miguel
Hans Jeremi Gonzalez

Introducción a EKS

Amazon Elastic Kubernetes Service (EKS) es un servicio de Kubernetes completamente administrado por AWS.

El principal objetivo es eliminar la complejidad operativa de crear, proteger y mantener clústeres de Kubernetes (más rentable que mantener centros de datos propios).

Las principales ventajas son:

- Velocidad: Implementación de aplicaciones más rápida y con menos carga operativa.
- Escalabilidad: Adaptación sin interrupciones a los picos de demanda de las cargas de trabajo.
- Seguridad: Integración nativa con AWS y actualizaciones automatizadas.



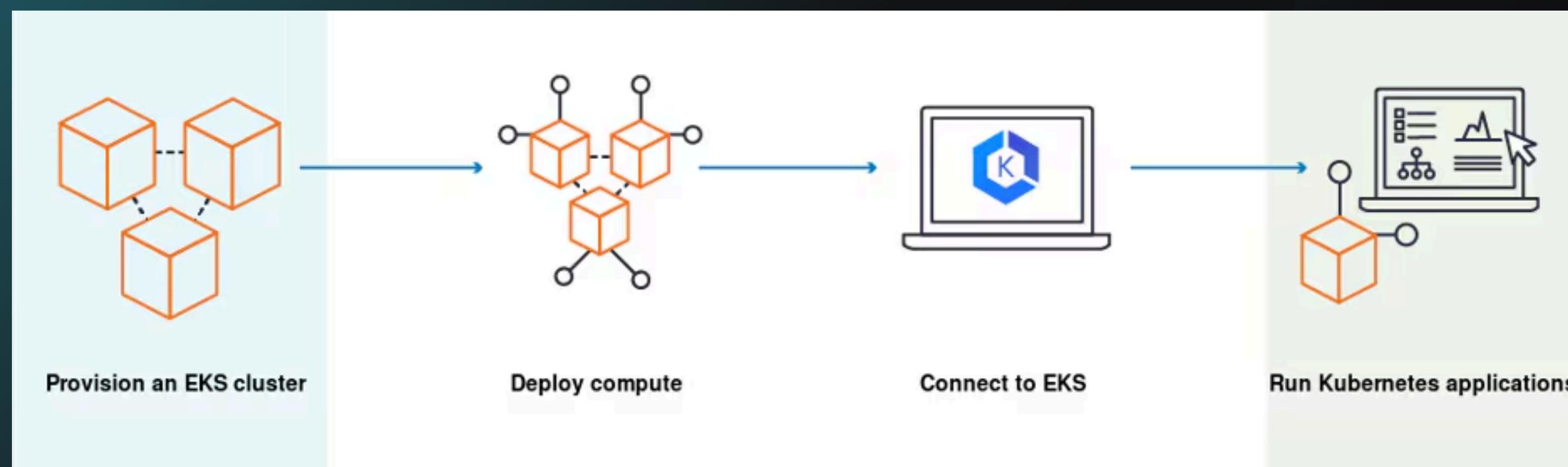
Arquitectura y Responsabilidad Compartida en EKS

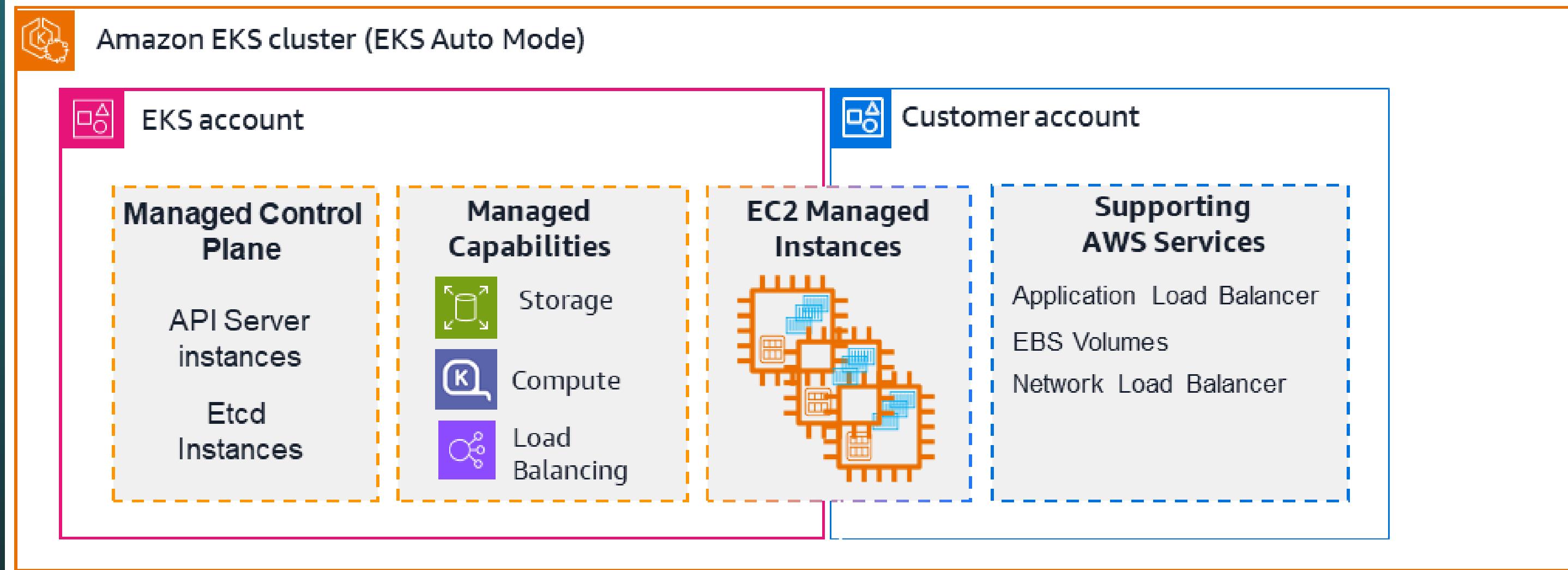
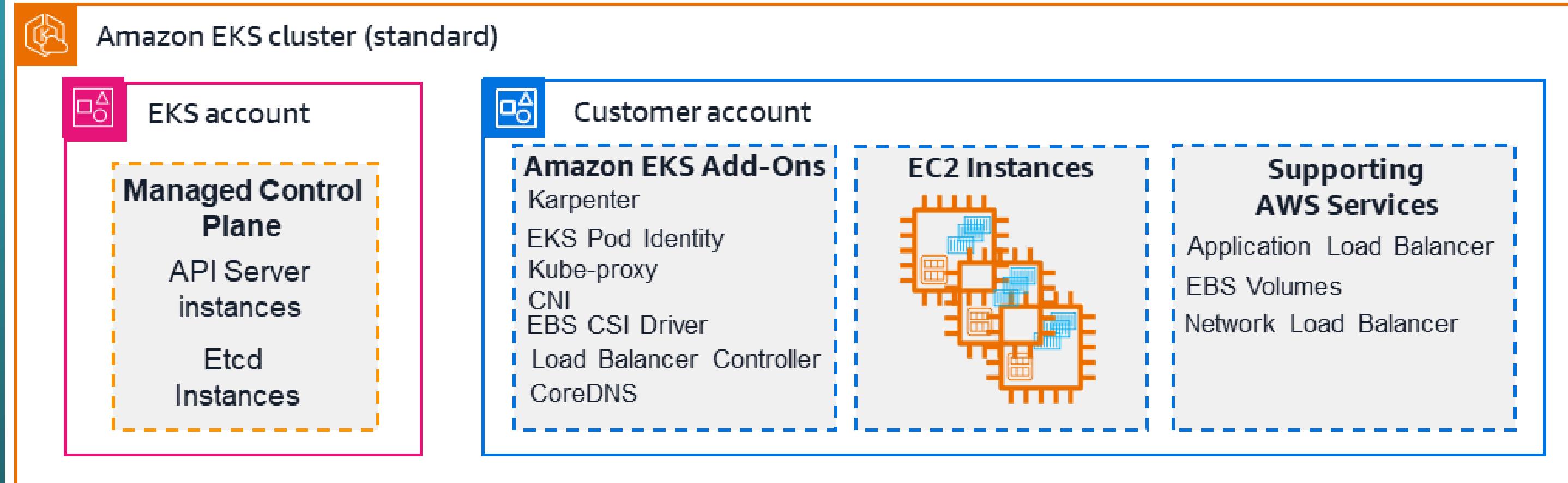
Control Plane (Gestionado por AWS):

- Aprovisionamiento y escalado automatizado de instancias del API Server y el clúster de clave-valor .
- Despliegue distribuido en múltiples Zonas de Disponibilidad (AZs) para garantizar Alta Disponibilidad (HA) y tolerancia a fallos.

Data Plane / Worker Nodes (Gestionado por el Cliente):

- Administración de instancias Amazon EC2 donde se ejecutan los contenedores (Pods).
- Uso de Managed Node Groups (Grupos de Nodos Administrados) para simplificar el aprovisionamiento, la capacidad de cómputo y las políticas de autoescalado desde la consola de AWS.





Arquitectura: WordPress + MySQL con almacenamiento persistente

- Datos a salvo en AWS: Usamos discos persistentes para no perder información si un contenedor se apaga o reinicia.
- App y Base de Datos separadas: Mejora la seguridad y hace que el sistema sea más estable.
- Requisito clave: Hay que instalar el EBS CSI Driver en EKS para que AWS nos deje crear los discos
- Contraseñas seguras: Usamos Secrets de Kubernetes para que las contraseñas (como la del root) estén encriptadas.
- Almacenamiento (PVCs): Pedimos dos discos de 10 GB (uno para los archivos de WordPress y otro para la base de datos MySQL).
- Red protegida: La base de datos (MySQL) se queda aislada en una red interna para que nadie pueda acceder desde fuera.

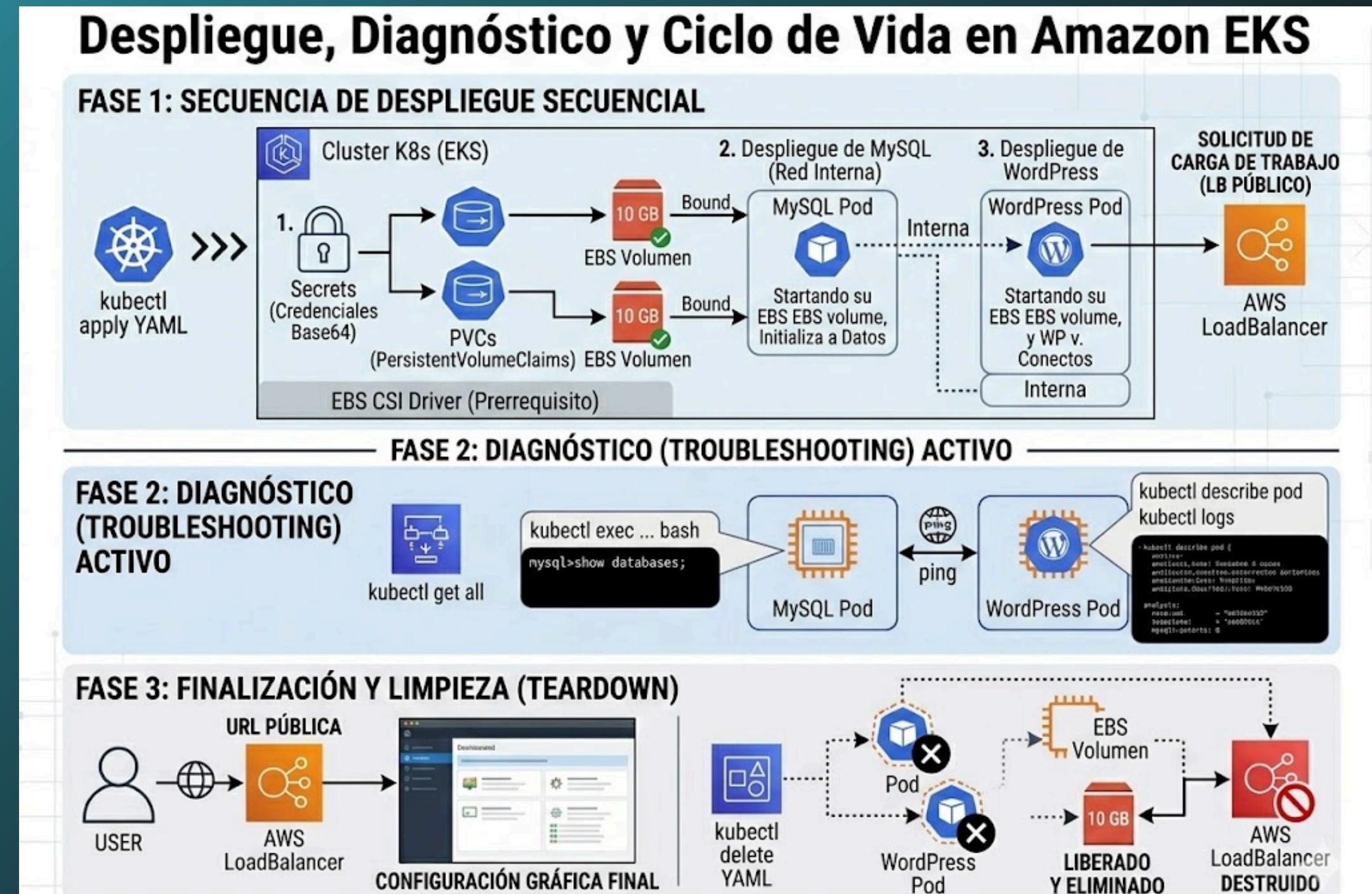


Despliegue, Diagnóstico y Ciclo de Vida

El despliegue requiere un orden estricto para resolver dependencias: primero levantar infraestructura, luego validar que todo funciona y, al final, hacer teardown para optimizar costes en la nube.

- 1. Secuencia de despliegue
 - Secrets + PVCs: inyectar credenciales (Base64) y crear PVCs para aprovisionar discos EBS (quedan en Pending hasta que se asignan).
 - MySQL + WordPress: desplegar la BD en red interna y después la app, exponiéndola con LoadBalancer hacia Amazon Web Services.
- 2. Diagnóstico (Troubleshooting)
 - Verificación de recursos y almacenamiento: kubectl get all, kubectl get pvc.
 - Acceso interactivo (bash) + revisión de eventos/logs: kubectl describe, kubectl logs para validar red y detectar fallos.
- 3. Finalización y limpieza (Teardown)
 - Configuración final accediendo a la URL pública del balanceador.
 - Borrar manifests para destruir recursos, liberar volúmenes y evitar facturación innecesaria.

Despliegue, Diagnóstico y Ciclo de Vida



Estado deseado y Autohealing: Deployment + ReplicaSet

- Kubernetes trabaja en modo declarativo: tú defines el estado deseado y el clúster lo mantiene.
- Deployment gestiona el ciclo de vida de la app: réplicas, actualizaciones y rollback.
- ReplicaSet asegura que siempre exista el número de Pods indicado (si cae uno, se recrea)
- Extra “pro”: permite rolling updates (actualizar sin cortar servicio) y rollback si algo falla.
- Frase para decir: “Esto demuestra alta disponibilidad real: el sistema se repara solo y puedo actualizar sin downtime.”

Calidad de servicio: Probes y recursos (que “funcione bien”)

- Readiness probe: un Pod solo recibe tráfico cuando está listo (evita enviar usuarios a una app “a medio arrancar”).
- Liveness probe: reinicia contenedores que se quedan colgados (auto-recuperación ante fallos).
- Startup probe: para apps que tardan en arrancar, evita falsos reinicios al inicio.
- Requests/Limits de CPU/RAM: garantizan recursos mínimos y evitan que un Pod “se coma” el nodo.



Conectividad estable: nombres DNS y desacoplamiento

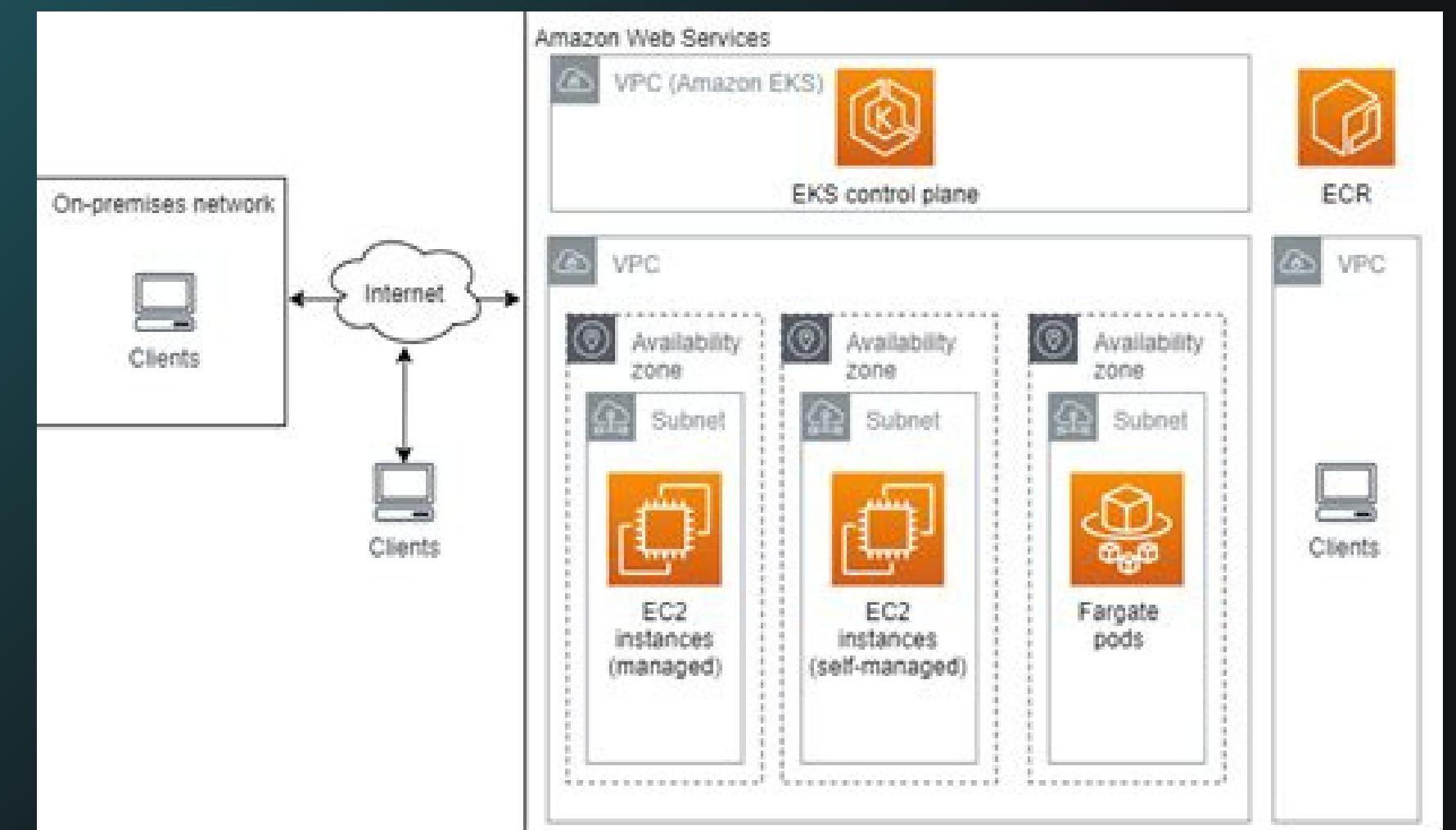
- Dentro del clúster, los componentes se descubren por nombre (DNS), no por direcciones “temporales”.
- CoreDNS permite resolver nombres internos de servicios y simplifica la comunicación entre capas.
- Ventaja clave: puedes reiniciar/recrear Pods y la app sigue conectando porque el punto de conexión lógico se mantiene.
- Esto también facilita documentación y troubleshooting: la arquitectura es predecible (siempre conectas al mismo nombre lógico).

Escalado y Observabilidad: pasar de “demo” a “operable”

- HPA (Horizontal Pod Autoscaler): escala réplicas automáticamente según CPU/RAM o métricas.
- Para métricas base, suele usarse Kubernetes Metrics Server.
- Escalado de nodos: con autoscaling del clúster, el sistema añade/quita capacidad cuando faltan recursos.
- Observabilidad: logs y métricas para detectar problemas antes de que los vea el usuario (por ejemplo con Amazon CloudWatch, Prometheus y Grafana).

HelloWorld en Kubernetes (K8s) dentro de EKS

- Se crea una imagen Docker muy simple que devuelve "Hello World".
- Esa imagen se sube a un repositorio.
- Kubernetes la ejecuta dentro del clúster EKS



Despliegue del HelloWorld en EKS

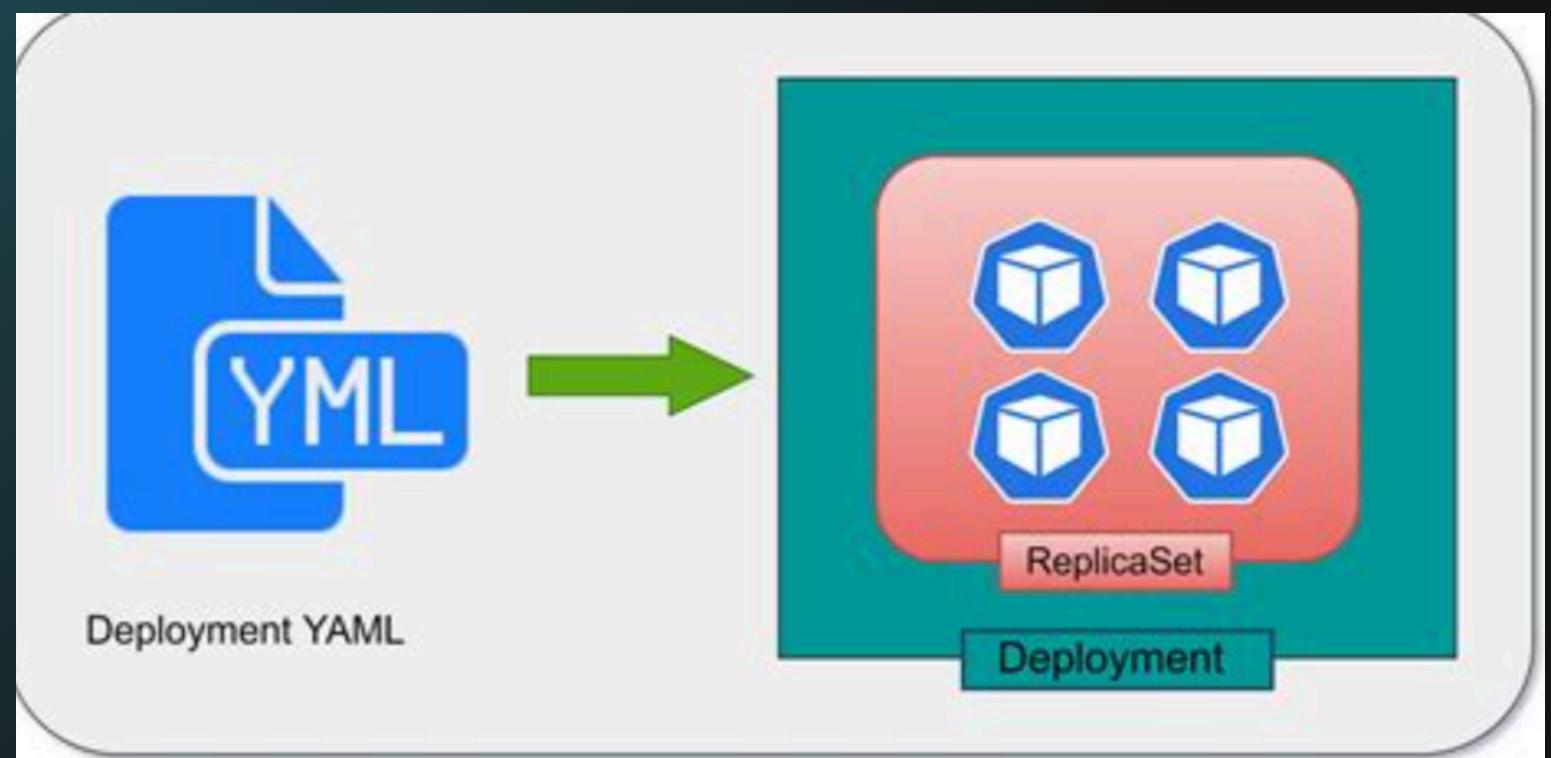
Despliegue del HelloWorld en EKS

- Se aplica un archivo YAML (de confirmación) que indica a K8s cuántas réplicas ejecutar.
- Kubernetes crea los pods (contenedores dentro de los nodos, como si fuesen dockers) automáticamente.
- Si un pod falla, K8s lo reemplaza sin intervención humana.

Despliegue del HelloWorld en EKS

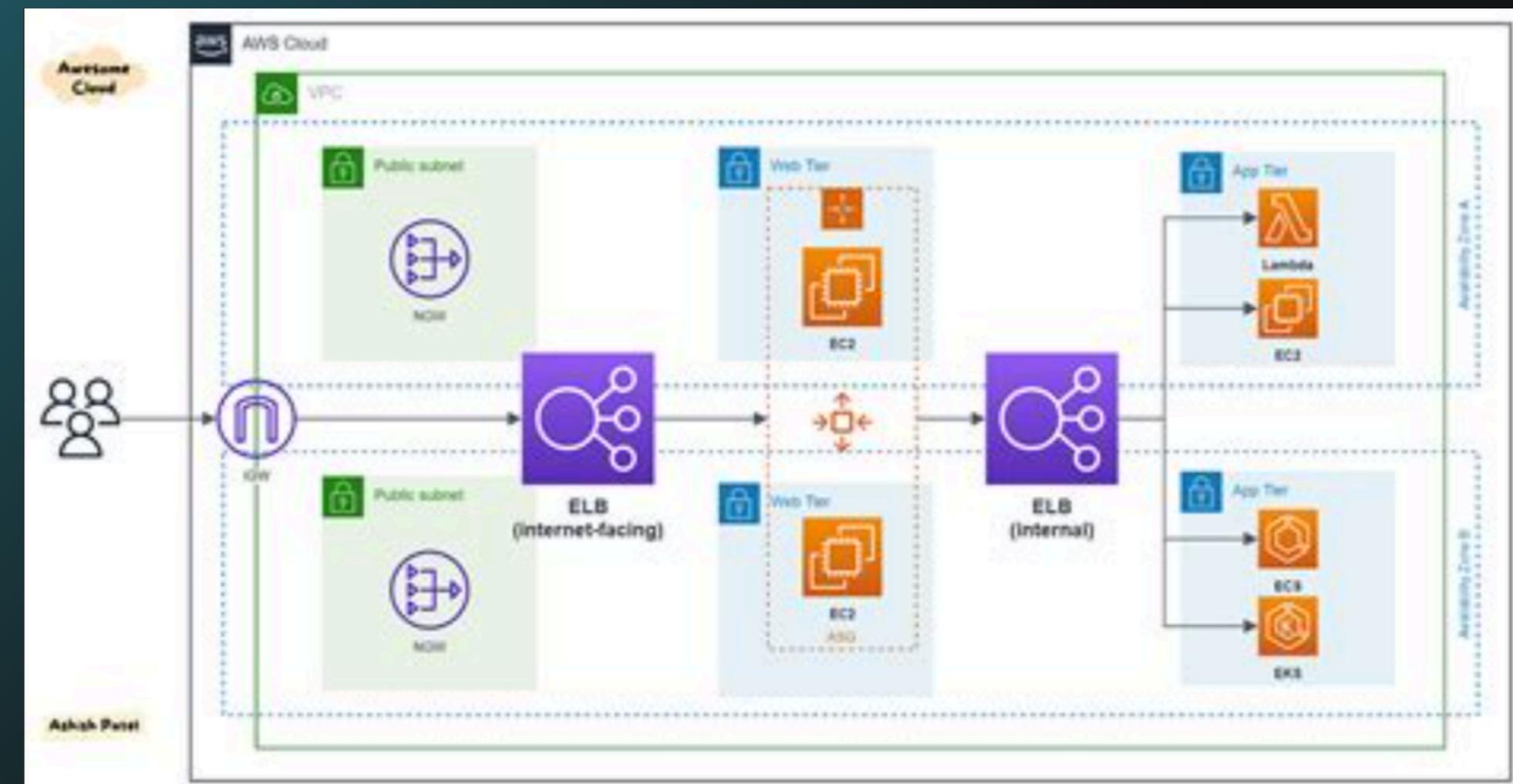
Despliegue del HelloWorld en EKS

- Se aplica un archivo YAML (de confirmación) que indica a K8s cuántas réplicas ejecutar.
- Kubernetes crea los pods (contenedores dentro de los nodos, como si fuesen dockers) automáticamente.
- Si un pod falla, K8s lo reemplaza sin intervención humana.



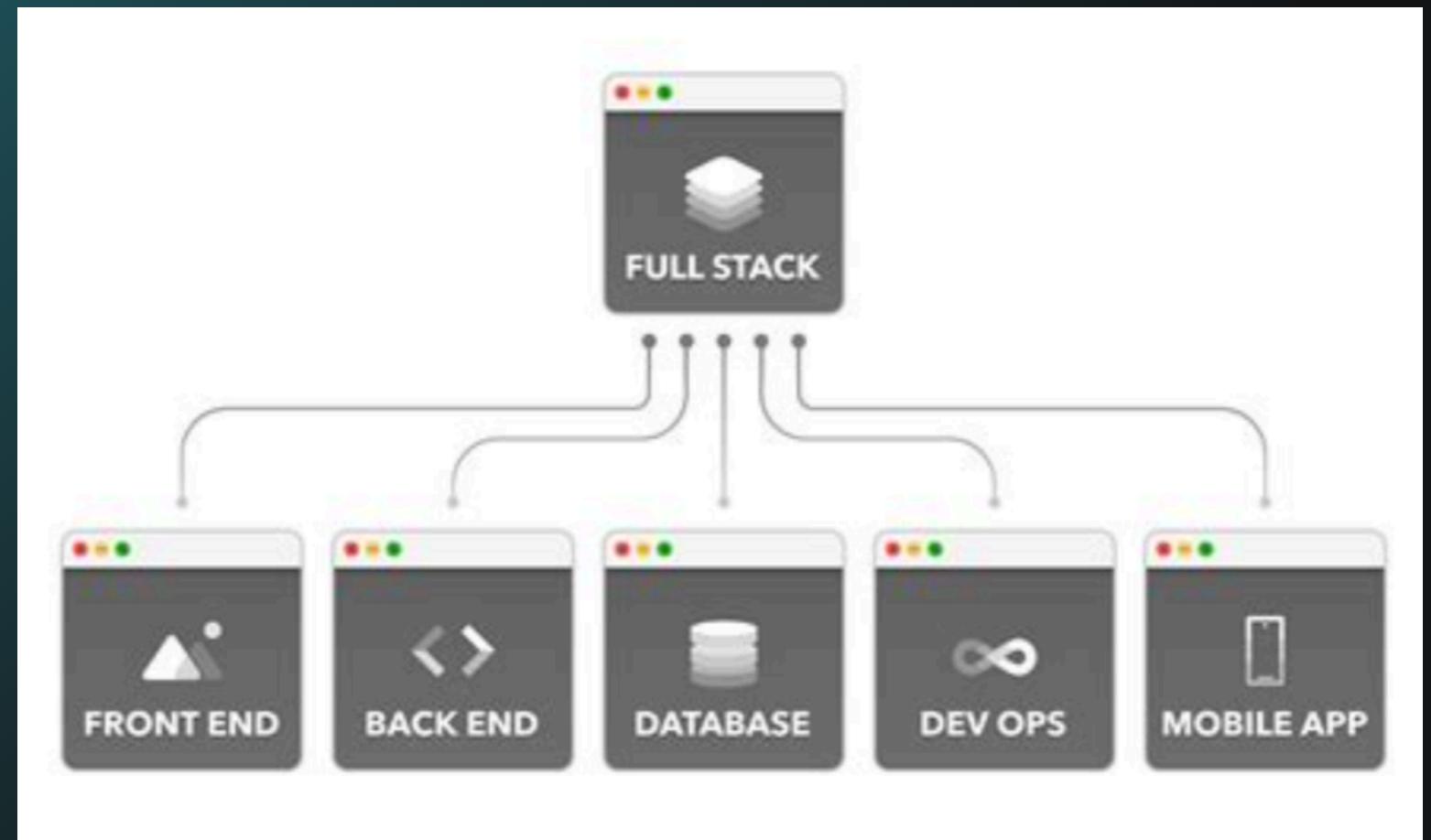
Acceso al HelloWorld desde fuera

- Se crea un Service tipo LoadBalancer.
- AWS genera una IP pública o URL para acceder al HelloWorld.
- El usuario abre el navegador y ve la respuesta “Hello World”.



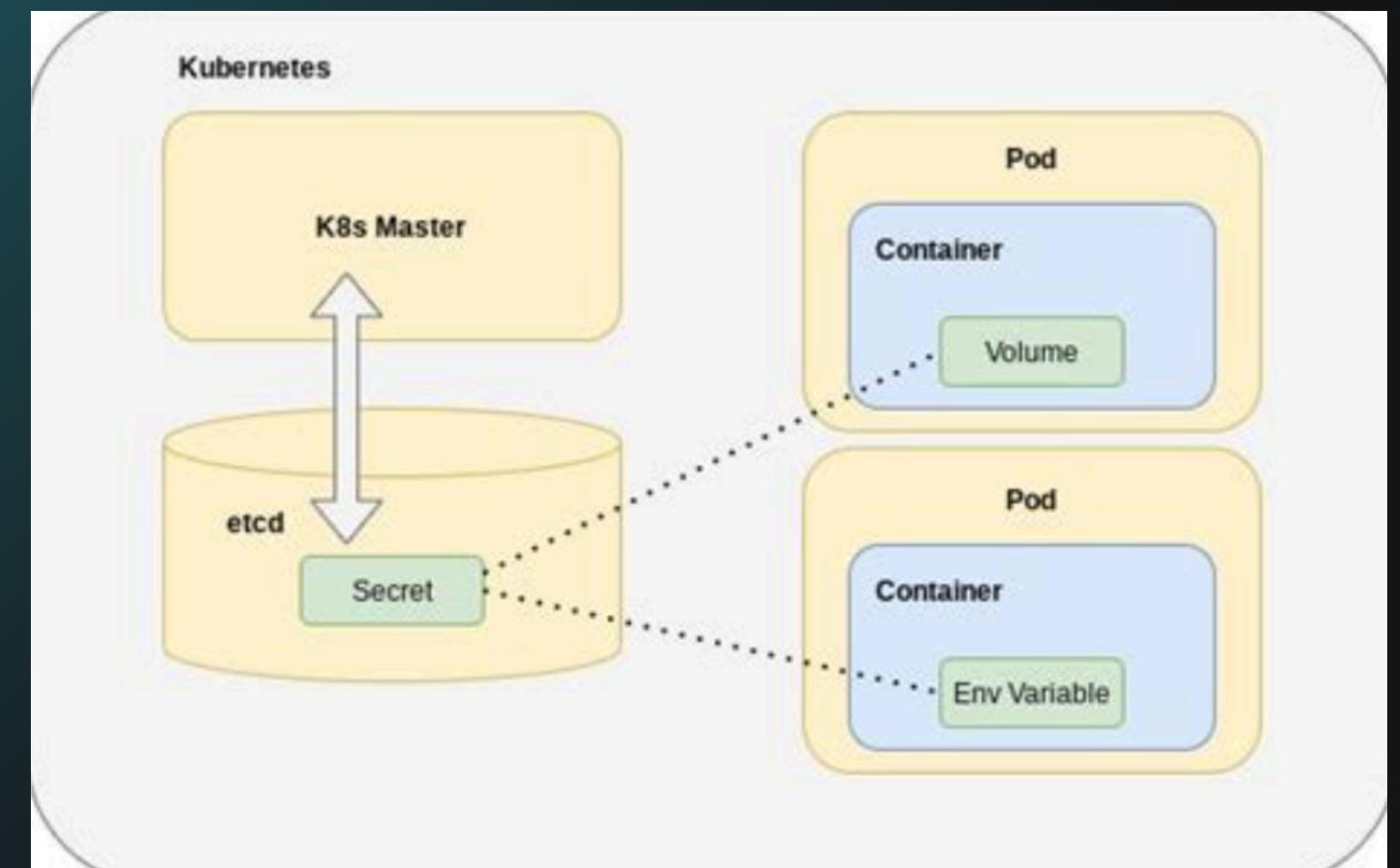
Arquitectura FullStack en EKS

- El frontend y el backend se ejecutan como contenedores dentro del clúster.
- Cada parte es un despliegue independiente.
- Kubernetes gestiona la comunicación interna entre servicios.



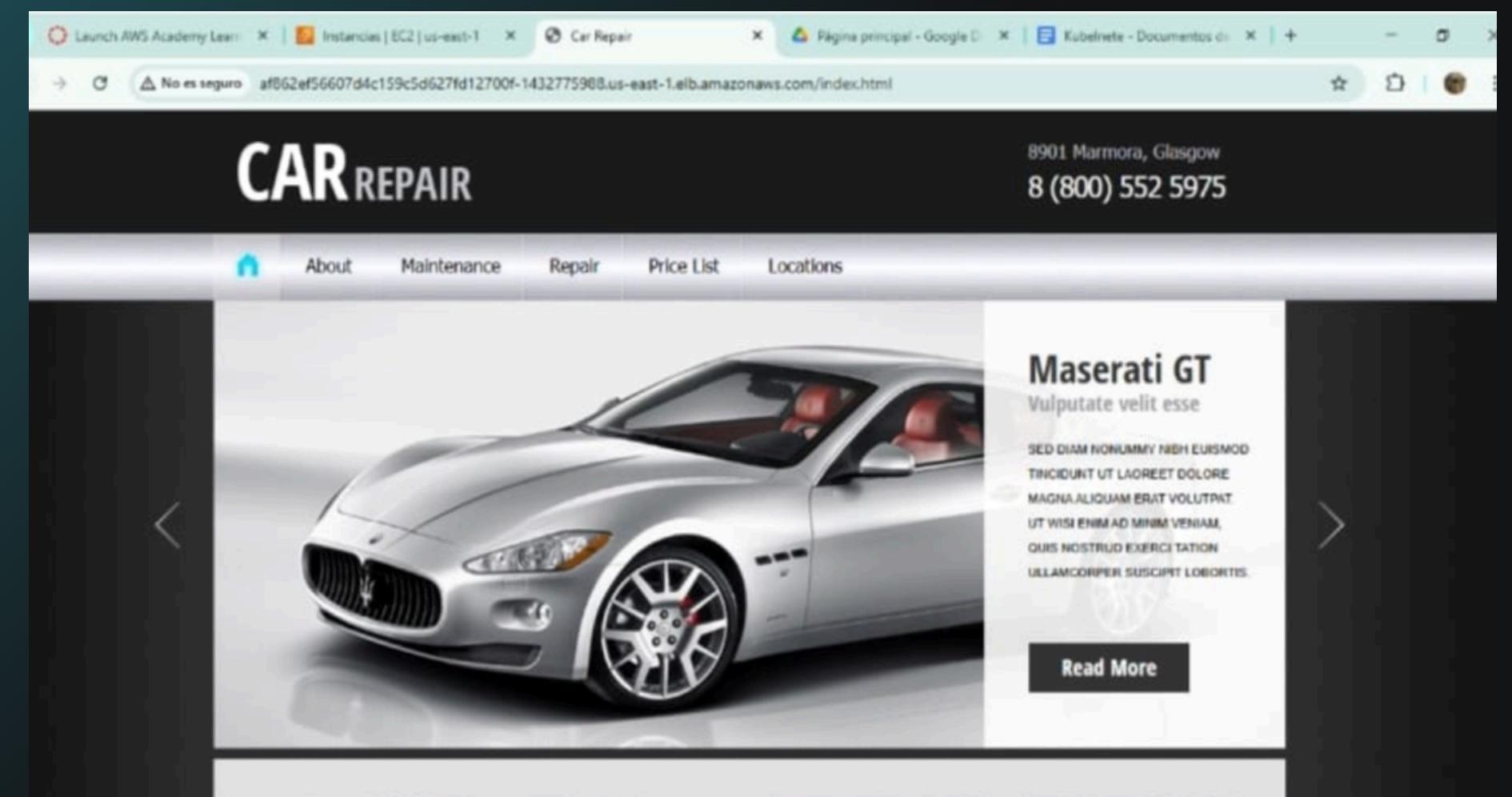
Conexión del backend con RDS

- El backend se conecta a una base de datos RDS mediante una URL interna.
- No se expone la base de datos a internet.
- Las credenciales se guardan en Secrets de Kubernetes.



Flujo completo de la aplicación FullStack

- El usuario accede al frontend. (Es decir, nuestra pagina web)
- El frontend llama al backend dentro del clúster.
- El backend consulta o guarda datos en RDS.
- Todo ocurre dentro de la red privada de AWS.

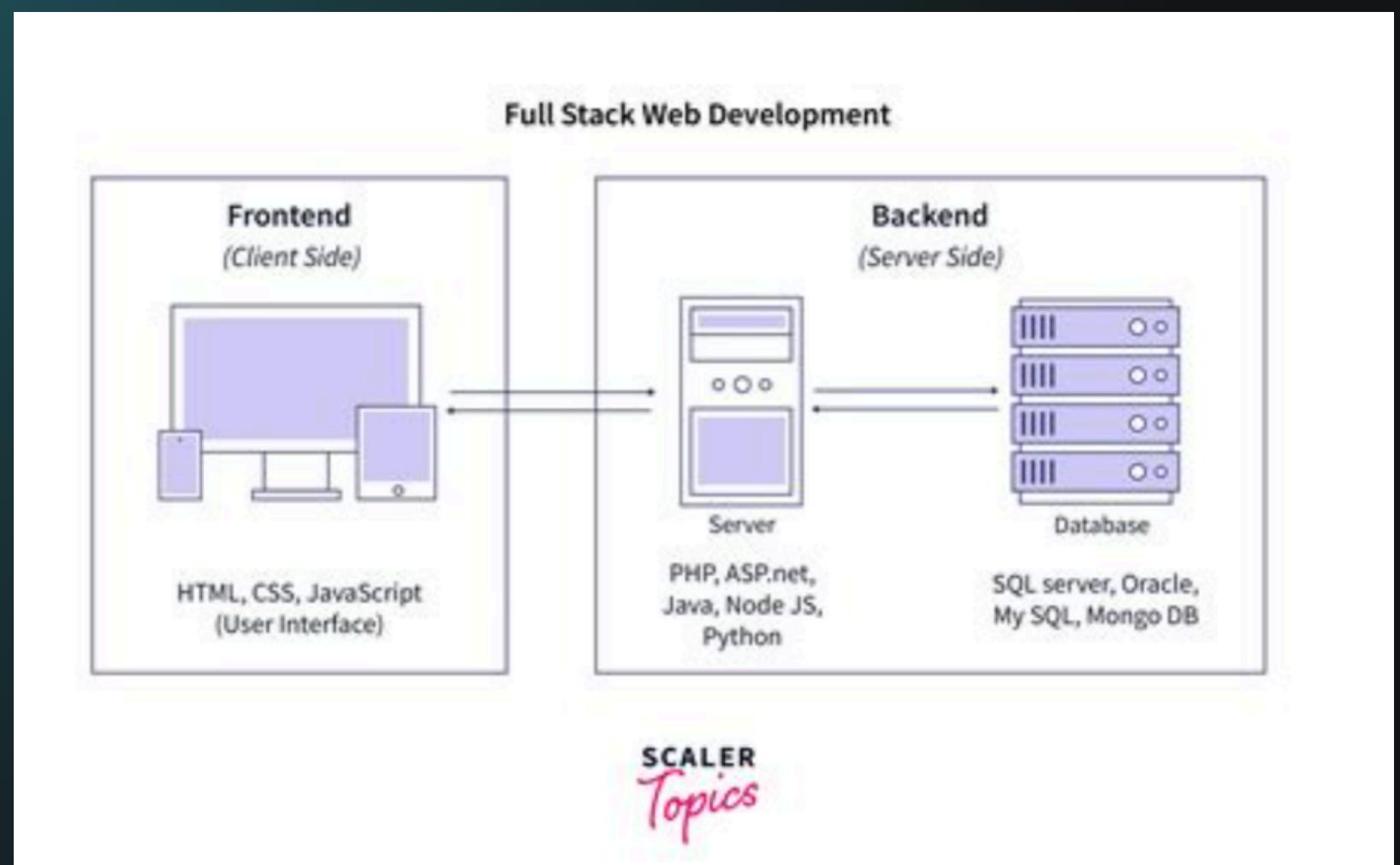


Ejemplo real: Aplicación de reservas (FullStack en EKS + RDS)

Una empresa quiere una app para que los clientes reserven citas online.

La arquitectura FullStack en EKS + RDS funcionaría así:

- Frontend: página web donde el usuario elige día y hora.
- Backend: recibe la reserva, valida disponibilidad y guarda los datos.
- RDS: almacena usuarios, horarios y reservas.
- EKS: ejecuta frontend y backend en contenedores, asegurando que siempre estén disponibles.



Para qué sirve realmente un FullStack en EKS + RDS

Este tipo de arquitectura se usa cuando una empresa necesita:

- Escalar automáticamente cuando hay muchos usuarios (por ejemplo, Black Friday).
- Evitar caídas: si un contenedor falla, Kubernetes crea otro.
- Separar responsabilidades: frontend, backend y base de datos están aislados.
- Actualizaciones sin interrupciones: se despliega una nueva versión sin parar el servicio.
- Seguridad: RDS queda en red privada, inaccesible desde internet.

