Unidad de Nivelación - Fundamentos Esenciales para la Nube (AWS Cloud Foundations)

Introducción: Preparando el Terreno para la Nube

¡Bienvenido/a, futuro/a experto/a en la nube! Este documento ha sido diseñado como un puente sólido y completo para conectar tus conocimientos actuales en desarrollo de aplicaciones con el fascinante y demandado mundo del *cloud computing*. El objetivo de esta unidad de nivelación no es simplemente repasar conceptos, sino construir una pirámide de conocimiento robusta, comenzando desde los cimientos.

Iniciaremos nuestro viaje en el corazón de cualquier sistema informático: el **sistema operativo**. Comprenderemos cómo gestiona los recursos para sentar las bases de todo lo que viene después. A continuación, exploraremos la **virtualización**, la tecnología que revolucionó los centros de datos al permitir una eficiencia sin precedentes y que es el pilar sobre el que se construyen los servicios en la nube. Desde allí, daremos el salto a los **contenedores**, la evolución natural que ha transformado la forma en que desarrollamos, empaquetamos y desplegamos software, ofreciendo una agilidad y portabilidad nunca antes vistas.

Este no es un recorrido teórico abstracto. A lo largo de estas páginas, encontrarás ejemplos prácticos, laboratorios paso a paso y ejercicios diseñados para afianzar tu comprensión. Al finalizar esta unidad, conceptos que hoy pueden sonarte lejanos como "instancia EC2", "contenedor ECS" o "microservicio" no serán meras palabras, sino que estarán anclados en un conocimiento fundamental profundo.

Dominar estos fundamentos es un diferenciador clave en el mercado laboral actual. Te proporcionará la confianza y la perspectiva necesarias para no solo utilizar los servicios de la nube, sino para entender *cómo* y *por qué* funcionan. ¡Empecemos a construir tu futuro en la nube!

Capítulo 1: El Corazón de la Máquina - Repaso de Sistemas Operativos y Virtualización

Para entender cómo funcionan los gigantescos centros de datos de proveedores como AWS,

primero debemos comprender cómo funciona una sola máquina. Todo empieza con el sistema operativo, el software fundamental que da vida al hardware.

1.1. Anatomía de un Sistema Operativo: El Gestor de Recursos

Un sistema operativo (SO) es la capa de software esencial que actúa como intermediario entre el hardware de un ordenador y las aplicaciones que ejecuta el usuario.[1] Piénsalo como el director de una orquesta: el hardware son los músicos (procesador, memoria, discos), cada uno con su función, y las aplicaciones son las partituras. El SO (el director) se asegura de que cada músico toque en el momento adecuado, con la intensidad correcta y en armonía con los demás, para que la sinfonía (lo que el usuario quiere hacer) suene a la perfección.

Componentes Clave

Todo SO moderno, ya sea Windows, macOS, Linux o Android, se estructura en torno a varios componentes fundamentales que trabajan en conjunto.

- **Núcleo (Kernel):** Es el componente central, el corazón del sistema operativo.[2] Es la primera parte del SO que se carga en la memoria y se ejecuta en un nivel de privilegio especial (modo kernel), lo que le da acceso directo y sin restricciones a todo el hardware del sistema.[3] Sus responsabilidades son críticas e intransferibles [1]:
    - **Gestión de Procesos:** Decide qué programa puede usar la CPU, cuándo y por cuánto tiempo, permitiendo la multitarea.[4]
    - **Gestión de Memoria:** Controla cuánta memoria RAM utiliza cada aplicación y dónde se almacena, evitando que los programas interfieran entre sí.[1]
    - **Controladores de Dispositivos:** Actúa como traductor entre el hardware (teclado, disco duro, tarjeta de red) y el software, proporcionando una interfaz estandarizada para que las aplicaciones los utilicen.[3]
    - **Llamadas al Sistema y Seguridad:** Recibe peticiones de servicio de las aplicaciones (como abrir un archivo o conectarse a una red) y gestiona los permisos para mantener la integridad del sistema.[5]

  Existen varios diseños de kernel, siendo los más comunes el **monolítico** (donde todos los servicios principales residen en un único gran bloque de código, como en Linux y Windows) y el **híbrido**, que combina la eficiencia del monolítico con la modularidad de otros diseños.[4]
- **Gestión de Procesos:** Un proceso es, simplemente, un programa en ejecución.[1] El SO, a través de un componente llamado **planificador** (*scheduler*), se encarga de asignar tiempo de CPU a cada proceso de forma equitativa y eficiente, creando la ilusión de que múltiples programas se ejecutan simultáneamente.[1]
- **Gestión de Memoria:** El SO administra el acceso a la memoria RAM, un recurso finito y

volátil. Se asegura de que cada proceso tenga su propio espacio de memoria protegido y, cuando la RAM se agota, utiliza parte del disco duro como **memoria virtual** (o *swap*) para seguir funcionando.[1]
- **Sistema de Archivos:** Este componente organiza cómo se almacenan, nombran y recuperan los datos en los dispositivos de almacenamiento a largo plazo (discos duros, SSDs). Define la estructura de directorios y archivos que vemos y usamos a diario.[2]

1.2. La Revolución de la Virtualización: Multiplicando el Hardware

En los inicios de la computación empresarial, el modelo era simple: un servidor físico, un sistema operativo, una aplicación.[7] Si una empresa necesitaba un servidor de correo, un servidor web y un servidor de bases de datos, necesitaba tres máquinas físicas distintas. Este enfoque, aunque robusto, era tremendamente ineficiente.

- **El Problema Original:** Los servidores físicos estaban crónicamente infrautilizados. La mayoría operaba a solo un 15-20% de su capacidad total, ya que debían estar dimensionados para picos de carga que rara vez ocurrían.[8] Esto generaba un enorme desperdicio en costes de hardware, consumo eléctrico, refrigeración y espacio físico en los centros de datos. Además, la infraestructura era rígida; desplegar un nuevo servidor podía llevar semanas.[8] A estos servidores se les trataba como "mascotas" (*pets*): cada uno era único, configurado a mano, y si fallaba, su recuperación era un proceso largo y doloroso.[8]
- **La Solución: Abstracción del Hardware:** La virtualización surgió como la tecnología disruptiva que resolvió este problema. En lugar de instalar el SO directamente sobre el hardware, se introduce una capa de software intermedia que permite a un único servidor físico ejecutar múltiples **Máquinas Virtuales (VMs)**. Cada VM es un entorno completamente aislado que emula un ordenador completo, con su propio sistema operativo, aplicaciones y recursos virtuales (CPU, RAM, disco).[8] Esta capa mágica es el hipervisor.

1.3. El Director de Orquesta: El Hipervisor

Un **hipervisor**, también conocido como Monitor de Máquina Virtual (VMM), es el software que crea, ejecuta y gestiona las máquinas virtuales.[7] Su trabajo es tomar los recursos físicos del servidor anfitrión (*host*) y repartirlos de forma controlada entre las diferentes máquinas virtuales invitadas (*guests*). Existen dos tipos principales de hipervisores, y su diferencia es fundamental para entender el rendimiento y los casos de uso en el mundo real, incluyendo la nube.

- **Tipo 1 (Bare-Metal):** Este tipo de hipervisor se instala directamente sobre el hardware físico del servidor, actuando como un sistema operativo muy ligero y especializado en virtualización.[14] No hay un SO anfitrión de por medio.

    - **Ventajas:** Son extremadamente eficientes, seguros y estables, ya que tienen acceso directo al hardware sin intermediarios.
    - **Casos de Uso:** Son el estándar en centros de datos empresariales y en la infraestructura de los proveedores de cloud como AWS.
    - **Ejemplos:** VMware ESXi, Microsoft Hyper-V, KVM (Kernel-based Virtual Machine, la base de muchas nubes de código abierto), Xen.
- **Tipo 2 (Hosted o Alojado):** Este hipervisor se ejecuta como una aplicación más sobre un sistema operativo anfitrión convencional (como Windows, macOS o Linux).[14]
    - **Ventajas:** Son muy fáciles de instalar y utilizar, ideales para que desarrolladores, estudiantes y entusiastas puedan ejecutar diferentes sistemas operativos en sus ordenadores personales.
    - **Casos de Uso:** Desarrollo de software, pruebas, aprendizaje y ejecución de aplicaciones que no son compatibles con el SO principal.
    - **Ejemplos:** Oracle VirtualBox, VMware Workstation, VMware Fusion (para Mac), Parallels Desktop (para Mac).

La distinción entre estos dos tipos no es solo técnica; representa un equilibrio fundamental entre rendimiento y conveniencia que se repite en toda la tecnología. Un hipervisor de Tipo 1 elimina la capa del SO anfitrión, lo que se traduce en una menor sobrecarga (*overhead*) de recursos y un acceso más directo y rápido al hardware.[12] Esta arquitectura es la que se busca en entornos de producción donde cada ciclo de CPU y cada milisegundo de latencia cuentan. Por otro lado, un hipervisor de Tipo 2 prioriza la facilidad de uso; se instala como cualquier otro programa, pero este confort tiene un precio: cada solicitud de recursos de la VM debe atravesar tanto el hipervisor como el SO anfitrión antes de llegar al hardware, introduciendo una latencia inevitable.[12] Este dilema (Rendimiento y Seguridad vs. Facilidad y Flexibilidad) es un patrón que observarás constantemente en el mundo cloud.

1.4. ¿Qué Hace a un Hipervisor Mejor que Otro? Criterios de Rendimiento

En un entorno profesional, la elección de un hipervisor no es trivial y se basa en métricas de rendimiento cuantificables. Un buen hipervisor es aquel que introduce la menor sobrecarga posible, es decir, que es casi "invisible" en términos de consumo de recursos.

- **Sobrecarga de CPU (Overhead):** Mide el porcentaje de la capacidad del procesador que el propio hipervisor consume para gestionar las VMs. Un overhead bajo significa que más potencia de cálculo está disponible para las aplicaciones.[17] Un valor saludable es aquel que se mantiene por debajo del 60% del consumo total del procesador en condiciones de carga normales.[19]
- **Gestión de Memoria:** La eficiencia con la que el hipervisor asigna la RAM es crucial. Técnicas avanzadas como el *memory ballooning* permiten al hipervisor reclamar memoria no utilizada de una VM para asignarla a otra que la necesite más.[18] Una métrica clave a monitorizar en el host es la memoria libre disponible; valores por debajo

del 10-25% indican un posible cuello de botella.[19]

- **Rendimiento de E/S de Disco (I/O):** La latencia de disco, es decir, el tiempo que tarda el disco en responder a una petición de lectura o escritura, es uno de los factores que más impactan en la percepción de velocidad de una aplicación. Se mide en milisegundos (ms).[19]
    - **Saludable:** 1-15 ms
    - **Advertencia:** 15-25 ms
    - **Crítico:** >25 ms (el rendimiento se verá afectado negativamente).[19]
- **Rendimiento de Red:** Se evalúa el ancho de banda disponible para las VMs, la latencia (el tiempo que tarda un paquete en ir y volver) y la pérdida de paquetes. En una red local (LAN), la latencia entre VMs debería ser inferior a 1 ms.[19] El uso del ancho de banda de la interfaz de red no debería superar de forma sostenida el 65% para evitar la saturación.[19]

1.5. Laboratorio Práctico: VMware vs. VirtualBox

Para que puedas experimentar con la virtualización en tu propio equipo, es útil comparar dos de los hipervisores de Tipo 2 más populares. La elección entre uno y otro no debe ser aleatoria, sino basada en tus necesidades específicas, una habilidad de evaluación que te será muy útil en tu carrera.

Característica	Oracle VirtualBox	VMware Workstation Player/Pro
Licencia/Coste	Gratuito y Open Source (GPLv2) [20]	Player gratuito para uso personal; Pro es de pago [20]
Rendimiento	Bueno para uso general, pero puede ser más lento en tareas con gráficos 3D intensivos [20]	Generalmente superior, mejor optimización para gráficos 3D y tareas pesadas [22]
Características Avanzadas	Funcionalidad básica de snapshots, clonado limitado [20]	Snapshots múltiples avanzados, clonado completo, mejores herramientas de red virtual [21]
Soporte de SO (Host)	Muy amplio: Windows, macOS, Linux, Solaris [22]	Windows, Linux. Para macOS existe VMware Fusion [22]
Facilidad de Uso	Interfaz más simple, considerada más fácil para principiantes [20]	Interfaz pulida, pero con una curva de aprendizaje ligeramente mayor por sus opciones avanzadas [20]
Caso de Uso Ideal	Estudiantes, desarrolladores individuales, pruebas rápidas,	Entornos empresariales, desarrolladores que necesitan

	entornos que no requieren máximo rendimiento [20]	alto rendimiento, pruebas de software complejas [23]

Esta tabla ilustra un trade-off clásico: VirtualBox es la navaja suiza gratuita y universalmente compatible, perfecta para empezar sin barreras.[22] VMware, por su parte, es la herramienta de precisión de alto rendimiento, cuyas características avanzadas justifican su coste en entornos profesionales.[23]

1.6. Ejercicios Propuestos (Capítulo 1)

1. **Pregunta Conceptual:** Explica con tus propias palabras por qué un hipervisor de Tipo 1 es generalmente más rápido que uno de Tipo 2. ¿Qué capa de software se elimina y cuál es el impacto directo en el rendimiento de las máquinas virtuales?
2. **Caso Práctico:** Quieres probar la última versión de Ubuntu Desktop en tu portátil con Windows para un proyecto de la universidad. No necesitas el máximo rendimiento gráfico, pero sí una instalación rápida, sencilla y sin coste. Basándote en la tabla comparativa, ¿qué hipervisor elegirías y por qué? Justifica tu respuesta mencionando al menos dos características.
3. **Investigación:** KVM (Kernel-based Virtual Machine) es un hipervisor de Tipo 1 que está integrado directamente en el kernel de Linux.[14] Investiga y explica brevemente cómo funciona esta integración y por qué crees que esta característica lo ha convertido en la base tecnológica para muchos de los gigantes de la computación en la nube.

Capítulo 2: Docker - De Cero a Héroe en Contenedores

La virtualización con VMs fue un paso de gigante, pero la industria tecnológica nunca se detiene en su búsqueda de mayor eficiencia y velocidad. Las VMs, aunque eficaces, arrastraban un peso considerable que limitaba la agilidad. La siguiente revolución ya está aquí y se llama "contenedores".

2.1. El Siguiente Salto Evolutivo: Contenedores vs. Máquinas Virtuales

- **El Problema con las VMs:** A pesar de sus ventajas, las VMs tienen una ineficiencia inherente: cada una de ellas empaqueta un sistema operativo invitado completo. Esto significa que si tienes 10 VMs en un servidor, tienes 10 copias completas de un SO (con sus kernels, librerías y binarios) consumiendo recursos, ocupando gigabytes de espacio en disco y tardando minutos en arrancar.[27] Es como si para construir 10 apartamentos en un edificio, construyeras 10 casas completas, con sus propios cimientos y tejados,

una encima de la otra.

- **La Solución de los Contenedores:** Los contenedores abordan este problema con un enfoque radicalmente diferente. En lugar de virtualizar el hardware, realizan una **virtualización a nivel de sistema operativo**. Todos los contenedores que se ejecutan en un host comparten el mismo kernel del sistema operativo de esa máquina. Un contenedor es simplemente un paquete que contiene una aplicación y *solo* sus dependencias directas (librerías y binarios).[27] Esto los hace increíblemente ligeros (suelen pesar megabytes), rápidos (arrancan en segundos) y extremadamente portátiles.

La siguiente tabla resume las diferencias fundamentales, que son cruciales para entender por qué los contenedores han cambiado las reglas del juego en el desarrollo de software.

Característica	Máquinas Virtuales (VMs)	Contenedores (Docker)
Nivel de Abstracción	Virtualización del **Hardware**	Virtualización del **Sistema Operativo** [27]
Arquitectura	Cada VM tiene su propio **SO Invitado** completo (con su propio kernel) sobre un Hipervisor [27]	Comparten el **Kernel del SO Anfitrión**. Solo empaquetan la app y sus dependencias [27]
Tamaño	Pesadas (varios Gigabytes)	Ligeros (decenas de Megabytes) [30]
Tiempo de Arranque	Lento (minutos)	Rápido (segundos) [27]
Uso de Recursos	Alto (CPU, RAM y disco por cada SO invitado)	Bajo (comparten recursos de forma más eficiente) [27]
Aislamiento	**Completo**. Aislamiento a nivel de kernel. Muy seguro.[27]	**A nivel de proceso**. Comparten el kernel del host. Menos aislado que una VM por defecto.[27]
Portabilidad	Portátiles, pero mover imágenes de GB es costoso.	Altamente portátiles. Resuelven el "funciona en mi máquina". [30]

La diferencia arquitectónica es la causa de todo lo demás. Al no tener que emular hardware ni cargar un SO completo, los contenedores son órdenes de magnitud más eficientes en tamaño y velocidad. Esto permite una mayor "densidad" de aplicaciones por servidor, reduciendo costes. Sin embargo, existe un compromiso: el aislamiento. Las VMs ofrecen una barrera de seguridad muy fuerte a nivel de kernel. Si un atacante compromete una VM, las demás en el mismo host están a salvo. En los contenedores, al compartir el kernel del host, una vulnerabilidad a ese nivel podría, teóricamente, afectar a todos los contenedores.[27] Esta es una consideración de seguridad vital en entornos de producción.

2.2. El Ecosistema Docker: Piezas del Puzzle

Docker no es solo una tecnología, es una plataforma completa con varias piezas que trabajan juntas para hacer posible la magia de los contenedores.

- **Docker Engine:** Es el motor, el corazón de Docker. Funciona con una arquitectura cliente-servidor [33]:
    - **Docker Daemon (dockerd):** Es un proceso que se ejecuta constantemente en segundo plano en el host. Es el responsable de construir, ejecutar y gestionar los objetos de Docker (imágenes, contenedores, redes, etc.).[33]
    - **API REST:** Es el lenguaje que utilizan los programas para hablar con el daemon y darle órdenes.[33]
    - **Cliente Docker (docker):** Es la herramienta de línea de comandos (CLI) que nosotros, como usuarios, utilizamos para interactuar con el daemon a través de la API. Cada vez que escribes docker run, estás usando el cliente para enviar una instrucción al daemon.[33]
- **Imágenes Docker:** Son plantillas de solo lectura que sirven como base para crear contenedores. Una imagen es **inmutable**: una vez creada, no se puede cambiar.[35] Está compuesta por una serie de capas apiladas, donde cada capa representa una instrucción en el proceso de construcción. Contiene todo lo necesario: el código de la aplicación, un runtime (como Node.js o Python), librerías, variables de entorno y archivos de configuración.[36] La analogía perfecta es que una imagen es la **receta** o el **plano** de nuestra aplicación.[38]
- **Contenedores Docker:** Un contenedor es una instancia en ejecución de una imagen. Es el resultado tangible de la "receta". Puedes crear múltiples contenedores a partir de la misma imagen, igual que puedes cocinar muchos platos idénticos con la misma receta.[35] Por defecto, los contenedores son **efímeros**: cualquier cambio que hagas dentro de un contenedor (como crear un archivo) se perderá cuando el contenedor se elimine.
- **Dockerfile:** Es un simple archivo de texto que contiene las instrucciones, paso a paso, que Docker sigue para construir una imagen. Es la receta escrita en un formato que Docker entiende.[32] Automatiza por completo la creación de imágenes, garantizando que sean siempre idénticas y reproducibles.
- **Docker Hub / Registry:** Un *registry* es un sistema de almacenamiento y distribución de imágenes. **Docker Hub** es el registry público más grande y conocido, gestionado por Docker Inc. Es el equivalente a GitHub para el código fuente: un lugar donde puedes encontrar miles de imágenes pre-construidas (oficiales y de la comunidad) y donde puedes subir y compartir las tuyas.[33]

2.3. Flujo de Trabajo Esencial con la CLI de Docker

Manos a la obra. La interacción con Docker se realiza principalmente a través de su potente interfaz de línea de comandos (CLI).

- **Instalación:**
  - Para **Windows** y **macOS**, la forma más sencilla es instalar **Docker Desktop**, que incluye el motor, la CLI y otras herramientas gráficas. Puedes descargarlo desde el(https://www.docker.com/products/docker-desktop/).
  - Para **Linux**, se instala el **Docker Engine**. El proceso varía ligeramente según la distribución (Ubuntu, CentOS, etc.), pero generalmente implica añadir el repositorio oficial de Docker e instalar los paquetes correspondientes.[32]
- **Comandos Fundamentales (con ejemplos):**

  Gestión de Imágenes

  - **Descargar una imagen de un registry:**
    Bash
    # Descarga la última versión de la imagen de Nginx desde Docker Hub
    docker pull nginx:latest

    pull trae la imagen a tu máquina local para que puedas usarla.[40]
  - **Listar imágenes locales:**
    Bash
    docker images

    Muestra todas las imágenes que tienes en tu sistema, con su nombre, tag (versión) y tamaño.[40]
  - **Eliminar una imagen:**
    Bash
    # Elimina la imagen de nginx (puedes usar el nombre o el ID de la imagen)
    docker rmi nginx

    rmi significa *remove image*.[40]
    Gestión de Contenedores
  - **Crear y arrancar un contenedor:**
    Bash
    # Crea y arranca un contenedor a partir de la imagen de Ubuntu
    # -it: modo interactivo con una terminal
    # --rm: el contenedor se eliminará automáticamente al salir
    docker run -it --rm ubuntu bash

    run es el comando más versátil. Opciones comunes son -d (modo

detached **o segundo plano), -p (publicar puertos) y --name (asignar un nombre).**[32]

- **Listar contenedores:**
  Bash
  # Lista los contenedores que están actualmente en ejecución
  docker ps

  # Lista todos los contenedores (en ejecución y detenidos)
  docker ps -a

  Este comando es tu panel de control para ver qué está pasando.[35]
- **Detener e iniciar contenedores:**
  Bash
  # Detiene un contenedor en ejecución (usando su nombre o ID)
  docker stop mi-contenedor

  # Inicia un contenedor que estaba detenido
  docker start mi-contenedor

  Estos comandos permiten gestionar el ciclo de vida de tus contenedores.[40]
- **Eliminar un contenedor:**
  Bash
  # Elimina un contenedor que está detenido
  docker rm mi-contenedor

  rm solo funciona con contenedores detenidos. Para forzar la eliminación de uno en ejecución, se puede usar docker rm -f.[40]
- **Ver los logs de un contenedor:**
  Bash
  # Muestra la salida estándar del contenedor (muy útil para depurar)
  docker logs mi-contenedor

  Para seguir los logs en tiempo real, añade la opción -f.[42]
- **Ejecutar un comando dentro de un contenedor:**
  Bash
  # Abre una sesión de shell (bash) dentro de un contenedor que ya está en ejecución
  docker exec -it mi-contenedor bash

  exec es increíblemente útil para inspeccionar el estado de un contenedor o ejecutar comandos de mantenimiento.[42]
  Construcción de Imágenes

- **Construir una imagen desde un Dockerfile:**
  Bash
  # Construye una imagen usando el Dockerfile del directorio actual (.)
  # -t: asigna un nombre (name) y una etiqueta (tag) a la imagen
  docker build -t mi-app:1.0.

  Este comando lee tu Dockerfile, ejecuta las instrucciones y crea una nueva imagen local.[32]

2.4. Caso Práctico: Tu Primer Servidor Web con Nginx

Vamos a aplicar lo aprendido para levantar un servidor web Nginx que sirva una página HTML personalizada desde tu máquina.

- Paso 1: Crear la estructura del proyecto.
  En tu terminal, crea una carpeta para el proyecto y los archivos necesarios:
  Bash
  mkdir mi-web-nginx
  cd mi-web-nginx
  mkdir html
  echo "<h1>¡Hola DAM desde Nginx en Docker!</h1>" > html/index.html

  Ahora tienes una carpeta mi-web-nginx que contiene una subcarpeta html con un archivo index.html dentro.
- Paso 2: Levantar el contenedor Nginx con un bind mount.
  Un bind mount es una técnica que "monta" o sincroniza una carpeta de tu máquina (el host) dentro del contenedor. Esto es ideal para desarrollo, ya que los cambios en tu código se reflejan instantáneamente sin tener que reconstruir la imagen.
  Bash
  docker run \
  -d \
  --name mi-servidor-web \
  -p 8080:80 \
  -v "$(pwd)"/html:/usr/share/nginx/html:ro \
  nginx:latest

  Desglosemos este comando:
  - docker run: El comando para crear y ejecutar un contenedor.
  - -d: Modo *detached*. El contenedor se ejecuta en segundo plano.
  - --name mi-servidor-web: Le damos un nombre descriptivo para identificarlo fácilmente.
  - -p 8080:80: Mapeo de puertos. Redirige el tráfico que llega al puerto 8080 de tu

máquina (host) hacia el puerto 80 dentro del contenedor, que es donde Nginx escucha por defecto.
    - -v "$(pwd)"/html:/usr/share/nginx/html:ro: Este es el *bind mount*.
        - "$(pwd)"/html: Es la ruta absoluta a tu carpeta html local. $(pwd) se expande al directorio de trabajo actual.
        - /usr/share/nginx/html: Es la ruta dentro del contenedor donde Nginx espera encontrar los archivos web a servir.
        - :ro: Significa *read-only*. Es una buena práctica de seguridad para asegurar que el contenedor no pueda modificar los archivos de tu máquina.
    - nginx:latest: La imagen que queremos utilizar.
- Paso 3: Verificar.
  Abre tu navegador web y visita http://localhost:8080. Deberías ver tu mensaje: "¡Hola DAM desde Nginx en Docker!".
  Ahora, la magia del *bind mount*: abre el archivo html/index.html en tu editor de código, cambia el mensaje y guarda el archivo. Refresca la página en tu navegador. ¡El cambio aparece al instante! Esto demuestra por qué esta técnica es tan poderosa para el ciclo de desarrollo.

2.5. Profundizando en Nginx: El Rey de los Servidores Web Modernos

Has usado Nginx, pero ¿por qué es tan omnipresente en el mundo del desarrollo web y la nube?

- **¿Qué es Nginx?** Nginx es un software de código abierto de alto rendimiento que puede actuar como **servidor web**, **proxy inverso**, **balanceador de carga** y **caché HTTP**.[43] Es una auténtica navaja suiza para la gestión del tráfico web.
- **¿Por qué es tan popular?**
    - **Rendimiento y Escalabilidad Superiores:** La principal razón de su éxito es su arquitectura. A diferencia de servidores tradicionales como Apache, que históricamente usaban un hilo por cada conexión (un modelo que consume mucha memoria bajo alta carga), Nginx utiliza una arquitectura **asíncrona y basada en eventos**.[44] Un único proceso de Nginx puede manejar miles de conexiones simultáneas de manera muy eficiente, con un consumo de CPU y RAM increíblemente bajo.[46] Esto lo hace perfecto para sitios web con un volumen de tráfico masivo.
    - **Eficiencia con Contenido Estático:** Es extremadamente rápido sirviendo archivos estáticos (HTML, CSS, JavaScript, imágenes, etc.).[44]
    - **Proxy Inverso:** Esta es una de sus funciones más potentes. Un proxy inverso es un servidor que se sitúa "delante" de tus servidores de aplicaciones. Recibe las peticiones de los clientes y las reenvía al servidor de aplicación adecuado.[48] Esto tiene enormes ventajas:
        - **Seguridad:** Oculta la estructura de tu red interna. Los clientes solo hablan

con Nginx, nunca directamente con tus servidores de aplicación.
        - **Terminación SSL/TLS:** Puede gestionar todo el cifrado y descifrado de las conexiones HTTPS, liberando a los servidores de aplicación de esta pesada tarea.
        - **Flexibilidad:** Permite componer sistemas complejos detrás de una única fachada.
    - **Balanceo de Carga:** Cuando tienes múltiples réplicas de tu aplicación para manejar más tráfico, Nginx puede distribuir las peticiones entrantes entre ellas de forma inteligente, asegurando que ninguna se sobrecargue y mejorando la disponibilidad del servicio.[45]

La combinación de Docker y Nginx es la base de gran parte de la arquitectura web moderna. En un sistema de **microservicios**, donde una aplicación se descompone en muchos servicios pequeños e independientes, es una práctica estándar usar un contenedor Nginx como **API Gateway** o punto de entrada. Este Nginx recibe todo el tráfico y lo enruta inteligentemente al microservicio correcto (por ejemplo, el servicio de usuarios, el de productos, el de pagos), cada uno ejecutándose en su propio contenedor. Nginx se convierte en el director de tráfico, mientras que Docker proporciona los entornos aislados y portátiles para cada componente de la aplicación. Entender Nginx, por tanto, no es solo aprender sobre un servidor web; es comprender una pieza clave para construir los sistemas distribuidos, escalables y resilientes que se despliegan en la nube.

2.6. Ejercicios Resueltos y Propuestos (Capítulo 2)

- **Ejercicio Resuelto 1: Creando una imagen personalizada con un Dockerfile.**
    - **Objetivo:** En el caso práctico anterior, nuestra web dependía de una carpeta local. Ahora, vamos a empaquetar la web *dentro* de la imagen de Nginx para que sea totalmente autocontenida y portable.
    - **Pasos:**
        1. Dentro de tu carpeta mi-web-nginx, crea un nuevo archivo llamado Dockerfile (sin extensión).
        2. Añade el siguiente contenido al Dockerfile:
           Dockerfile
           # 1. Usar la imagen oficial de Nginx como imagen base
           FROM nginx:latest

           # 2. Copiar el contenido de nuestra carpeta local 'html'
           #   al directorio por defecto de Nginx dentro de la imagen
           COPY./html /usr/share/nginx/html

        3. Ahora, construye la imagen desde tu terminal (asegúrate de estar en la carpeta mi-web-nginx):
           Bash

docker build -t mi-web-personalizada:1.0.

4. Una vez construida, puedes ejecutar un contenedor a partir de tu nueva imagen. Fíjate que ahora no necesitamos un *bind mount* (-v):
   Bash
   docker run -d --name mi-web-desde-imagen -p 8888:80 mi-web-personalizada:1.0

5. Verifica el resultado abriendo tu navegador en http://localhost:8888. Deberías ver tu página. ¡Ahora tienes una imagen portable que contiene tu aplicación!

- **Ejercicio Propuesto 1:** Modifica el Dockerfile del ejercicio resuelto para añadir metadatos a tu imagen. Investiga las instrucciones LABEL y MAINTAINER (aunque esta última está obsoleta, es bueno conocerla). Añade una etiqueta LABEL version="1.1" y otra con tu nombre como autor. Vuelve a construir la imagen con el tag 1.1 y utiliza el comando docker image inspect mi-web-personalizada:1.1 para verificar que los metadatos se han añadido correctamente.
- **Ejercicio Propuesto 2:** Busca en Docker Hub la imagen oficial del servidor web Apache, que se llama httpd. El reto es levantar un contenedor usando la imagen httpd que sirva el mismo archivo index.html del caso práctico 2.4. Para ello, tendrás que investigar cuál es el directorio por defecto donde Apache sirve los archivos web (pista: no es el mismo que Nginx). Utiliza un *bind mount* para lograrlo. Compara la sintaxis del comando con la que usaste para Nginx.

Capítulo 3: Conectando los Puntos - Redes en Docker

Hemos visto cómo crear y ejecutar contenedores aislados. Pero en el mundo real, las aplicaciones no son islas; necesitan comunicarse entre sí. Un servidor web necesita hablar con una base de datos, una API necesita conectarse a un servicio de caché, etc. El subsistema de red de Docker es el que hace posible esta comunicación de forma segura y eficiente.

3.1. Aislamiento y Comunicación: Tipos de Redes en Docker

Docker gestiona la conectividad de los contenedores a través de un sistema de *drivers* de red. Cada driver implementa una estrategia de red diferente.[51] Estos son los más importantes:

- **bridge (Puente):** Es el driver por defecto para contenedores independientes. Cuando instalas Docker, se crea una red bridge llamada bridge. Al arrancar un contenedor sin especificar una red, se conecta a esta. Esta red es privada y está aislada del exterior. Los contenedores dentro de la misma red bridge pueden comunicarse entre sí usando

sus direcciones IP internas. Para que un servicio dentro de un contenedor sea accesible desde fuera del host, es necesario mapear un puerto del host a un puerto del contenedor con la opción -p.[52]

- **host (Anfitrión):** Este driver elimina por completo el aislamiento de red entre el contenedor y el host. El contenedor comparte directamente la pila de red del host, lo que significa que no tiene su propia IP; utiliza la IP del host. Si una aplicación en el contenedor escucha en el puerto 80, estará disponible directamente en el puerto 80 del host.
    - **Ventaja:** Ofrece el máximo rendimiento de red, ya que no hay ninguna capa de virtualización de red.
    - **Desventaja:** Pierde el beneficio del aislamiento. Si ejecutas varios contenedores que quieren usar el mismo puerto, tendrás un conflicto. Es menos seguro.
    - **Caso de uso:** Aplicaciones que necesitan un rendimiento de red extremo y donde el aislamiento de puertos no es una preocupación, como ciertos tipos de monitorización o enrutamiento de red.[52]
- **none (Ninguna):** Este driver deja al contenedor completamente aislado de la red. El contenedor tendrá una interfaz de red de *loopback* (lo) para comunicarse consigo mismo, pero no podrá hablar con otros contenedores ni con el exterior.
    - **Caso de uso:** Para contenedores que realizan tareas que no requieren conectividad, como trabajos de procesamiento por lotes, o cuando se desea configurar una solución de red completamente personalizada.[51]
- **overlay (Superpuesta):** Este es un driver avanzado diseñado para la comunicación entre contenedores que se ejecutan en **diferentes hosts Docker**. Crea una red virtual distribuida que se "superpone" a la red física de las máquinas. Es la tecnología de red que utilizan los orquestadores de contenedores como Docker Swarm y Kubernetes para permitir que los servicios se comuniquen de forma transparente a través de un clúster de máquinas.[52] No profundizaremos en él, pero es fundamental que sepas que existe y para qué sirve.

3.2. Creando Redes Privadas: El Poder de las Redes bridge Personalizadas

Aunque la red bridge por defecto es útil, tiene una limitación muy importante para aplicaciones reales: **no proporciona resolución de nombres automática**. Esto significa que si tienes dos contenedores, app y db, en la red bridge por defecto, app solo puede conectarse a db si conoce su dirección IP interna (ej. 172.17.0.3). El problema es que esta IP puede cambiar cada vez que el contenedor db se reinicia, lo que haría que la aplicación app se rompiera.[53]

La solución es utilizar **redes bridge personalizadas**, creadas por el usuario. Ofrecen ventajas enormes:

- **Aislamiento de Red Superior:** Puedes crear múltiples redes para segmentar tus

aplicaciones. Por ejemplo, una red para el frontend y otra para el backend. Los contenedores solo pueden comunicarse con otros que estén en la misma red, proporcionando un aislamiento lógico y de seguridad.[56]

- **Resolución de Nombres (DNS) Automática:** ¡Esta es la característica estrella! Dentro de una red bridge personalizada, Docker proporciona un servidor DNS interno. Esto permite que los contenedores se descubran y comuniquen entre sí utilizando sus **nombres de contenedor** como si fueran nombres de dominio. El contenedor app puede ahora conectarse a db simplemente usando el hostname db en su cadena de conexión. Docker se encarga de resolver db a la IP interna correcta, sin importar si esta cambia.[56]
- **Gestión Dinámica:** Puedes conectar y desconectar contenedores de redes en caliente, sin necesidad de detenerlos o reiniciarlos, lo que ofrece una gran flexibilidad.[56]

Esta capacidad de resolución de nombres es el pilar técnico que hace viables las arquitecturas de microservicios en un entorno Docker. Sin ella, la comunicación entre los cientos de servicios que pueden componer una aplicación moderna sería una pesadilla de gestión de IPs. Docker lo soluciona de forma nativa y elegante, introduciendo el concepto de **descubrimiento de servicios** (*service discovery*), un patrón fundamental en los sistemas distribuidos. Al enseñar a usar redes bridge personalizadas, no solo se está enseñando un comando de Docker, sino el primer y más importante paso para construir aplicaciones compuestas, resilientes y escalables.

3.3. Laboratorio de Redes: Comunicación Segura entre Contenedores

Vamos a demostrar el poder de las redes personalizadas con un ejemplo práctico.

- **Objetivo:** Crear una aplicación simulada con un contenedor "frontend" y un "backend", y demostrar que pueden comunicarse por nombre dentro de una red privada, pero que el "backend" está aislado del mundo exterior.
- **Paso 1: Crear una red bridge personalizada.**
  Bash
  ```
  # Creamos una nueva red de tipo bridge llamada 'mi-app-net'
  docker network create mi-app-net
  ```

  Puedes verificar que se ha creado con el comando docker network ls.
- Paso 2: Lanzar un contenedor "backend".
  Usaremos una imagen ligera como alpine y la mantendremos en ejecución con un comando que no hace nada (sleep infinity).
  Bash
  ```
  # Lanzamos el contenedor en nuestra red, le damos el nombre 'backend'
  # y lo dejamos corriendo en segundo plano (-d)
  docker run -d --name backend --network mi-app-net alpine sleep infinity
  ```

- Paso 3: Lanzar un contenedor "frontend" e intentar la comunicación.
  Ahora, lanzamos otro contenedor en la misma red. Esta vez, lo haremos en modo interactivo (-it) para poder ejecutar comandos dentro de él.
  Bash
  # Lanzamos el frontend, lo conectamos a la misma red, y abrimos una shell (sh)
  # --rm hará que el contenedor se elimine automáticamente cuando salgamos de la shell
  docker run -it --rm --name frontend --network mi-app-net alpine sh

  Una vez que estés dentro de la shell del contenedor frontend (verás un prompt como / #), ejecuta el siguiente comando:
  Bash
  # Hacemos ping al contenedor 'backend' ¡USANDO SU NOMBRE!
  ping -c 4 backend

  Observarás una salida como esta:
  PING backend (172.19.0.2): 56 data bytes
  64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.102 ms
  64 bytes from 172.19.0.2: seq=1 ttl=64 time=0.098 ms

...
```

¡Funciona! El DNS interno de Docker ha resuelto el nombre backend a su dirección IP interna (172.19.0.2 en este caso) y la comunicación es exitosa.

- **Paso 4: Demostrar el aislamiento.**
  1. Sal del contenedor frontend (escribe exit y pulsa Enter).
  2. Ahora, desde la terminal de tu máquina host, intenta hacer ping al backend:
     Bash
     ping backend

     El comando fallará con un error tipo "unknown host". Esto demuestra que backend no es accesible desde fuera de su red Docker privada.
  3. Para una prueba final, lanza un contenedor en la red bridge por defecto e intenta comunicarte:
     Bash
     docker run -it --rm alpine sh
     / # ping backend

     De nuevo, fallará. Esto confirma el aislamiento entre diferentes redes Docker.

## 3.4. Ejercicios Resueltos y Propuestos (Redes)

- **Ejercicio Resuelto 1: Conectar un contenedor a múltiples redes.**
```

- **Escenario:** Tenemos nuestro contenedor backend en la red mi-app-net. Ahora queremos que un nuevo contenedor de "monitorización" pueda acceder a él, pero sin que este contenedor de monitorización esté en la misma red que el frontend, por seguridad.
- **Pasos:**
    1. **Crear una segunda red:**
       Bash
       docker network create red-monitoring

    2. **Conectar el contenedor backend (que ya está en ejecución) a esta nueva red:**
       Bash
       docker network connect red-monitoring backend

    3. **Inspeccionar el contenedor backend:**
       Bash
       docker inspect backend

       Si buscas en la salida JSON la sección "Networks", verás que ahora backend está conectado tanto a mi-app-net como a red-monitoring, y tiene una IP en cada una.
    4. **Lanzar el contenedor de monitorización y verificar:**
       Bash
       docker run -it --rm --name monitor --network red-monitoring alpine sh
       / # ping backend

       El ping funcionará, ya que ambos están en la red red-monitoring. Sin embargo, el contenedor frontend no puede ver al contenedor monitor, y viceversa, porque están en redes diferentes (salvo por el backend que actúa de puente).

- **Ejercicio Propuesto 1:** Diseña y crea una arquitectura de tres contenedores con los siguientes requisitos de seguridad:
    - Un contenedor servidor-web (usando la imagen nginx).
    - Un contenedor api (usando la imagen alpine).
    - Un contenedor base-de-datos (usando la imagen alpine).
    - **Reglas de comunicación:**
        1. El servidor-web SÓLO debe poder comunicarse con la api.
        2. La api debe poder comunicarse con la base-de-datos.
        3. El servidor-web NO debe poder comunicarse directamente con la base-de-datos.
    - **Tarea:** Escribe y ejecuta los comandos docker network create y docker run necesarios para implementar esta topología. (Pista: necesitarás más de una red y al menos un contenedor conectado a dos de ellas).

Capítulo 4: La Nube por Dentro - De la Infraestructura Física a los Servicios de AWS

Hasta ahora, hemos construido nuestro conocimiento desde los cimientos: el SO, la virtualización y los contenedores. Ahora es el momento de levantar la vista y ver cómo estos conceptos fundamentales se manifiestan a escala masiva en la nube. Lo que has aprendido no es solo teoría; es el lenguaje con el que se construyen los servicios de AWS.

4.1. La Gran Historia de la Infraestructura TI: Una Narrativa de Evolución

La historia de la infraestructura de TI es una búsqueda constante de mayor abstracción, eficiencia y agilidad. Cada etapa ha sido una respuesta a las limitaciones de la anterior, impulsada tanto por innovaciones técnicas como por las necesidades cambiantes del negocio.

- **Era 1: Servidores Físicos ("Pets"):** En los inicios, cada servidor era una "mascota".[8] Se le daba un nombre, se configuraba manualmente con mimo y, si enfermaba (fallaba), todo el equipo se volcaba en curarlo. El negocio dependía directamente de la salud de ese hardware específico.
    - **Motivación para el cambio:** Este modelo era insostenible. Era **ineficiente** (baja utilización), **costoso** (hardware, energía, espacio), **lento** (semanas para aprovisionar un nuevo servidor) y **frágil**.[8] Las empresas no podían innovar a la velocidad que el mercado exigía.
- **Era 2: Máquinas Virtuales ("Cattle"):** La virtualización introdujo el paradigma del "ganado".[8] Las VMs son homogéneas, idénticas y reemplazables. Si una VM falla, se elimina y se crea una nueva a partir de una plantilla, sin sentimentalismos. El foco pasó de mantener máquinas individuales a mantener el servicio que estas proporcionaban.
    - **Motivación para el cambio:** Aunque supuso una mejora drástica en la utilización de recursos y la resiliencia, las VMs seguían siendo relativamente pesadas y lentas. Desplegar una nueva VM todavía implicaba arrancar un SO completo, un proceso que podía llevar varios minutos. La gestión de parches y actualizaciones para cientos de sistemas operativos seguía siendo una carga operativa considerable.[8] El negocio demandaba aún más velocidad.
- **Era 3: Contenedores ("Insects"):** Los contenedores representan el siguiente nivel de abstracción. Son tan ligeros, rápidos y desechables que se les puede comparar con "insectos". Se pueden crear y destruir miles de ellos en segundos para responder a la demanda.
    - **Motivación de negocio:** La adopción masiva de contenedores está impulsada

por beneficios de negocio directos:

- **Agilidad Extrema:** Permiten a los equipos de desarrollo lanzar nuevas funcionalidades mucho más rápido.[59]
- **DevOps y CI/CD:** Son la tecnología perfecta para las prácticas de Integración Continua y Despliegue Continuo, automatizando el camino del código desde el desarrollador hasta producción.[30]
- **Portabilidad y Consistencia:** Un contenedor se ejecuta exactamente igual en el portátil de un desarrollador, en un servidor de pruebas y en la nube de producción, eliminando el clásico "en mi máquina funciona".[30]
- **Eficiencia de Costes:** La alta densidad de contenedores por servidor reduce drásticamente los costes de infraestructura.[29]

4.2. ¿VMs o Contenedores? Criterios de Decisión en la Nube

En la nube moderna, las VMs y los contenedores no son enemigos; son herramientas diferentes para trabajos diferentes. La elección correcta depende del problema que se quiera resolver.

- **Cuándo usar Máquinas Virtuales (como Amazon EC2):**
  - **Aislamiento de Seguridad Fuerte:** Cuando el aislamiento es la máxima prioridad. Para aplicaciones que manejan datos muy sensibles (financieros, médicos) o para arquitecturas *multi-tenant* donde se debe garantizar que un cliente no pueda afectar a otro, el aislamiento a nivel de kernel de las VMs es superior.[10]
  - **Requisitos de Sistema Operativo Específicos:** Cuando necesitas ejecutar un sistema operativo completo y diferente al del host (ej. una aplicación Windows en la infraestructura Linux de AWS) o cuando una aplicación requiere un control total y modificaciones a bajo nivel del kernel.[10]
  - **Aplicaciones Heredadas (*Legacy*):** Para migrar aplicaciones monolíticas antiguas a la nube ("lift-and-shift"). A menudo es más fácil y rápido empaquetar toda la máquina antigua en una VM que re-arquitecturizar la aplicación para que funcione en un contenedor.[62]
- **Cuándo usar Contenedores (como Amazon ECS/EKS):**
  - **Arquitecturas de Microservicios:** Es el caso de uso por excelencia. Descomponer una aplicación grande en una colección de servicios pequeños, independientes y que se pueden desplegar y escalar por separado es mucho más eficiente con contenedores.[10]
  - **Agilidad y DevOps:** Cuando el objetivo es acelerar los ciclos de desarrollo y despliegue. Los contenedores se integran perfectamente en pipelines de CI/CD, permitiendo construir, probar y desplegar código de forma automática y fiable.[30]
  - **Portabilidad y Entornos Híbridos:** Cuando se quiere evitar la dependencia de un único proveedor de nube (*vendor lock-in*). Una aplicación contenerizada

puede ejecutarse en AWS, Azure, Google Cloud o en un centro de datos propio con cambios mínimos o nulos.[30]

- **Máxima Eficiencia de Recursos:** Cuando se busca optimizar al máximo los costes de infraestructura. La ligereza de los contenedores permite ejecutar muchas más aplicaciones en el mismo hardware en comparación con las VMs.[59]

4.3. Así Funciona AWS (A Vistazo de Pájaro): Conectando los Puntos

Ahora que tienes los conceptos claros, puedes entender los servicios de computación de AWS no como una caja negra, sino como la implementación a escala de los principios que hemos estudiado.

- **Amazon EC2 (Elastic Compute Cloud):** Es el servicio de **Infraestructura como Servicio (IaaS)** de AWS. Cuando lanzas una "instancia EC2", lo que estás haciendo es pedirle a AWS que te aprovisione una **máquina virtual** en su gigantesca infraestructura global de hipervisores.[63] Tú eliges el sistema operativo (Amazon Linux, Ubuntu, Windows Server, etc.), el tamaño de la VM (vCPUs, RAM) y eres responsable de todo lo que ocurre dentro de ella: instalar software, aplicar parches de seguridad, configurar la red, etc..[65]
  EC2 es la materialización directa de los conceptos de virtualización del Capítulo 1.
- **Amazon ECS (Elastic Container Service) y EKS (Elastic Kubernetes Service):** Son servicios de **Contenedores como Servicio (CaaS)**. Su función no es proporcionar un contenedor, sino **orquestar** la ejecución de cientos o miles de contenedores Docker a escala.[65]
  - **ECS:** Es la solución de orquestación propietaria de AWS. Es más sencilla de usar y está profundamente integrada con otros servicios de AWS como IAM, VPC y CloudWatch.[65]
  - **EKS:** Es el servicio de Kubernetes gestionado de AWS. Kubernetes es el orquestador de contenedores de código abierto que se ha convertido en el estándar de la industria. EKS es más complejo que ECS, pero ofrece más potencia, flexibilidad y la portabilidad del ecosistema Kubernetes.[65]
  - **El Modelo de Responsabilidad Compartida:** Con ECS y EKS, tú te encargas de definir tus aplicaciones como contenedores (usando Dockerfiles) y de decirle al orquestador cómo quieres que se ejecuten (cuántas réplicas, qué puertos exponer, etc.). AWS se encarga de la compleja tarea de gestionar el "plano de control" del orquestador. Además, con **AWS Fargate**, AWS puede incluso abstraer por completo los servidores subyacentes. Tú solo pides ejecutar un contenedor con una cierta cantidad de CPU y memoria, y AWS se encarga de todo lo demás. Este es un modelo *serverless*. **ECS, EKS y Fargate son la materialización a escala de los conceptos de contenedores y redes del Capítulo 2 y 3.**

La siguiente tabla es el resumen final de nuestro viaje, conectando cada concepto fundamental que has aprendido con su servicio equivalente en AWS. Este mapa mental será

tu guía al iniciar el curso de AWS Cloud Foundations.

Concepto Fundamental	Servicio Equivalente en AWS	Nivel de Abstracción
Máquina Virtual (VM)	**Amazon EC2** (Elastic Compute Cloud)	Infraestructura como Servicio (IaaS)
Contenedor Docker	Se ejecuta en **Amazon ECS** o **Amazon EKS**	Contenedor como Servicio (CaaS)
Registro de Imágenes (Docker Hub)	**Amazon ECR** (Elastic Container Registry)	Plataforma como Servicio (PaaS)
Red Virtual Privada	**Amazon VPC** (Virtual Private Cloud)	Infraestructura como Servicio (IaaS)
Ejecución sin Servidor	**AWS Fargate** (para ECS/EKS)	Serverless / CaaS

4.4. Conclusión: ¡Listos para Conquistar la Nube!

Hemos completado un viaje intenso pero revelador. Hemos descendido hasta el núcleo del sistema operativo para entender cómo se gestiona un proceso, hemos ascendido a la capa del hipervisor para ver cómo gestiona un sistema operativo completo, y hemos llegado a la cima de la eficiencia con el motor de contenedores, que gestiona aplicaciones aisladas.
Este conocimiento no es trivial. Ahora posees el vocabulario y los modelos mentales para comprender no solo los servicios de AWS, sino los de cualquier proveedor de nube. Cuando en el próximo curso oigas hablar de "lanzar una instancia EC2", sabrás que en el fondo se está instanciando una VM en un hipervisor de Tipo 1. Cuando te hablen de "desplegar un servicio en ECS con Fargate", entenderás que se está pidiendo a un orquestador que ejecute tus contenedores Docker en una infraestructura gestionada, aprovechando la ligereza y portabilidad que has practicado.
Ya no ves solo los nombres de los servicios; entiendes los principios de ingeniería que los hacen posibles. Aborda el curso de AWS Cloud Foundations con la confianza de que tienes una base sólida y profunda sobre la cual construir conocimientos más avanzados de arquitectura, seguridad, redes y operaciones en la nube. El camino para convertirte en un profesional de la nube de alto nivel ha comenzado. ¡Adelante!

Obras citadas

1. Componentes del Sistema Operativo., fecha de acceso: septiembre 16, 2025, https://so.momrach.es/componentes-del-sistema-operativo/
2. Componentes de un Sistema Operativo - Zetra Services, fecha de acceso: septiembre 16, 2025, https://zetra.es/componentes-de-un-sistema-operativo/
3. Sistema operativo - Wikipedia, la enciclopedia libre, fecha de acceso: septiembre 16, 2025, https://es.wikipedia.org/wiki/Sistema_operativo
4. ¿Qué es el kernel y por qué es el corazón de tu sistema operativo? - Cibersafety,

fecha de acceso: septiembre 16, 2025, https://cibersafety.com/que-es-el-kernel-funciones/
5. ¿Qué es el kernel? Tareas del kernel del sistema operativo - IONOS, fecha de acceso: septiembre 16, 2025, https://www.ionos.com/es-us/digitalguide/servidores/know-how/que-es-el-kernel/
6. tema 3: el núcleo de un sistema operativo - UAL, fecha de acceso: septiembre 16, 2025, https://www.ual.es/~rguirado/so/tema3.pdf
7. ¿Qué son los hipervisores? - IBM, fecha de acceso: septiembre 16, 2025, https://www.ibm.com/mx-es/think/topics/hypervisors
8. The Evolution of Infrastructure: How We Got to Containers | Capital ..., fecha de acceso: septiembre 16, 2025, https://www.capitalone.com/tech/software-engineering/evolution-of-infrastructure-how-we-got-to-containers/
9. The Evolution of Virtualization Platforms: The Rise of Managed Services and Local Providers' Edge Against Hyperscalers | CNCF, fecha de acceso: septiembre 16, 2025, https://www.cncf.io/blog/2025/07/18/the-evolution-of-virtualization-platforms-the-rise-of-managed-services-and-local-providers-edge-against-hyperscalers/
10. Contenedores frente a VMs (máquinas virtuales) - Google Cloud, fecha de acceso: septiembre 16, 2025, https://cloud.google.com/discover/containers-vs-vms?hl=es-419
11. The Evolution from Virtual Machines to Containers - Scaler Topics, fecha de acceso: septiembre 16, 2025, https://www.scaler.com/topics/docker/virtual-machine-and-container/
12. ¿Qué es un hipervisor? - Explicación de los hipervisores - AWS, fecha de acceso: septiembre 16, 2025, https://aws.amazon.com/es/what-is/hypervisor/
13. www.nutanix.com, fecha de acceso: septiembre 16, 2025, https://www.nutanix.com/es/info/hypervisor#:~:text=Un%20hipervisor%20es%20un%20software,hacer%20posible%20la%20cloud%20computing.
14. Hipervisor - Wikipedia, la enciclopedia libre, fecha de acceso: septiembre 16, 2025, https://es.wikipedia.org/wiki/Hipervisor
15. Hipervisores: definición, tipos y soluciones - Stackscale, fecha de acceso: septiembre 16, 2025, https://www.stackscale.com/es/blog/hipervisores/
16. Ciberseguridad 101: Hipervisor | Illumio, fecha de acceso: septiembre 16, 2025, https://www.illumio.com/es-mx/cybersecurity-101/hypervisor
17. Evaluación del rendimiento del sistema - IBM, fecha de acceso: septiembre 16, 2025, https://www.ibm.com/docs/es/cics-ts/6.x?topic=techniques-assessing-performance-your-system
18. Métricas de rendimiento para máquinas virtuales - IBM, fecha de acceso: septiembre 16, 2025, https://www.ibm.com/docs/es/storage-insights?topic=metrics-performance-virtual-machines
19. Lista de comprobación: Medición del rendimiento en Hyper-V ..., fecha de acceso:

septiembre 16, 2025, https://learn.microsoft.com/es-es/biztalk/technical-guides/checklist-measuring-performance-on-hyper-v
20. VMware vs VirtualBox: What's the Difference?, fecha de acceso: septiembre 16, 2025, https://www.shiksha.com/online-courses/articles/vmware-vs-virtualbox-whats-the-difference-blogId-151329
21. VirtualBox vs VMware: Which Virtualization Tool is Right for You? - Redwood Compliance, fecha de acceso: septiembre 16, 2025, https://www.redwoodcompliance.com/virtualbox-vs-vmware-which-virtualization-tool-is-right-for-you/
22. VirtualBox vs VMware: Overview with Key Differences [2025], fecha de acceso: septiembre 16, 2025, https://www.net-usb.com/virtual-usb/virtualbox-vs-vmware/
23. VMware vs VirtualBox: What's The Difference? - InterviewBit, fecha de acceso: septiembre 16, 2025, https://www.interviewbit.com/blog/vmware-vs-virtualbox/
24. www.redwoodcompliance.com, fecha de acceso: septiembre 16, 2025, https://www.redwoodcompliance.com/virtualbox-vs-vmware-which-virtualization-tool-is-right-for-you/#:~:text=enterprise%2Dgrade%20environments.-,Advanced%20Features,and%20small%20business%20use%20cases.
25. Difference between VMware and VirtualBox - GeeksforGeeks, fecha de acceso: septiembre 16, 2025, https://www.geeksforgeeks.org/operating-systems/difference-between-vmware-and-virtualbox/
26. Hipervisor: ¿qué es, cómo funciona y qué tipos hay? - Red Hat, fecha de acceso: septiembre 16, 2025, https://www.redhat.com/es/topics/virtualization/what-is-a-hypervisor
27. Contenedores frente a máquinas virtuales: Una comparación ..., fecha de acceso: septiembre 16, 2025, https://www.datacamp.com/es/blog/containers-vs-virtual-machines
28. aws.amazon.com, fecha de acceso: septiembre 16, 2025, https://aws.amazon.com/es/compare/the-difference-between-containers-and-virtual-machines/#:~:text=Los%20contenedores%20virtualizan%20el%20sistema,de%20hardware%20de%20manera%20eficaz.
29. What is a Container? - Docker, fecha de acceso: septiembre 16, 2025, https://www.docker.com/resources/what-container/
30. ¿Qué son los contenedores? - IBM, fecha de acceso: septiembre 16, 2025, https://www.ibm.com/es-es/think/topics/containers
31. Contenerización vs. Virtualización: 7 diferencias técnicas - Trianz, fecha de acceso: septiembre 16, 2025, https://www.trianz.com/es/insights/containerization-vs-virtualization
32. Guía de Docker para principiantes: cómo crear tu primera aplicación Docker, fecha de acceso: septiembre 16, 2025, https://www.freecodecamp.org/espanol/news/guia-de-docker-para-principiantes-como-crear-tu-primera-aplicacion-docker/
33. Tutorial de Docker: introducción a la popular plataforma de contenedores -

IONOS, fecha de acceso: septiembre 16, 2025, https://www.ionos.com/es-us/digitalguide/servidores/configuracion/tutorial-docker-instalacion-y-primeros-pasos/
34. Introducción a Docker: Desarrollo y despliegue con contenedores - OpenWebinars, fecha de acceso: septiembre 16, 2025, https://openwebinars.net/blog/introduccion-a-docker-desarrollo-y-despliegue-con-contenedores/
35. Imagen de Docker y contenedor: diferencia entre tecnologías de implementación de aplicaciones - AWS, fecha de acceso: septiembre 16, 2025, https://aws.amazon.com/es/compare/the-difference-between-docker-images-and-containers/
36. Guía Rápida de Docker: Comandos Esenciales y Conceptos Clave para Desarrolladores, fecha de acceso: septiembre 16, 2025, https://www.kranio.io/blog/guia-rapida-de-docker-comandos-y-conceptos-claves
37. ¿Qué es un Docker Image? - IONOS, fecha de acceso: septiembre 16, 2025, https://www.ionos.com/es-us/digitalguide/servidores/know-how/docker-image/
38. Qué es DockerFile - OpenWebinars, fecha de acceso: septiembre 16, 2025, https://openwebinars.net/blog/que-es-dockerfile/
39. Dominando la gestión de contenedores con Dockerfiles y Docker Compose, fecha de acceso: septiembre 16, 2025, https://diegochavez-dc.medium.com/dominando-la-gesti%C3%B3n-de-contenedores-con-dockerfiles-y-docker-compose-330ea4b7c3f6
40. Docker para principiantes: Guía práctica de contenedores | DataCamp, fecha de acceso: septiembre 16, 2025, https://www.datacamp.com/es/tutorial/docker-tutorial
41. Cómo aprender Docker desde cero: Guía para profesionales de los datos - DataCamp, fecha de acceso: septiembre 16, 2025, https://www.datacamp.com/es/blog/learn-docker
42. Los 18 mejores comandos Docker para construir, ejecutar y gestionar contenedores, fecha de acceso: septiembre 16, 2025, https://www.datacamp.com/es/blog/docker-commands
43. www.f5.com, fecha de acceso: septiembre 16, 2025, https://www.f5.com/es_es/glossary/nginx#:~:text=NGINX%20es%20un%20open%20source,el%20m%C3%A1ximo%20rendimiento%20y%20estabilidad.
44. ¿Qué Es NGINX Y Cómo Funciona? - Guía Completa - Hostinger, fecha de acceso: septiembre 16, 2025, https://www.hostinger.com/es/tutoriales/que-es-nginx
45. ¿Por qué Nginx es tan popular? - Palentino Blog, fecha de acceso: septiembre 16, 2025, https://www.palentino.es/blog/por-que-nginx-es-tan-popular/
46. Nginx qué es, ventajas y desventajas - Hostinet, fecha de acceso: septiembre 16, 2025, https://www.hostinet.com/formacion/vps-servidores/nginx-que-es-y-como-funciona/
47. ¿Qué es Nginx y cómo funciona? Entérate de todo ahora - Hackio, fecha de

acceso: septiembre 16, 2025, https://www.hackio.com/blog/que-es-nginx
48. How to create a reverse proxy with Nginx - SW Hosting, fecha de acceso: septiembre 16, 2025, https://www.swhosting.com/en/comunidad/manual/how-to-create-a-reverse-proxy-with-nginx
49. NGINX Reverse Proxy | NGINX Documentation, fecha de acceso: septiembre 16, 2025, https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/
50. NGINX Reverse Proxy y Granja de Servidores IPv6 Only: Conectividad Web Eficiente - LACNIC, fecha de acceso: septiembre 16, 2025, https://www.lacnic.net/innovaportal/file/6684/1/lacnic-conectividad_web_eficiente.pdf
51. ¿Qué son las redes en Docker? | KeepCoding Bootcamps, fecha de acceso: septiembre 16, 2025, https://keepcoding.io/blog/que-son-las-redes-en-docker/
52. Docker Networking - Basics, Network Types & Examples - Spacelift, fecha de acceso: septiembre 16, 2025, https://spacelift.io/blog/docker-networking
53. Introducción a las redes en docker. Enlazando contenedores docker - PLEDIN 3.0, fecha de acceso: septiembre 16, 2025, https://www.josedomingo.org/pledin/2020/02/redes-en-docker/
54. Uso de Docker Overlay Networks: Guía de configuración | by Be Tech! with Santander, fecha de acceso: septiembre 16, 2025, https://medium.com/be-tech-with-santander/uso-de-docker-overlay-networks-gu%C3%ADa-de-configuraci%C3%B3n-f784c309fe7b
55. Networking with overlay networks - Docker Docs, fecha de acceso: septiembre 16, 2025, https://docs.docker.com/engine/network/tutorials/overlay/
56. Redes de Docker: Personalizadas y Casos Reales - Q2B Studio, fecha de acceso: septiembre 16, 2025, https://www.q2bstudio.com/nuestro-blog/20530/redes-de-docker-personalizadas-y-casos-reales
57. Docker: Contenedores en redes independientes y como conectarlos - Tira que libras, fecha de acceso: septiembre 16, 2025, https://blog.tiraquelibras.com/?p=887
58. Cómo probar la conectividad entre contenedores de Docker - LabEx, fecha de acceso: septiembre 16, 2025, https://labex.io/es/tutorials/docker-how-to-test-connectivity-between-docker-containers-411613
59. ¿Qué es la contenerización? ¿Cuáles son los beneficios? - Veritas, fecha de acceso: septiembre 16, 2025, https://www.veritas.com/es/es/information-center/containerization
60. ¿Qué es la contenerización? - IBM, fecha de acceso: septiembre 16, 2025, https://www.ibm.com/es-es/think/topics/containerization
61. ¿Qué son los contenedores? - NetApp, fecha de acceso: septiembre 16, 2025, https://www.netapp.com/es/devops/what-are-containers/
62. Diferencias entre los contenedores y las máquinas virtuales - Red Hat, fecha de acceso: septiembre 16, 2025, https://www.redhat.com/es/topics/containers/containers-vs-vms

63. Todavía un poco confundido acerca de ECS vs EC2 : r/aws - Reddit, fecha de acceso: septiembre 16, 2025, https://www.reddit.com/r/aws/comments/180ad9n/still_a_bit_confused_about_ecs_vs_ec2/?tl=es-es
64. ECS vs EKS vs EC2 on AWS: Where Should You Deploy Your Containers? | CloudForecast, fecha de acceso: septiembre 16, 2025, https://www.cloudforecast.io/blog/ecs-vs-eks/
65. ECS vs EKS: ¿Qué servicio de contenedores de AWS es el ..., fecha de acceso: septiembre 16, 2025, https://www.datacamp.com/es/blog/ecs-vs-eks
66. ECS vs EKS vs EC2: Choosing the Best AWS Service for Docker Deployment - YouTube, fecha de acceso: septiembre 16, 2025, https://www.youtube.com/watch?v=PQidBErmQIo
67. Contenedores en AWS: ECS, EC2, Fargate, EKS y ECR | Tutorial en ESPAÑOL! - YouTube, fecha de acceso: septiembre 16, 2025, https://www.youtube.com/watch?v=iZC_8Jo2P70