Técnicas Avanzadas en la Construcción de Software 2025

[Arquitectura de APIs y Comunicación Web 4](#_Toc202134368)

[Modelos de comunicación 4](#_Toc202134369)

[Sockets 4](#_Toc202134370)

[RPC / RMI 4](#_Toc202134371)

[Fundamentos de REST 4](#_Toc202134372)

[Richardson Maturity Model 5](#_Toc202134373)

[Seguridad e Idempotencia de Verbos HTTP 6](#_Toc202134374)

[HATEOAS - Formatos de Datos para APIs 6](#_Toc202134375)

[Códigos HTTP 7](#_Toc202134376)

[Alternativas 7](#_Toc202134377)

[SOAP 7](#_Toc202134378)

[Arquitectura Web y Frontend 8](#_Toc202134379)

[Frontend 8](#_Toc202134380)

[Modelo de Objetos del Documento (DOM) y CSSOM 8](#_Toc202134381)

[Compatibilidad entre Navegadores 8](#_Toc202134382)

[Performance Web 9](#_Toc202134383)

[Manejo asincrónico en JS 9](#_Toc202134384)

[Navegación Client-Side y el API de History 9](#_Toc202134385)

[Métodos de Estilizado y Herramientas Frontend 9](#_Toc202134386)

[Module Bundlers 10](#_Toc202134387)

[CORS (Cross-Origin Resource Sharing) 10](#_Toc202134388)

[Técnicas de rendering 10](#_Toc202134389)

[Infraestructura 11](#_Toc202134390)

[Métricas 11](#_Toc202134391)

[Performance 11](#_Toc202134392)

[Escalabilidad 11](#_Toc202134393)

[Alta Disponibilidad 11](#_Toc202134394)

[Estrategias de Despliegue 11](#_Toc202134395)

[Balanceo de Carga 12](#_Toc202134396)

[Failover / Transparent Failover 12](#_Toc202134397)

[CDN (Content Delivery Network) 12](#_Toc202134398)

[Hardware vs Virtualización 12](#_Toc202134399)

[Cloud Computing 13](#_Toc202134400)

[Modelos Cloud 13](#_Toc202134401)

[Tendencias Nuevas 13](#_Toc202134402)

[Microservicios 14](#_Toc202134403)

[Serverless 16](#_Toc202134404)

[Ejecución (FaaS Lifecycle) 17](#_Toc202134405)

[Serverless Framework 17](#_Toc202134406)

[Docker y Contenerización 18](#_Toc202134407)

[Contenedores vs VMs 19](#_Toc202134408)

[Ciclo de Vida de Docker 19](#_Toc202134409)

[Dockerfile 19](#_Toc202134410)

[Data Volumes 20](#_Toc202134411)

[Docker Compose 20](#_Toc202134412)

[Comandos Básicos de Docker 20](#_Toc202134413)

[Observabilidad 22](#_Toc202134414)

[Pilares de observabilidad 22](#_Toc202134415)

[APM 22](#_Toc202134416)

[Tracing distribuido 22](#_Toc202134417)

[OpenTelemetry 23](#_Toc202134418)

[Bases de Datos NoSQL 24](#_Toc202134419)

[Tipos de Bases de Datos NoSQL 24](#_Toc202134420)

[Teorema CAP 24](#_Toc202134421)

[Modelos de Consistencia: ACID vs BASE 25](#_Toc202134422)

[ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) 25](#_Toc202134423)

[BASE (Basically Available, Soft-State, Eventual Consistency) 25](#_Toc202134424)

[Particionamiento (Sharding) 25](#_Toc202134425)

[Replicación 26](#_Toc202134426)

[Estrategias de Optimización en Bases de Datos 26](#_Toc202134427)

[DevOps 27](#_Toc202134428)

[CI/CD (Integración y Entrega Continua) 27](#_Toc202134429)

[Infrastructure as Code (IaC) 27](#_Toc202134430)

[Kubernetes 27](#_Toc202134431)

[Git Branch Model - GitFlow 28](#_Toc202134432)

[GitOps 28](#_Toc202134433)

[Site Reliability Engineering (SRE) 29](#_Toc202134434)

[Error Budget 29](#_Toc202134435)

[Métricas Clave 29](#_Toc202134436)

[Principios y Prácticas 30](#_Toc202134437)

[Pasos para implementar SER 30](#_Toc202134438)

[Inteligencia Artificial 31](#_Toc202134439)

[IA Generativa 31](#_Toc202134440)

[Modelos de Lenguaje Grandes (LLMs) 31](#_Toc202134441)

[Prompt Engineering 33](#_Toc202134442)

[Agent-to-Agent (A2A) 33](#_Toc202134443)

# Arquitectura de APIs y Comunicación Web

Una **Application Programming Interface (API)** permite que distintos sistemas o procesos se comuniquen entre sí.

## Modelos de comunicación

### Sockets

Son una abstracción de bajo nivel para permitir la **comunicación entre procesos**, ya sea en la misma máquina o entre máquinas distintas (por red).

* Comunicación a bajo nivel (TCP/UDP).
* Necesita protocolo propio, manejo manual de errores y seguridad.

### RPC / RMI

* RPC (remote procedure call): mecanismo que permite que un programa ejecute funciones en otro sistema como si fueran locales. Es SÍNCRONO.
* RMI (remote method invocation): implementación de RPC específica de **Java**.
* Problemas: alto acoplamiento, dependencias de lenguaje, errores como excepciones.

## Fundamentos de REST

REST (Representational State Transfer) es un estilo arquitectónico para sistemas distribuidos, basado en HTTP. Sus principios clave incluyen:

* **Arquitectura Cliente-Servidor:** El cliente y el servidor son independientes.
* **Stateless (Sin Estado):** Cada solicitud del cliente al servidor contiene toda la información necesaria para que el servidor la entienda y procese, sin que el servidor dependa del contexto de solicitudes previas. Esto mejora la escalabilidad.
* **Cache Aware (Conciencia de Caché):** Las respuestas pueden ser marcadas como cacheables para mejorar la eficiencia y el rendimiento. Se pueden usar encabezados como no-store, no-cache, private/public, max-age y if modified since.
* **Interfaz Uniforme:** Se utiliza una interfaz coherente para la interacción entre componentes, facilitando la comprensión y la estandarización. Esto incluye el uso de sustantivos para definir recursos (similar a un filesystem) y verbos HTTP.
* **Sistema de Capas:** Permite la adición de capas intermedias (proxies, balanceadores de carga) sin afectar la comunicación entre cliente y servidor.
* **Código Bajo Demanda (Opcional):** El servidor puede extender la funcionalidad del cliente enviando código ejecutable.

### Richardson Maturity Model

Este modelo describe la evolución de los servicios web hacia una arquitectura RESTful completa, en cuatro niveles:

* **Nivel 0: Swamp of POX (Plain Old XML):** Servicios que usan HTTP solo como transporte, sin aprovechar sus características, a menudo con un solo endpoint y mensajes complejos.
* **Nivel 1: Resources (Recursos):** Se empiezan a usar URI para identificar recursos individuales. "Uso de sustantivos para definir recursos. Similar a un filesystem."
* **Nivel 2: HTTP Verbs / Status Codes (Verbos HTTP / Códigos de Estado):** Se utilizan los verbos HTTP estándar (GET, POST, PUT, PATCH, DELETE, OPTIONS) para realizar operaciones sobre los recursos y los códigos de estado HTTP para indicar el resultado de la operación.
  + **GET:** "Traer recurso"
  + **POST:** "Crear recurso"
  + **PUT:** "Modificar recurso" (modifica todo el recurso)
  + **PATCH:** "Modificar parcialmente un recurso"
  + **DELETE:** "Eliminar recurso"
  + **OPTIONS:** "Operaciones sobre un recurso" (indica qué verbos están permitidos).
* **Nivel 3: HATEOAS (Hypermedia As The Engine Of Application State):** El cliente interactúa dinámicamente con la aplicación web mediante la información de hipermedia proporcionada en las respuestas del servidor.

### Seguridad e Idempotencia de Verbos HTTP

**Idempotente:** "el server queda en el mismo estado si hacemos un mismo request 1 o N veces".

Los verbos HTTP se clasifican según su seguridad e idempotencia:

* **Seguro:** "Sin efecto de lado, aka read only". El servidor permanece en el mismo estado después de la operación.
* **GET:** Seguro, Idempotente.
* **OPTIONS:** Seguro, Idempotente.
* **PUT:** No Seguro, Idempotente.
* **PATCH:** No Seguro, No Idempotente.
* **DELETE:** No Seguro, Idempotente.
* **POST:** No Seguro, No Idempotente.

### HATEOAS - Formatos de Datos para APIs

Para la comunicación entre sistemas, se utilizan diversos formatos de serialización de datos:

* **JSON:** "human readable, se usa en JS y APIs, esquema de autovalidación." Es ligero y ampliamente adoptado.
* **YAML:** "human readable, se usa en swagger, se pueden tener referencias, tipos de datos definidos custom, sin esquema de validación."
* **Binary encoding (ej. Protocol Buffers):** "Rápida serialización/deserialización (es más ligero pasar datos por el caño)." "No importa la plataforma (x86, x64 da igual)." "Serializado binario => not human readable." Ofrecen mayor eficiencia en la transferencia de datos.
* **GraphQL:** Brilla en el mundo del frontend porque permite al cliente especificar exactamente los datos que necesita, evitando la sobre-obtención o sub-obtención de datos.

HATEOAS no está estandarizado, lo que dificulta su adopción general. Existen múltiples implementaciones distintas, lo que complica la interoperabilidad. Por eso es considerado opcional y poco usado en la práctica.

### Códigos HTTP

* 2xx: OK, Created
* 4xx: Bad Request, Unauthorized, Not Found
* 5xx: Server Error, Timeout

### Alternativas

#### GraphQL

* APIs orientadas a grafos.
* Queries tipadas y declarativas.
* Evita over-fetching y N+1 queries.
* Mutaciones, introspección, caché parcial.
* No cachea fácilmente POST.
* No soluciona problemas de caching, por lo que no asegura buena performance.

#### gRPC

* RPC moderno con **Protocol Buffers**.
* Funciona sobre **HTTP/2**.
* Full-duplex streaming, autenticación, tracing, flow control.
* Ideal para microservicios y alta performance.

## SOAP

Protocolo XML formal, extensible, agnóstico a transporte (HTTP, SMTP).

Usa WSDL para describir servicios.

# Arquitectura Web y Frontend

## Frontend

Es la parte visible al usuario: puede ser una web app, mobile app, o desktop app. Una página web se construye principalmente con:

* **HTML:** Estructura el contenido.
* **CSS:** "Colores, fuentes, tamaños y posiciones, todo puede ser definido mediante este lenguaje" para dar estilo y apariencia.
* **JavaScript (JS):** "un lenguaje de programación que maneja el browser." Inicialmente para "agregar comportamiento", hoy en día "se puede manipular el DOM íntegramente con JS" y "se usa para darle interactividad a las páginas web".

**¿Qué pasa al ingresar una URL?**

1. **DNS**: se resuelve la IP del dominio.
2. **TCP**: se establece conexión.
3. **HTTP**: se hace la solicitud.
4. **HTML**: el navegador renderiza la respuesta.

### Modelo de Objetos del Documento (DOM) y CSSOM

* **DOM (Document Object Model):** "Representación interna del browser al leer el HTML." Es una interfaz de programación para documentos HTML y XML.
* **CSSOM (CSS Object Model):** Representación del CSS.
* **Render Tree:** Combina el DOM y el CSSOM para mostrar el contenido en la pantalla.

### Compatibilidad entre Navegadores

"Cada browser muestra lo que quiere mediante el DOM, entonces hay que probar mi app en todos los browsers en los que quiera que corra mi APP". Es crucial verificar que la aplicación funcione correctamente en los navegadores más populares para una "plataforma web de uso masivo".

Herramientas como Lighthouse permiten auditar la performance y accesibilidad del sitio. Otras como Amplitud y BrowserStack ayudan a evaluar el comportamiento del sitio en diferentes navegadores y dispositivos.

### Performance Web

Indicadores clave:

* **TTFB** (Time to First Byte): ideal < 0.8s
* **LCP** (Largest Contentful Paint): ideal < 2.5s

Mejoras comunes:

* Minimizar HTML/CSS
* Cargar JS async o defer
* Cargar solo lo necesario primero (critical path)
* Usar herramientas como **Lighthouse**

### Manejo asincrónico en JS

* **AJAX** (XMLHttpRequest)
* **jQuery callbacks**
* **Promises**
* **Async/Await**

Permiten actualizar partes de la página sin recargar todo.

## Navegación Client-Side y el API de History

La navegación hacia adelante y atrás en la historia del navegador no funciona automáticamente con la navegación client-side en JS. Librerías como React Router resuelven esto utilizando el API de History.

Las SPA (Single Page Applications) cargan una única página HTML y luego actualizan dinámicamente el contenido. Mejoran la UX pero se llevan mal con el SEO y requieren mayor complejidad en el frontend.

## Métodos de Estilizado y Herramientas Frontend

* **Mobile First:** Priorizar el diseño para dispositivos móviles.
* **Frameworks CSS:** Bootstrap, Material Design.
* **Utility-first CSS:** Tailwind.
* **Preprocesadores CSS:** SCSS (Sass), LESS.
* **CSS-in-JS:** Styled components.

## Module Bundlers

Herramientas como Webpack que permiten empaquetar y optimizar módulos de JavaScript y otros recursos para el navegador. Sus características incluyen:

* Soporte para ES6 modules.
* Soporte a través de loaders (CSS, JS).
* Compatibilidad cross-browser (con target fijado).
* Polyfills para características modernas de JS.
* Minificación y "uglifying" (ofuscación) del código.
* Hot reload para desarrollo rápido.

## CORS (Cross-Origin Resource Sharing)

Es un mecanismo de seguridad del navegador que restringe las solicitudes HTTP de un origen diferente al de la URL actual. Es fundamental para la seguridad en las interacciones entre distintos dominios.

## Técnicas de rendering

* **CSR (Client Side Rendering)**: todo se carga en el cliente.
* **SSR (Server Side Rendering)**: el server renderiza HTML por cada request.
* **SSG (Static Site Generation)**: HTML generado al momento del build.
* **ISG (Incremental Static Generation)**: mezcla de SSR y SSG (Next.js).
* **RSC (React Server Components)**: render parcial en server con control del cliente.

# Infraestructura

La infraestructura es el conjunto de recursos físicos y virtuales que dan soporte a una aplicación:

* Servidores físicos / virtuales.
* Contenedores de aplicación.
* Redes.
* Bases de datos.

## Métricas

### Performance

Cantidad de trabajo útil realizado. Se mide por:

* Tiempo de respuesta del sistema y del usuario.
* Saturación → puede desencadenar fallas en cadena.

### Escalabilidad

Capacidad de manejar más carga manteniendo la performance.

* **Vertical**: aumentar recursos de un servidor.
* **Horizontal**: sumar servidores (requiere clustering y balanceo).

### Alta Disponibilidad

Capacidad de continuar operando ante fallas:

* Se mide con “9s” (ej.: 99.99%).
* Fórmula: **Availability = ATTF / (ATTF + ATTR)**.
* Requiere: Load balancing, Failover, Cluster activo/pasivo.

## Estrategias de Despliegue

* **Canary**: probar en un nodo antes de liberar a todos.
* **Blue/Green**: dos versiones conviven, luego se cambia.
* **A/B Testing**: solo una parte de usuarios ve la nueva versión.
* **Disabled Features**: funcionalidades cargadas pero no habilitadas.

## Balanceo de Carga

Distribuye tráfico entre múltiples servidores. Tipos:

* **Round Robin** (por turnos)
* **Por carga**: menos conexiones, menos latencia
* **IP Hash**: misma IP → mismo servidor (problemático si el server cae)

**Consideraciones**

* **Stateless**: más simple, no requiere replicación
* **Stateful**: necesita mantener sesiones → más complejidad

**Replicación de Sesiones**

* **Triggers**: SET, GET, ACCESS (impactan en performance)
* **Granularidad**: Atributo sucio, toda sesión o campos puntuales

### Failover / Transparent Failover

* **Failover**: redirigir tráfico ante caídas
* **Transparent**: el cliente no nota la caída

## CDN (Content Delivery Network)

Red de servidores distribuidos geográficamente para servir contenido estático.

* Menor latencia
* Menor carga del servidor principal
* Mejora el caching

## Hardware vs Virtualización

**Ventajas de virtualizar**:

* Menor downtime
* Mayor eficiencia
* Administración centralizada
* Acelera aprovisionamiento
* Soporte para recuperación ante desastres

## Cloud Computing

Acceso **on-demand** a recursos virtualizados:

* **Características**: self-service, pooling, escalabilidad, pago por uso
* **Ventajas**: elasticidad, menor costo inicial, geodistribución
* **Desventajas**: menos control, vendor lock-in, costos variables

### Modelos Cloud

* **IaaS**: Infraestructura (servidores, redes, VMs)
  + Ej.: AWS EC2, Azure, GCP
* **PaaS**: Plataforma (runtimes, BDs, herramientas)
  + Ej.: Heroku, Google App Engine
* **SaaS**: Software como servicio (acceso web, suscripción)
  + Ej.: Gmail, Trello, Slack

### Tendencias Nuevas

* **FaaS** (Function as a Service / Serverless): código que se ejecuta sin servidor dedicado (Ej.: AWS Lambda)
* **BaaS** (Backend as a Service): servicios listos como Auth0, Firebase

# Microservicios

Las **aplicaciones monolíticas** son aquellas construidas como un único bloque que agrupa todas las funcionalidades del sistema. Este enfoque puede ser válido en etapas iniciales de un proyecto, pero a medida que crece, surgen limitaciones importantes.

**Ventajas del enfoque monolítico:**

* Simplicidad inicial de diseño y despliegue.
* Baja latencia al estar todo en el mismo proceso.
* Un único artefacto para deployar.
* Fácil integración entre componentes internos.

**Desventajas del monolito:**

* Deploys costosos y lentos: cualquier cambio implica recompilar y testear toda la app.
* Escalabilidad limitada en equipos: muchos desarrolladores tocando el mismo código.
* Tiempos largos de testing y build.
* Fallos en un módulo afectan a toda la aplicación (ej. un memory leak puede derribar todo).
* Dificultad para usar diferentes tecnologías/configuraciones para distintos módulos.
* Mayor complejidad para mantener y remover código legacy.
* Requiere coordinación simultánea de múltiples personas para cada deploy.

Los **microservicios** son pequeñas unidades independientes que implementan una funcionalidad específica y se comunican entre sí a través de APIs. Cada uno puede ser desarrollado, desplegado y escalado por separado.

**Ventajas:**

* Desacoplamiento: si un microservicio falla, no necesariamente cae todo el sistema (ej. búsqueda en Spotify).
* Equipos pequeños, más enfocados y autónomos.
* Posibilidad de desplegar funcionalidades sin afectar al resto.
* Facilidad para integrar solo ciertos servicios desde otras apps (ej. una app mobile accede solo a destacados y promociones).
* Uso de tecnologías diferentes por microservicio si se desea.
* Microservicios individuales más simples y mantenibles.

**Desventajas:**

* Overhead en comunicación y latencia: llamadas en cadena entre servicios pueden acumular timeouts.
* Mayor complejidad operativa: monitoreo, logging, tracing distribuido.
* Dificultad para definir límites claros entre responsabilidades.
* Dependencias entre equipos: para acceder a datos de otro micro, se debe coordinar.
* Riesgo de carga desbalanceada entre equipos.
* Más puntos de falla: cada red, servicio o integración puede fallar.

En resumen, los microservicios aportan **flexibilidad y escalabilidad**, pero a costa de **complejidad operativa**. La migración desde un monolito debe ser planificada, gradual y justificada estratégicamente ante el negocio.

# Serverless

Arquitectura que permite ejecutar código propio en infraestructura de terceros, sin administrar servidores. Utiliza:

* **BaaS (Backend as a Service)**: servicios listos (ej. Firebase, Auth0)
* **FaaS (Functions as a Service)**: código ejecutado como funciones en contenedores efímeros (ej. AWS Lambda)

El servidor existe pero es invisible al desarrollador. Está pensado para **modelos orientados a eventos**.

**¿Por qué usarlo?**

* **Simplicidad operativa**: sin administración de servidores
* **Alta escalabilidad automática**
* **Alta disponibilidad**
* **Costo bajo**: se paga solo lo usado
* **Alta confiabilidad** desde el comienzo
* **Tolerancia a fallos**
* **Soporte multilenguaje**

Ideal para startups, microservicios y funciones puntuales.

**Desventajas**

* Vendor lock-in
* Cold start
* Debugging complejo
* Tests de integración difíciles
* Límites de tiempo, memoria y tamaño
* Orquestación entre funciones puede ser compleja
* Observabilidad limitada (logging, tracing, etc.)

**¿Cuándo conviene Serverless?**

* Funciones pequeñas, event-driven
* Procesamiento esporádico
* Cuando el foco está en la lógica, no en la infraestructura

## Ejecución (FaaS Lifecycle)

1. Cold Start (si la función no estaba activa)
   1. El proveedor (AWS, GCP, etc.) levanta un contenedor nuevo, carga tu código y dependencias.
   2. Esta etapa puede tomar desde cientos de milisegundos hasta varios segundos.
2. Ejecución: una vez lista, la función se ejecuta con el evento que la disparó.
3. Stand-by corto
   1. Después de ejecutarse, el contenedor permanece activo un breve tiempo.
   2. Si llega otra invocación en ese intervalo, se evita el cold start (se llama warm start).
   3. Este tiempo de espera es corto y depende del proveedor.
4. Terminación
   1. Si no hay nuevas invocaciones, el contenedor se elimina automáticamente.
   2. Esto libera recursos y evita costos innecesarios.

## Serverless Framework

Herramienta open-source para gestionar aplicaciones serverless (multi proveedor).

**Pasos generales:**

1. Crear proyecto (serverless create)
2. Definir endpoints y handlers en archivo YML
3. Empaquetar (mvn clean install)
4. Test offline (serverless offline start)
5. Deploy (serverless deploy)
6. Logs / Clean-up (serverless logs / remove)

# Docker y Contenerización

Docker es una plataforma que permite empaquetar una aplicación y todas sus dependencias en un "container" aislado, garantizando que el software funcione de manera consistente en cualquier entorno.

* **Container:** "instancia runtime de una docker image". Es una "capa r/w agregada arriba de la imagen", "efímero" y "aislado del resto". Provee aislamiento a nivel de SO, compartiendo el kernel del host con otros contenedores, a diferencia de las VMs que tienen su propio kernel. Usa namespaces (vista privada de red, PID, mountpoints) y cgroups (recursos limitados).
* **Image:** "Templates read-only desde los que se crean containers". "Armada por un conjunto de capas que combinadas, conforman el filesystem". "Construida a partir de un Dockerfile". Cada capa es hasheada y cacheada, lo que optimiza el espacio en disco ya que las capas se pueden reutilizar.
* **Registry:** "almacena docker images", similar a un repositorio Git. Docker Hub es un ejemplo público.

Docker basa su manejo de imágenes y administración de contenedores en 2 tecnologías:

**Stackable Image Layers (Capas apilables)**

Las imágenes Docker se construyen como una pila de capas. Cada instrucción del Dockerfile (ej. RUN, COPY, ADD) crea una **nueva capa**.

* Las capas inferiores se **comparten** entre imágenes si son iguales.
* Esto **reduce espacio** y acelera los builds y pulls.

**Copy-on-Write (COW)**

Cuando un contenedor se crea desde una imagen, **no se copia toda la imagen**.  
Solo cuando el contenedor **modifica algo**, se crea una copia privada de esa parte.

* Minimiza el uso de disco.
* Permite que muchos contenedores compartan la misma imagen base sin duplicarla.

## Contenedores vs VMs

|  |  |  |
| --- | --- | --- |
|  | **Containers** | **VMs** |
| **Aislamiento** | A nivel de proceso (comparten kernel) | A nivel de SO. |
| **Peso** | Livianos | Pesados |
| **Arranque** | Rápido | Lento |
| **Eficiencia** | Alta | Baja |
| **Portabilidad** | Alta | Intermedia |

## Ciclo de Vida de Docker

* **Build:** Creación de imágenes a partir de Dockerfiles.
* **Deploy:** Despliegue de contenedores. Incluye estrategias como "Canary deploy" (despliegue gradual a un subconjunto de usuarios) y "Blue green deploy" (mantener dos entornos, uno viejo y uno nuevo, y cambiar el tráfico).
* **Run:** Ejecución de contenedores.
* **Monitor:** Monitoreo del rendimiento y salud de los contenedores. Es crucial para sistemas efímeros y distribuidos.

### Dockerfile

Es un archivo de texto que contiene las instrucciones para construir una imagen Docker.

* Cada comando en el Dockerfile crea una nueva capa en la imagen.
* "Capas jerárquicas: cambiar una capa baja, destruye el cache de las capas superiores".
* **CMD:** "Define un comando a ser ejecutado al crear un container", puede ser redefinido en runtime.
* **ENTRYPOINT:** Similar a CMD, pero permite pasar parámetros en runtime.

#### Mejores Prácticas de Dockerfile

* "Containers deben ser efímeros".
* "Reducir tamaño de las imágenes" utilizando .dockerignore, evitando paquetes innecesarios.
* "Cada container un concern" (un solo propósito por contenedor).
* "Minimizar la cantidad de capas" para reducir el tamaño y mejorar el rendimiento.

## Data Volumes

"Cuando se borra un container, cualquier dato escrito en el container, que no es almacenado en un data volume, es eliminado junto con el container."

* Un Data Volume es un "Directorio o archivo en el filesystem del host, montado directamente dentro de un container".
* "No son controlados por el storage driver".
* "Operan a la velocidad nativa del host".
* Permiten la persistencia de datos y que "Múltiples containers pueden compartir uno o más data volumes".
* Son ideales para almacenar datos persistentes como logs o archivos compartidos entre contenedores. Para bases de datos, se recomienda usar data volumes en lugar de almacenamiento interno del container.

## Docker Compose

"Herramienta que nos permite definir y correr aplicaciones docker multi containers". Permite "definir su arquitectura mediante código".

Se utiliza también para levantar entornos de desarrollo consistentes y simular clústeres en local. Por ejemplo, levantar frontend, backend y base de datos juntos, conectados por red.

## Comandos Básicos de Docker

* docker run: Crea y ejecuta un contenedor a partir de una imagen.
* docker ps -a: Lista todos los contenedores (activos e inactivos).
* docker logs -f [containerId]: Muestra los logs de un contenedor en tiempo real.
* docker rm $(docker ps -qa): Elimina todos los contenedores.
* docker stop [container ID]: Detiene un contenedor.
* docker start [container ID or name]: Inicia un contenedor detenido.
* docker exec -it [container ID] /bin/bash: Permite ejecutar un comando dentro de un contenedor en ejecución (acceso por terminal).
* docker build -t [repository:tag] [path]: Construye una imagen a partir de un Dockerfile.

# Observabilidad

La observabilidad es la capacidad de inferir el estado interno de un sistema a partir de sus salidas externas (logs, métricas, traces).

No es lo mismo que monitoreo: implica no solo recolectar métricas, logs y trazas, sino también cruzarlas y analizarlas para entender el estado interno del sistema. Herramientas comunes incluyen Prometheus, Grafana, ELK, y Jaeger.

Pilares de observabilidad

* Métricas: datos numéricos a lo largo del tiempo.
  + Tipos: COUNT, RATE, etc.
* Logs
* Tracing distribuido
* Visualización y alertas

## APM

Permite:

* Detectar cuellos de botella y errores
* Analizar queries lentas, throughput, stack traces
* Ver Apdex, tiempos de respuesta, agrupamiento de errores

## Tracing distribuido

Clave para arquitecturas de microservicios. Permite:

* Seguir una solicitud entre servicios
* Analizar latencias y dependencias
* Ver qué parte del sistema falló

**Conceptos:**

* **Trace**: conjunto de spans relacionados (operaciones)
* **Span**: unidad de trabajo, con atributos y relaciones

## OpenTelemetry

Framework estándar para instrumentación:

* Reemplaza OpenTracing + OpenCensus
* Vendor-agnostic
* Recoge: métricas, logs, trazas

Componentes:

* API / SDK / Exporters
* Procesadores
* Backends pluggables

Soportado por CNCF (como Kubernetes).

# Bases de Datos NoSQL

NoSQL surge como alternativa a las bases de datos relacionales tradicionales (SQL) para manejar grandes volúmenes de datos, alta concurrencia y requisitos de escalabilidad que no se adaptan bien al modelo relacional. "NoSQL es útil cuando tenés muchos datos y muchas requests, caso contrario => SQL". En NoSQL, el diseño se enfoca en los casos de uso y luego en el esquema ("Schema on read").

## Tipos de Bases de Datos NoSQL

* **Key-Value:** Los datos se almacenan como pares clave-valor (ej. Redis). "flexibilidad nula (solo busco por la clave) pero es altamente performante."
* **Wide Column Stores:** Extensión del modelo clave-valor donde el valor es otra estructura (ej. DynamoDB, Cassandra, Bigtable). "Poco flexibles y altamente performante." Se define una clave de partición (PK) y una clave de ordenamiento (SK).
* **Orientadas a Documentos:** Almacenan datos en "Colecciones de documentos", a menudo en formato JSON. "Estructura jerárquica". "Schemaless → Schema on read", lo que las hace "Flexibles → Puedo hacer muchas queries". "Índices sobre cualquier campo". Ejemplos: MongoDB, CouchDB, Elasticsearch.
* **Orientadas a Grafos:** Las entidades son nodos y sus relaciones. Permiten realizar "queries más complejas que con SQL" para explorar conexiones (ej. Neo4J). Problema: "No son particionables, todo debe estar conectado."

## Teorema CAP

Establece que un sistema distribuido puede satisfacer un máximo de dos de las siguientes tres propiedades:

* **Consistencia (C):** Todos los clientes ven los mismos datos al mismo tiempo.
* **Disponibilidad (A):** Todos los clientes pueden leer o escribir datos, incluso en presencia de fallas.
* **Tolerancia a Particiones (P):** El sistema continúa funcionando a pesar de las fallas en la red que dividen el sistema en subredes. En la realidad de un sistema distribuido, la tolerancia a particiones es casi un requisito (siempre hay riesgo de fallas de red), lo que lleva a las bases de datos NoSQL a elegir entre consistencia y disponibilidad.

## Modelos de Consistencia: ACID vs BASE

### ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad)

Típicamente asociado con bases de datos relacionales, garantiza transacciones confiables.

* **Atomicidad:** "La transacción se ejecuta entera o no se ejecuta".
* **Consistencia:** "La BD pasa de un estado consistente a otro".
* **Aislamiento (Isolation):** "la transacciones no se afectan entre sí".
* **Durabilidad:** "una vez que termina una transacción los datos de esta son almacenados".

### BASE (Basically Available, Soft-State, Eventual Consistency)

Asociado con bases de datos NoSQL, prioriza la disponibilidad y la tolerancia a particiones sobre la consistencia fuerte.

* **Basic availability:** "la BD está disponible la mayor parte del tiempo".
* **Soft State:** "el estado de la BD puede cambiar incluso sin entradas".
* **Eventual consistent:** Los datos eventualmente serán consistentes en todo el sistema, pero no necesariamente de inmediato.

## Particionamiento (Sharding)

Técnica para distribuir datos en múltiples servidores. "No asociado al negocio".

* **Hash Distribuido (Módulo N):** Cada elemento tiene una clave que se hashea, y el resultado se usa con el módulo del número de servidores para asignar un servidor. Problema: Si se cae o se agrega un servidor, "Cambia la función de asignación de servidor", lo que requiere rebalancear gran parte de los datos.
* **Consistent Hashing:** Asigna servidores y claves a un anillo de hash. Cuando se agrega o quita un servidor, solo una fracción de las claves necesita ser reubicada, lo que minimiza el impacto. "Se afectan a menos servidores si se cae uno o se agrega otro y balancea la carga." También se usan nodos virtuales para una distribución más homogénea.

## Replicación

Duplicar los datos en **múltiples nodos** para mejorar la **disponibilidad** y la **tolerancia a fallos**.

* Si un nodo cae, otro con la réplica puede responder.
* Puede haber **replicación primaria/secundaria** o múltiple activa.

## Estrategias de Optimización en Bases de Datos

* **Lectura y Escritura Secuencial:** Son más rápidas que las aleatorias.
* **Índices:** Aceleran las consultas, pero las escrituras aleatorias en el índice pueden ser lentas.
* **LSMT (Log Structured Merge Trees):** Para índices grandes, agrupa escrituras, las ordena y guarda en disco, manteniendo índices más pequeños en memoria para búsquedas.
* **Bloom Filter:** "estructura probabilística que nos dice si un elemento se encuentra o no en un conjunto." "Si nos dice cuál elemento se encuentra, puede no ser cierto, pero si dice que no está, es 100% seguro." Ocupan poco espacio y se usan para evitar lecturas innecesarias en disco.

# DevOps

**DevOps** es una **cultura y conjunto de prácticas** que buscan integrar el trabajo de desarrollo (**Dev**) y operaciones (**Ops**) para:

* Entregar software más rápido y confiable
* Automatizar la infraestructura y los procesos de despliegue
* Mejorar la colaboración entre equipos técnicos

## CI/CD (Integración y Entrega Continua)

* **CI (Continuous Integration):** Automatiza validaciones desde que hay un nuevo cambio en el código.
  + Linting, tests, builds automáticos
  + Chequeo de dependencias y vulnerabilidades
  + Feedback rápido y estandarizado
* **CD (Continuous Delivery / Deployment)**
  + *Delivery*: El código está listo para deploy, pero se requiere intervención.
  + *Deployment*: El deploy ocurre automáticamente al aprobar un cambio.

## Infrastructure as Code (IaC)

Infraestructura definida como código, usualmente con DSLs como YAML o HCL (Terraform):

* Versionable
* Automatizable
* Puede ser **declarativo** (decís qué querés) o **imperativo** (cómo lograrlo)
* Riesgo de *vendor lock-in* según la herramienta usad

## Kubernetes

Plataforma de orquestación de contenedores (open-source):

* Modelo **declarativo**: se define el estado deseado
* Soporte para múltiples entornos

**Componentes:**

* **Nodos**: master y workers
* **API Server**: expone la API de Kubernetes
* **ETCD**: almacén clave-valor con el estado del clúster
* **Scheduler**: decide dónde corre cada pod
* **Controllers**: mantienen el estado deseado
* **Objetos**: Pods, Services, Deployments, etc.

## Git Branch Model - GitFlow

Modelo de branching para ordenar el desarrollo:

* **Feature Branch**: para nuevas funcionalidades
* **Hotfix Branch**: para correcciones urgentes
* **Release Branch**: para preparar una nueva versión
* **Main / Estable**: rama productiva

Permite backports (llevar fixes de una rama a otra).

### GitOps

Extensión de DevOps que usa Git como **fuente de verdad** para toda la infraestructura y aplicaciones.

* Toda configuración está versionada en Git
* Los cambios son automatizados
* Se separa el código de app del manifiesto de infraestructura

# Site Reliability Engineering (SRE)

SRE es "una implementación de DEVOps" que se enfoca en lograr un "sistema altamente confiable desde la perspectiva del usuario". El objetivo principal no es 100% de disponibilidad, ya que esto es "la confiabilidad equivocada para básicamente todo" debido a su "costo exponencial".

## Error Budget

Es un principio clave de SRE.

"Error budget (unavailability) = 100% - Availability Target".

Representa el tiempo máximo de indisponibilidad aceptable para un servicio.

* Permite "alinear el riesgo asumido por un servicio con el riesgo que la empresa está dispuesta a soportar."
* "Evidencia y da soporte al conflicto entre los objetivos de desarrollo y operaciones".
* Si se excede el Error Budget, se debe "freeze" (congelar) la implementación de nuevas características y "utilizar el tiempo para ganar resiliencia", trabajando en minimizar el riesgo y mejorar la confiabilidad. Esto pone a prueba "la voluntad para implementar SRE".
* "Outage pasa a ser parte del proceso de innovación."
* Permite a Product Management y SRE tener un "lenguaje común" y decidir cuántos releases habrá y simplificar las discusiones sobre interrupciones con stakeholders.

## Métricas Clave

* **SLI (Service Level Indicator):** "Una medición bien definida de successful enough". Medición cuantitativa de algún aspecto del nivel de servicio (ej. latencia de solicitud, tasa de error).
* **SLO (Service Level Objective):** "target de interacciones exitosas". Es un objetivo acordado para un SLI, definido para no "hacer sobreingeniería y quizás no lo vale". "El SLO no es independiente al negocio, sino que se calcula en base al daño que se puede soportar."
* **SLA (Service Level Agreement):** "Incluye consecuencias si no se alcanza el nivel deseado". Es un contrato explícito o implícito, a menudo con implicaciones económicas o de reputación si no se cumple. "SLA = SLO + margen".

## Principios y Prácticas

* **Voluntad y Sponsor Ejecutivo:** La implementación de SRE requiere un fuerte apoyo de la dirección.
* **Empezar Pequeño:** "Una aplicación a la vez" para entender "qué entiendo por 'aplicación'?" (un dominio de falla discreto).
* **Definir el Error Budget Primero:** "Si hacés esto primero, el resto de las cosas irán funcionando por sí mismas."
* **Automatización:** Es fundamental para eliminar el "toil" (trabajo manual repetitivo, sin valor duradero). "Si a human operator needs to touch your system during normal operations, you have a bug."
* **Monitoreo y Alertas:** Es como "volar sin instrumentos" sin monitoreo. Las alertas deben ser significativas y afectar al usuario, evitando la "fatiga de alerta".
* **Cultura de No Culpabilización:** En caso de fallas, el enfoque es entender "qué pasó, cómo evitar que vuelva a pasar" (Root Cause Analysis - RCA, Post Mortem).
* **Aceptar Riesgos:** "La confiabilidad tiene costo." Se debe aceptar que "va a haber bugs" y el objetivo es utilizar el error budget para maximizar la velocidad de entrega de características.

## Pasos para implementar SER

1. Una aplicación a la vez.
2. Definir el Error Budget.
   1. Se desprende del SLO. Indica si podemos acelerar o frenar la salida de features del sistema.
3. Monitoreo y carga de operaciones
   1. Solo alertar si afecta al usuario.
   2. Entender caminos y apps críticas.
   3. Carga operativa solo a cargo del equipo on call (“dejar al resto del equipo enfocado”). Importante saber cuanto tiempo se destina a ops.
4. Blameless Culture
   1. “Culpar sistemas, no humanos”.
5. No hacer todo a la vez.

# Inteligencia Artificial

## IA Generativa

Subcampo de la IA que crea contenido nuevo (texto, imágenes, audio) a partir de patrones aprendidos.

Usa **modelos generativos** como LLMs (Large Language Models).

### Modelos de Lenguaje Grandes (LLMs)

Los Modelos de Lenguaje Grandes (LLMs) se presentan como sistemas de IA con capacidades avanzadas de aprendizaje continuo y generalización.

**Características Principales:**

* **Sistema Único y de Aprendizaje Continuo:** El conocimiento de un dominio puede aplicarse a otro.
* **Generalización Completa:** Capacidad de aprender y aplicar conocimiento a cualquier dominio.
* **Transferencia de Aprendizaje:** Lo aprendido en un área se aplica a otras.
* **Razonamiento Abstracto:** Habilidad de pensamiento conceptual y creativo.
* **Autonomía:** Pueden establecer sus propios objetivos y estrategias.

**Funcionamiento Interno de los LLMs:**

1. **Tokenización:** Las frases se dividen en "piezas pequeñas" llamadas tokens.
2. **Representación Vectorial:** Cada token se representa como un vector con valores. "Si representamos estos vectores en dos dimensiones los que tengan significados similares van a estar más cerca. Si ejemplificamos Vaca va a estar más cerca de Gata que Toro."
3. **Bloque de Atención:** Los vectores pasan por "un montón de operaciones" que se llama bloque de atención, permitiendo que los tokens interactúen entre sí.
4. **Perceptrón Multicapa (MLP):** Después, pasan por un "proceso de operación diferente, Perceptron Multilayer, acá los vectores no se hablan entre si todos pasan por la misma operación en paralelo." Este paso es "como hacer una larga lista de preguntas sobre cada vector y luego actualizarlo en base a las respuestas."
5. **Cálculos Matriciales:** "Todo esto se ve como un montón de multiplicaciones de matrices en paralelo." Estas operaciones se realizan contra "parámetros", que forman "matrices de pesos". GPT-3, por ejemplo, tiene "~175.000.000.000" parámetros.
6. **Iteración y Predicción:** El proceso de atención y multicapas se repite "muchísimas veces" hasta obtener un vector final. Sobre este vector se realiza una "operación de distribución de probabilidad sobre todos los tokens posibles, es decir todos los posibles segmentos que podrían venir a continuación." Este proceso se repite, prediciendo la siguiente palabra dada una palabra semilla y el contexto.

Los LLMs se entrenan con grandes cantidades de texto y ajustan sus pesos mediante aprendizaje profundo (deep learning), principalmente usando arquitecturas de tipo Transformer.

#### RAG (Retrieval Augmented Generation)

Técnica para mejorar respuestas de un LLM con **información externa** (docs, PDFs, etc.).

**Proceso:**

* Se consulta una **Vector Store**
  + Archivo con conocimiento (PDFs, etc.) que se puede asociar a un **Assistant o Thread**.
* Se inyecta el contenido relevante al prompt
* El LLM responde con más precisión y citando fuentes

#### MCP (Model Context Protocol)

Protocolo estándar tipo “USB-C” para conectar modelos a herramientas/APIs de forma declarativa. Permite a los LLMs **interactuar directamente con funcionalidades externas**, sin que la app actúe como intermediaria.

## Prompt Engineering

Disciplina que busca **diseñar instrucciones efectivas** para obtener mejores respuestas de un LLM.

**Buenas prácticas**:

* Objetivo claro y conciso.
* Contexto bien definido.
* Ser detallista y evitar ambigüedades.
* Incluir ejemplos, estructura, pasos.
* Refinar prompts mediante prueba/error.

## Agent-to-Agent (A2A)

Protocolos y librerías para que **dos agentes IA colaboren entre sí**, compartan contexto y resuelvan tareas complejas juntos.