Resumen Desarrollo de Software – L.A.A.

Introducción al desarrollo de SW:

Software: Conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora.

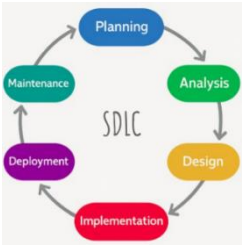
Programas: Instrucciones que ejecuta la computadora.

Datos: Información que los programas manipulan.

Documentación: Por ejemplo, manuales, especificaciones, etc.

Desarrollo de Software: Es el proceso de idear, diseñar, programar, documentar, probar y mantener aplicaciones. Va mucho más allá de simplemente escribir código.

Ciclos de vida del Desarrollo de SW:



- ***Planificación:*** Lo que se quiere hacer, para que y con qué recursos.
- ***Análisis:*** Necesidades del sistema, tanto SW como armado de HW.
- ***Diseño:*** Cómo funcionará y se verá el sistema.
- ***Implementación:*** Prueba y despliegue para preparar el sistema para su uso.
- ***Mantenimiento:*** Se corrigen errores y se hacen mejoras.
- ***Mejoras/Despliegue:*** Utilización del sistema para aplicar mejoras continuas.

Modelos SDLC: Existen muchos, pero los otros se basan en ellos:

- ***Cascada:*** se va haciendo el primer paso y hasta que no termina no se avanza, es un proceso lineal y secuencial.
- ***Iterativo:*** "mini-cascadas" se divide el proceso entero en partes, entonces en vez de hacer lo del modelo de cascada, se divide en etapas aún más pequeñas.
- ***Desarrollo Ágil:*** Es un desarrollo iterativo e incremental, y está orientado al despliegue rápido. Tiene una participación continua con el cliente y está basado en hacer entregas en menos tiempo que el modelo anteriormente mencionado.

Introducción a la arquitectura de SW:

Arquitectura de SW: Descripción abstracta de un sistema que define sus componentes, responsabilidades, interacciones y principios de diseño. Como disciplina, establece una visión técnica y organizativa que guía el diseño, desarrollo, evolución y mantenimiento de sistemas complejos.

Aspectos clave:

- ***Modularidad:*** Capacidad de un sistema para dividirse en módulos independientes entre sí, pero que trabajan en conjunto. Cada módulo realiza una función específica, lo que facilita la reutilización, el mantenimiento y la comprensión del sistema
- ***Escalabilidad:*** Capacidad de un sistema para manejar un aumento de carga sin bajar su rendimiento. Puede ser vertical (mejorar el HW) o horizontal (agregar más nodos o instancias).
- ***Mantenibilidad:*** Facilidad con la que se puede modificar un sistema para corregir errores, mejorar funcionalidades o adaptarlo a nuevos requisitos.
- ***Rendimiento:*** Eficiencia con la que un sistema ejecuta sus tareas.
- ***Seguridad:*** Capacidad de un sistema para protegerse contra accesos no autorizados, manipulación de datos, ataques cibernéticos y otros riesgos.
- ***Disponibilidad:*** Grado en que un sistema está operativo y accesible cuando se necesita.
- ***Portabilidad:*** Facilidad con la que un sistema o SW puede ejecutarse en diferentes entornos con pocas o ninguna modificación.

Componentes principales:

- ***Componentes / módulos:*** Unidades funcionales del sistema. Pueden ser clases, servicios, bibliotecas o subsistemas completos.
- ***Conexiones:*** Mecanismos que permiten la comunicación entre componentes.
- ***Estructura:*** Es la forma en que están organizados y conectados los componentes que forman el sistema.
- ***Estilos y patrones arquitectónicos:*** Conjunto de prácticas y soluciones reutilizables que guían el diseño arquitectónico, como cliente-servidor.

Evolución histórica de las arquitecturas de SW:

- ***1960s–1970s – Aplicaciones monolíticas:*** Todo el código en un solo programa. Difíciles de mantener. Ej.: Sistemas de contabilidad, mainframes.
- ***1980s–1990s – Cliente-servidor:*** División en cliente y servidor. Centraliza datos y lógica. Ej.: Aplicaciones con bases de datos empresariales.
- ***1990s–2000s – Tres Capas:*** División en Presentación, Lógica y Datos. Mejor mantenimiento y escalabilidad. Ej.: Aplicaciones web, sistemas ERP.
- ***2000s–2010s – Orientada a Servicios (SOA):*** Servicios independientes con interfaces definidas. Reutilización y flexibilidad. Ej.: Sistemas bancarios, plataformas de e-commerce.
- ***2010s–Actualidad – Microservicios:*** Servicios pequeños y específicos. Desarrollo y escalado independiente. Ej.: Netflix, Amazon, Spotify.
- ***Actualidad – Arquitecturas serverless:*** Código sin gestionar servidores. Escalado automático. Ej.: AWS Lambda, Azure Functions.

Estilos Arquitectónicos:

Monolítica: En esta todas las funcionalidades de la aplicación se tratan como una sola unidad. Todo el código esta interconectado y ejecutado en un único proceso. Es de fácil desarrollo y su rendimiento es muy eficiente.

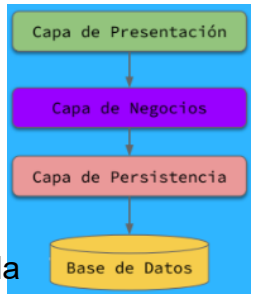
Cliente-Servidor: Se divide el sistema en dos componentes principales: el cliente (envía solicitudes al servidor) y el servidor (procesa las solicitudes y devuelve las respuestas). Esta facilita la escalabilidad horizontal y la separación de sus responsabilidades está bien marcada.

Orientada a servicios (SOA): Organiza una aplicación como un conjunto de servicios autónomos, que se comunican a través de una red. Cada servicio realiza una función específica y puede ser reutilizado en diferentes aplicaciones.

Microservicios: La aplicación se divide en pequeños servicios independientes que se encargan de tareas específicas, cada uno con su propia lógica y base de datos. Algunas de sus ventajas son la escalabilidad y flexibilidad y el aislamiento de fallos (un fallo en un microservicio no afecta a los demás).

Arquitectura multicapa: Patrón que organiza el sistema en niveles jerárquicos, donde cada capa tiene una función bien definida. Busca mejorar la organización del código y la interacción entre los distintos componentes.

- **Modularidad:** Cada capa tiene una función específica.
- **Escalabilidad:** Permite escalar componentes en forma independiente.
- **Mantenibilidad:** Facilita localizar y corregir el problema.
- **Reutilización:** Posibilita reutilizar componentes en diferentes aplicaciones.
- **Testing:** Facilita la implementación de pruebas unitarias
- **División del trabajo:** Permite que distintos equipos trabajen en diferentes capas.



Estructura multicapa: Todas las capas se colocan de forma horizontal, de tal forma que cada capa solo puede comunicarse con la capa que está inmediatamente por debajo.

Capa de Presentación (UI): Es la encargada de la interfaz de usuario. Esta interactúa con el usuario final, muestra información y captura datos de entrada e interactúa con la capa de negocios a través de APIs o servicios.

Capa de Negocios: Es la responsable de implementar las reglas y la lógica que determinan el funcionamiento de la aplicación. Se encarga de la implementación de reglas de negocio, el procesamiento de datos, coordinación de acciones y validación de datos.

Capa de Persistencia: Se encarga acceder y manejar los datos guardados, ya sea una base de datos o cualquier otro tipo de almacenamiento. Se encarga de la abstracción del acceso a datos, operaciones CRUD, mapeo objeto-relacional (ORM) y de la gestión de transacciones.

Control de versiones y trabajo colaborativo:

Control de versiones: Es como una máquina del tiempo para código. Es un sistema que registra cambios en archivos a lo largo del tiempo y que permite recuperar versiones específicas. Facilita la colaboración múltiple y registra quien realizó cada cambio y porqué. Antes se hacían copias manuales con diferentes nombres, se compartían por ejemplo por email, no se sabe quién cambió que, ni tampoco cuándo.

Tipos de sistemas de control de versiones:

- ***Sistemas locales:*** Copias en diferentes directorios, bases de datos simples en la maquina local, no permite colaboración.
- ***Sistemas centralizados (CVCS):*** Servidor central con todos los archivos versionados. El problema es que, si un componente falla, todo el sistema deja de funcionar. Ej. CVS, SVN
- ***Sistemas distribuidos (DVCS):*** Copia completa del repositorio. Mayor robustez y flexibilidad. Ej. Git.

GIT: Lo creó Linus Torvalds en 2005 para el desarrollo del kernel de Linux. Está diseñado para ser Rápido, simple en diseño, descentralizado, capaz de manejar grandes proyectos y cuenta con soporte para desarrollo no lineal.

Repositorio: Almacén de archivos y su historial de cambios.

Working directory: Archivos con los que trabajamos actualmente.

Staging area: Área intermedia donde preparamos los cambios.

Commit: Fotografía del estado de los archivos en un momento dado.

Branch: Línea independiente del desarrollo.

Remote: Versión del repositorio alojada en un servidor (Ej. GitHub).

Flujo de trabajo básico:

- ***git status:*** Muestra el estado del directorio de trabajo y del área de staging.
- ***git diff:*** Permite ver cambios en archivos entre commits/staging area y el espacio de trabajo.
- ***git add:*** Añade uno o más archivos al área de staging.
- ***git commit:*** Guarda los cambios del área de staging en el historial del repositorio local.
- ***git push:*** Sube tus commits locales a la rama especificada del repositorio remoto.
- ***git init:*** Inicializa un nuevo repositorio en un directorio existente.
- ***git clone:*** Copia un repositorio remoto a tu maquina local.
- ***git log:*** Muestra el historial de los últimos commits del Branch.
- ***git branch:*** Lista las ramas loales.
- ***git checkout:*** Cambia a la rama especificada. También sirve para revertir los cambios de archivos en el directorio de trabajo

- ***git merge:*** Integra los cambios de la rama específicada en la rama actual. Existen 2 tipos principales de merge:
    - Fast-forward merge: Cuando no hay cambios divergentes. Simplemente adelanta la rama destino hasta nuestra rama.
    - Three-way merge: Cuando ambas ramas tienen cambios diferentes. Git crea un nuevo commit que combina los cambios de ambas ramas.
- ***git fetch:*** Descarga las ramas y los commits del repositorio remoto especificado (sin mergearlas)
- ***git pull:*** Descarga los cambios del repositorio remoto y los integra (merge) en tu rama local actual. Es la combinación de los dos anteriores.

Ramas en git: Permiten trabajar por fuera de la línea principal de desarrollo. Sirve para desarrollar nuevas funcionalidades, corregir errores y experimentar sin afectar el código principal.

Mejores prácticas para trabajar en equipo en Git y GitHub:

- ***Commits frecuentes:*** Hacer pequeños cambios y varios constantemente para seguir el progreso y colaborar mejor.
- ***Convenciones de commits:*** Usar mensajes claros que sigan una estructura para facilitar el seguimiento.
- ***Gestión de ramas:*** Nombrar bien las ramas y reorganizarlas para mantener el historial limpio.
- ***Pull requests:*** Usarlos para documentar cambios y facilitar el recorrido del código.
- ***Automatización y revisión:*** Automatizar pruebas y revisar el código para asegurar calidad.

Conflictos en fusión de ramas: Pueden ocurrir si se modifican las mismas líneas en diferentes ramas o cuando Git no puede determinar qué cambio conservar. Para resolverlo manualmente debemos editar los archivos para arreglar los conflictos, luego marcarlo como resueltos (git add) y finalmente completar el merge (git commit).

GitHub: Plataforma web que proporciona hosting para repositorios Git y añade funcionalidades para facilitar la colaboración.

Issues: Permiten hacer seguimiento de tareas, bugs o mejoras.

Pull Requests: Forma estructurada de proponer cambios y revisarlos en equipo antes de integrarlos.

Actions: Permite automatizar flujos de trabajo como pruebas o deploys.

Projects: Facilita la gestión visual de tareas con tableros tipo Kanban.

Pages: Permite publicar sitios web directamente desde un repositorio.

Requerimientos y modelado

Requerimiento Funcionales: ¿**Qué** debe hacer el sistema?

Requerimientos No Funcionales: ¿**Cómo** debe ser/hacer el sistema?

Restricciones: Son limitaciones: Técnicas, legales, de negocio, de recursos.

Procedimiento para armado de backlog:

- → Fase 1: Preparación y análisis inicial
- → Fase 2: Modelado de dominio
- → Fase 3: Elaboración del producto backlog
- → Fase 4: Validación y trazabilidad
- → Fase 5: Seguimiento y mejora continua.

Metodologías Ágiles:

Modelo en cascada: Es secuencial y rígido, hay documentación exhaustiva, es resistente al cambio y tiene feedback tardío del cliente. Problemas: requisitos cambiantes, alto riesgo al final del proyecto, baja visibilidad del progreso real, expectativa vs realidad. (hay otra definición antes).

Scrum: NO es una metodología completa sino un marco de trabajo que permite aplicar diversas técnicas y procesos. Está orientado a la gestión de proyectos complejos, y se basa en un enfoque iterativo e incremental, lo que permite controlar mejor los riesgos. Los 3 pilares son: Transparencia, inspección y adaptación. Scrum tiene un enfoque en roles, eventos y artefactos:

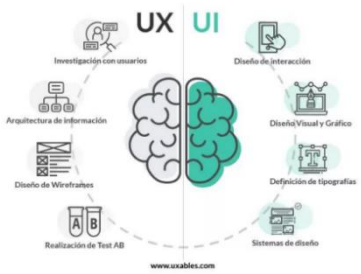
- → *Roles:* Product owner (representa al cliente), Scrum Master (facilita el proceso eliminando impedimentos) y equipo de desarrollo (responsable de entregar los incrementos).
- → *Artefactos:* Product Backlog (PB) (Lista ordenada de todo el trabajo), Sprint Backlog (Conjunto de elementos del PB) e Incremento (Suma de todos los elementos completados).
- → *Eventos:* Sprint (Periodo de tiempo fijo donde se trabaja para entregar un incremento), Sprint planning (Reunión clave al inicio de cada sprint), Daily scrum (Reunión diaria de 15 minutos), sprint review (Revisión del incremento al final del sprint), sprint retrospective (Inspección de equipo)

Kanban: Herramienta ágil que usa tableros visuales para organizar tareas, controlar el flujo de trabajo y evitar sobrecarga.

XP (Extreme Programming): Método ágil que mejora la calidad del software con prácticas como TDD (diseño guiado por los test), programación en pares e integración continua.

Beneficios del enfoque ágil:

→ **Mayor satisfacción del cliente**
→ **Adaptación rápida a cambios**
→ **Entrega constante de valor**
→ **Mejor calidad del producto**
→ **Mayor visibilidad y transparencia**
→ **Reducción de riesgo**
→ **Mejor ambiente de trabajo**



Introducción a UX/UI:

UX (User Experience): Hace foco en el usuario y en la experiencia que se quiere lograr. Es lo que experimenta el usuario antes, durante y después de interactuar con el dispositivo. Sin usuario no hay UX.

UI (User Interface): Hace foco en el dispositivo, más específicamente en lo que se ve en pantalla.

Casos de éxito: Airbnb (Su rediseño aumentó conversiones en 30%), Duolingo (gamificación: llevar cosas de los juegos), Apple (Simplicidad y consistencia)

Casos de fracaso: Snapchat (Su rediseño implicó pérdida de usuarios y caída en la bolsa), Windows 8 (Interfaz metro tuvo conflictos con usuarios), Google+ (complejidad y falta de propuesta clara)

Roles Principales en un equipo UI/UX:

- **UX Researcher:** Investiga necesidades de usuarios
- **UX Designer:** Define la estructura y flujos
- **UI Designer:** Diseña interfaces visuales

Entregables Comunes:

- **Wireframe:** Representación visual básica, de la estructura y diseño de una página web o aplicación, enfocada en la organización de la información y la disposición de los elementos en la página.
- **Mockup:** Representación visual más avanzada y detallada del diseño de una página web o aplicación, enfocada en los detalles de estilo, como los colores, tipografía, etc. Se usa para presentar el diseño a los clientes y otros miembros del equipo y así obtener retroalimentación.
- **Prototipo:** Versión interactiva y funcional de una página web o aplicación, centrada en la experiencia de usuario, permitiendo probar la funcionalidad y navegación en tiempo real. Puede incluir animaciones, interacciones y otros elementos que simulan una experiencia real.

Principios Fundamentales de diseño UX/UI:

- ◊ **Centrado en el usuario:** Priorizar al usuario en el proceso de diseño UX.
- ◊ **Jerarquía:** Ayudar a dirigir la atención del usuario a los elementos más importantes del sitio web, optimizando la navegación.
- ◊ **Usabilidad:** Facilidad con la que un usuario interactúa con el producto.
- ◊ **Consistencia:** *En UI* se busca consistencia visual. Es clave mantener coherencia en fuentes, colores, diseños y elementos en todas las páginas. *En UX* Se prioriza la consistencia funcional. Los usuarios esperan que los elementos funcionen como están acostumbrados. Se puede ser creativo, pero intentando mantener la lógica principal.
- ◊ **Accesibilidad:** Diseñar el producto para que pueda usarlo la mayor cantidad de personas posible
- ◊ **Contexto:** Personalización del producto según las situaciones, entornos y necesidades específicas en las que los usuarios interactúan con él.

Design Thinking (Enfoque para UX): Metodología para resolver problemas de manera creativa y centrada en las personas. *Pasos:*

- **Empatizar:** Comprender necesidades.
- **Definir:** Identificar problemas y oportunidades.
- **Idear:** Soluciones.
- **Prototipar:** Crear representaciones.
- **Testear:** Evaluar soluciones.

Planificación del primer sprint:

Taiga: Plataforma open source para gestión ágil de proyectos.

Fundamentos Frontend (HTML5 + CSS3 + JS ES6):

Diferencia entre página web de una aplicación:

- → ***Página web:*** Información estática, solo lectura, navegación simple.
- → ***Aplicación web:*** Interactiva, maneja estados, funcionalidades complejas.

HTML5 estructura semántica: Accesibilidad (Lectores de pantalla), SEO (mejor indexación), mantenimiento (Código más claro), etiquetas semánticas. Estructura ≠ de presentación.

CSS3: Box model (margin, border, padding, content), Flexbox (layout flexible y responsive), CSS Grid (layouts complejos en 2D), variables CSS (Reutilización y mantenimiento), Media Queries (responsive design).

Desarrollo FrontEnd Moderno: Introducción a React:

Problemas con Vanilla JavaScript: Manipulación compleja del DOM, Código repetitivo y difícil de mantener, Difícil organización en proyectos Grandes, Reutilización limitada de código.

Ventajas de React:

- Componentes reutilizables
- Virtual DOM = mejor rendimiento
- Curva de aprendizaje suave
- Comunidad masiva
- Alta demanda laboral
    - En Argentina aproximadamente el 70% de las ofertas frontend requieren React.

JSX: JavaScript que parece HTML, pero NO es HTML.

Componentes: Piezas reutilizables de interfaz de usuario. Son funciones que retornan JSX. Nombres siempre en PascalCase.

Props: Argumentos que se le pasan a un componente.

Estado: Información que puede cambiar y hace que el componente se actualice.

Desarrollo FrontEnd React: Del caos al orden:

Librerías UI: Colecciones de componentes y herramientas que se usan para no escribir CSS desde cero, tiene consistencia visual automática y componentes ya testeados. Ej: Bootstrap, Tailwind.

React: Hooks y Conexión con APIs:

Hooks (ganchos): Funciones especiales que nos permiten "engancharnos" a las funcionalidades internas de React.
Reglas: Solo se pueden usar en componentes funcionales. Siempre se llaman en el mismo orden. Siempre en el nivel superior (no dentro de loops, condiciones, etc).

API (Application Programming Interface: Interfaz de Programación de Aplicaciones): Forma estandarizada que permite que dos sistemas se comuniquen entre sí.

Fetch(): Forma de hacer peticiones HTTP en JS.

Custom Hooks: Funciones que nos permiten extraer lógica compleja y reutilizarla.

Review, Retrospectiva y Planning:

Demo: Demostración del incremento de producto terminado. Presentación a stakeholders. Recolección de feedback. **Objetivo:** Exponer el valor entregado.

Reunión Retrospectiva: Inspección del proceso de trabajo. Identificación de mejoras. Planificación de cambios. Momento de reflexión del equipo.

Fundamentos Backend:

Métodos HTTP:

- ***GET:*** Solicitar recursos.
- ***POST:*** Crear recursos.
- ***PUT:*** Actualizar/reemplazar completo.
- ***DELETE:*** Eliminar recursos.
- ***PATCH:*** Actualización parcial.

Códigos de estado:

- → ***2xx Éxito:*** EJ: 200 OK, 201 Created.
- → ***4xx Error del cliente:*** EJ: 400 Bad Request, 404 Not Found, 401 Unauthorized.
- → ***5xx Error del servidor:*** EJ:500 Internal Server Error.

NODE.JS: JS en todas partes. Mismo lenguaje para FRONTEND y BACKEND.

Ventajas de TypeScript (TS): Prevención de errores de tipos, interfaces y tipos personalizados, Generics para código reutilizable.

Backend: Primeras APIs:

Ventajas de Express.js: Ligero y minimalista, extensible, gran ecosistema y comunidad activa, compatibilidad con JS y TS.

Persistencia de Datos:

ORM (Object Relational Mapping): Herramienta que actúa de puente entre la POO y las BDD relacionales para permitirles interactuar.
Responsabilidades:

- **Mapeo**
- **Hidratado:** Convertir registros de la BDD en objetos.
- **Persistencia:** guardar objetos en la BDD
- **Conversión y validación de datos**
- **Abstracción:** Independencia del motor de bdd.

Relaciones y Validaciones:

Niveles de validación:

Frontend (UX): Validaciones básicas, no es suficiente

Backend (Security): Validacion confiable. NUNCA CONFIAR EN EL FRONT.

Base de Datos: Última línea de defensa.

Entrega Backend:

CORS (Cross Origin Resource Sharing): Mecanismo de seguridad que utilizan los navegadores, permitiendo o negando solicitudes.

Proyecto/Repositorio: Permite versionar archivos, puede haber un repositorio que contenga un proyecto o varios.

Servicio: Proceso de ejecución que ofrece funcionalidades.

Servidor/Host: Maquina o entorno donde se ejecutan los servicios. Puede almacenar uno o varios servicios.

Seguridad en Aplicaciones Web:

Principios de seguridad:

- Confidencialidad: Proteger la información, solo opera quien corresponde.
- Integridad: Asegurar que no se altere la información.
- Disponibilidad: Sistema accesible cuando se necesita.

Autenticación: Obtener identidad (Login con contraseña).

Autorización: Acceder a recursos (verificar permisos según rol).

JWT (JSON Web Token): Estándar abierto para transmitir información de forma segura entre partes.

Docker y Contenerización:

Docker: Plataforma de Contenerización que permite empaquetar una aplicación junto con todas sus dependencias dentro de una imagen.
Conceptos Clave:

→ **Dockerfile**: Archivo de texto que define pasos de construcción de una imagen.
→ **Imagen:** Resultado de la construcción. Contiene el sistema de archivos y dependencias.
→ **Contenedor:** Instancia de ejecución de una imagen.
→ **Docker Hub:** Repositorio público de imágenes oficiales.

Validaciones y autorización en Frontend:

La validación en frontend mejor la experiencia de usuario (UX). No es seguridad.

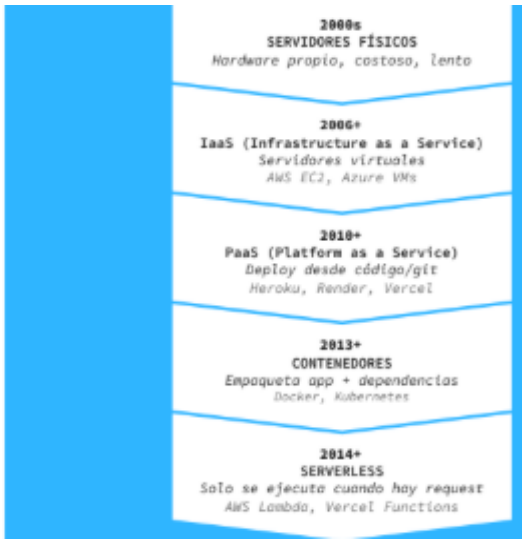
Deployment y entornos:

Deployment / Deploy: Proceso de publicar versiones de nuestros desarrollos en producción, es preparar un ambiente confiable.

Development: Donde programa cada DEV, experimentando. Ej. Nuestro equipo.

Staging: Es opcional. Copia “fiel” de producción. Se utiliza para testing antes de liberar a producción.

Production: Lo que usan los usuarios finales. Debe ser estable. Monitoreo todo el tiempo.



Modelo	Control	Complejidad	Costo
IaaS	Alto	Alta	$$$
PaaS ★	Medio	Media	$$
Serverless	Bajo	Baja	$

Introducción al testing:

Testing: Probar que el SW funciona correctamente. Sirve para una funcionalidad correcta, prevenir errores, casos límite, etc.

Tipos de testing:

- → **UI:** Prueban el flujo completo. Pocos, lentos, caros, Test End-to-End (E2E). Simulan usuarios reales.
- → **Integración:** Prueban varias piezas funcionando juntas. Algunos, medianos, APIs, BD.
- → **Unitarios:** Prueban una función o componente aislado. Muchos, rápidos, baratos.

Estructura AAA, se epite en todos los Tests en todos los lenguajes:

- ➢ **ARRANGE (Preparar):** Configurar el contexto y los datos de prueba.
- ➢ **ACT (Actuar):** Ejecutar la acción que queremos probar.
- ➢ **ASSERT (Verificar):** Verificar que el resultado es el esperado.

Jest: Framework de testing más usado en JS / TS. Funciona con Node y React.