Preguntas teóricas

1. ¿Cuál es la diferencia entre nube pública, privada e hibrida?

Cuando hablamos de nube pública, nos referimos a un modelo de computación en el que un proveedor externo (como AWS, Azure o Google Cloud) ofrece recursos como servidores, almacenamiento o plataformas a través de internet para múltiples usuarios o inquilinos simultáneamente. Este entorno de multi-tenencia permite pagar solo por lo que se utiliza, sin necesidad de inversión previa en infraestructura, y facilita una escalabilidad casi ilimitada con un mantenimiento gestionado por el proveedor. No obstante, compartir infraestructura con terceros implica un menor nivel de control y personalización, potenciales preocupaciones de seguridad y dependencia de una conexión estable a internet.

Por otro lado, la nube privada es un entorno exclusivo para una sola organización, ya sea hospedado internamente o por un proveedor externo. Aquí, todos los recursos son dedicados y accesibles únicamente por la entidad que los posee, lo que permite una profunda personalización, control completo sobre políticas de seguridad y cumplimientos normativos, como en sectores financieros o gubernamentales. No obstante, este modelo demanda una inversión significativa en hardware, personal técnico especializado y mantenimiento continuo, y presenta desafíos para escalar rápidamente, ya que requiere nuevos recursos físicos.

Finalmente, la nube híbrida combina lo mejor de ambos mundos: permite operar cargas sensibles en una nube privada mientras aprovecha la escalabilidad y eficiencia de la nube pública para demandas variables o menos críticas. Este enfoque aporta flexibilidad para migrar dinámicamente aplicaciones o datos según necesidades, potenciar ciclos de desarrollo ágil, o implementar estrategias como el "*cloud bursting*" para manejar picos de carga. No obstante, su arquitectura es más compleja y costosa de integrar y administrar, lo que implica retos en gobernanza de datos, seguridad, latencia de red e incluso riesgos de dependencia prolongada del proveedor (*vendor lock-in*).

2. Describa tres prácticas de seguridad en la nube.

Una de las prácticas fundamentales para asegurar los entornos en la nube es la gestión de identidad y acceso (IAM) con el principio de menos privilegio. Esto implica otorgar a cada usuario únicamente los permisos estrictamente necesarios para realizar su trabajo, así como implementar autenticación multifactor (MFA) para reforzar la protección de las credenciales, especialmente las de usuarios con roles elevados.

Además, es clave revisar y ajustar estos permisos periódicamente para evitar acumulación de derechos innecesarios y posibles vectores de ataque interno o externo.

Otro pilar de la seguridad en la nube es garantizar que todos los datos estén cifrados, tanto en reposo como en tránsito. Esto protege la confidencialidad e integridad de la información frente a interceptaciones o accesos no autorizados. Aunque la mayoría de los proveedores ofrece servicios de cifrado integrados, muchas organizaciones optan también por usar sus propias claves o soluciones (como BYOK o "*bring your own key*") para mantener un control más rígido sobre la protección de datos sensibles.

Finalmente, resulta igualmente crucial establecer un sistema de monitoreo continuo, auditoría y gestión de postura de seguridad (CSPM) en la nube. Revisar logs, detectar configuraciones erróneas y mantener actualizados los sistemas ayuda a identificar amenazas o brechas antes de que sean explotadas. También facilita responder rápidamente a incidentes y cumplir con normativas regulatorias.

3. **¿Qué es la IaC, y cuáles son sus principales beneficios? Mencione 2 herramientas de IaC y sus principales características.**

La **Infraestructura como Código** (IaC) es el enfoque que permite definir y gestionar recursos de TI, como servidores, redes o almacenamiento, mediante archivos de configuración legibles por máquinas, en lugar de hacerlo manualmente o mediante interfaces gráficas. Esto posibilita tratar la infraestructura como código, aplicando prácticas de desarrollo de software como control de versiones, revisiones y despliegues automatizados.

Entre sus beneficios clave, destaca la automatización y eficiencia, ya que permite aprovisionar entornos completos en minutos, evitando tareas manuales repetitivas y propensas a errores. La consistencia y confiabilidad son otra gran ventaja, ya que los despliegues reproducibles evitan la deriva de configuración (*configuration drift*) y garantizan que los entornos (desarrollo, pruebas, producción) sean equivalentes. Además, IaC facilita colaboración y control mediante integración con sistemas de gestión de versiones como Git, lo que permite auditar quién cambió qué, repetir o revertir cambios fácilmente. También mejora la capacidad de recuperación ante fallos, ya que toda la infraestructura puede reconstruirse rápidamente desde cero en otro entorno en caso de desastre. Finalmente, contribuye a optimizar costos al permitir aprovisionar y desmantelar recursos automáticamente según demanda, evitando sobrecostos en infraestructura infrautilizada.

Dos herramientas ampliamente utilizadas en el mundo IaC son Terraform y Ansible, cada una con características destacadas:

Terraform, desarrollado por HashiCorp, emplea un enfoque declarativo mediante su lenguaje HCL (*HashiCorp Configuration Language*), permitiendo expresar el estado deseado de infraestructura sin detallar cómo lograrlo. Es especialmente fuerte en entornos multi-cloud, al funcionar con múltiples proveedores como AWS, Azure, GCP, e incluso entornos on-premises, gracias a su ecosistema extensible de '*providers'*. Administra el estado de los recursos mediante un archivo de estado, lo que le permite calcular diferencias (plan) y aplicar cambios de forma precisa y segura, incluso en equipos colaborativos, asegurando idempotencia. Además, Terraform favorece la reutilización mediante módulos y suele integrarse muy bien en pipelines CI/CD.

Ansible, por su parte, es eminentemente usado para gestión de configuración, aunque también puede usarse como IaC en ciertos contextos. Utiliza archivos YAML para definir una serie de tareas que se ejecutan (imperativo), aunque permite prácticas idempotentes. No requiere agentes en los hosts gestionados (es *agentless*), lo que facilita su despliegue y reduce complejidad operativa. Su sintaxis sencilla lo hace accesible para equipos con menos experiencia en codificación, y cuenta con una amplia variedad de módulos para diversos sistemas y servicios.

4. **¿Qué métricas considera esenciales para el monitoreo de soluciones en la nube?**

Para asegurar la operatividad continua de cualquier solución en la nube, es fundamental comenzar por medir la disponibilidad (*uptime*) del sistema. Una alta disponibilidad protege contra interrupciones que pueden resultar costosas en reputación e ingresos. Pero contar con un servicio activo no basta: también es clave registrar la latencia, es decir, el tiempo que tarda en responder una solicitud. Una alta latencia degrada la experiencia del usuario, especialmente en aplicaciones sensibles al tiempo.

En adición, es esencial monitorear la tasa de errores (como respuestas HTTP 500 o 400, fallos en consultas o límites de tiempo). Esto permite detectar fallas funcionales incluso cuando el sistema aparenta estar disponible. Al mismo tiempo, el seguimiento de los usos de recursos, como CPU, memoria, I/O de disco, y uso de almacenamiento, facilita identificar cuellos de botella y ajustar el dimensionamiento o escalar adecuadamente para evitar saturaciones o infrautilización.

Además de las métricas técnicas, tener en cuenta el costo es fundamental. Observar el gasto real respecto a lo presupuestado, identificar recursos infrautilizados o evaluar sobrecostos ayuda a mantener la eficiencia financiera de la infraestructura cloud.

5. ¿Qué es Docker y cuáles son sus componentes principales?

Docker es una plataforma de código abierto que facilita el despliegue de aplicaciones mediante contenedores livianos que se ejecutan de forma aislada sobre el sistema operativo, sin la sobrecarga de máquinas virtuales completas. Esta tecnología aprovecha funcionalidades del kernel de Linux como namespaces y cgroups para proporcionar aislamiento y control de recursos (CPU, memoria, I/O, red), permitiendo ejecutar múltiples contenedores simultáneamente sobre una misma instancia de Linux sin un sistema operativo completo por contenedor.

El componente central de Docker es el **Docker Engine**, una aplicación cliente-servidor que incluye un demonio persistente (*dockerd*) responsable de gestionar imágenes, contenedores, redes y volúmenes; una interfaz de línea de comandos (CLI) (*docker*) que permite al usuario interactuar con dicho demonio; y una API REST que habilita esta comunicación tanto a usuarios como a otras herramientas. A partir de esa base, Docker define varios objetos clave que constituyen su ecosistema: las imágenes, que son plantillas de sólo lectura que contienen el código y dependencias necesarias para generar contenedores; los contenedores, que son instancias en ejecución de estas imágenes, aisladas y portables; y los registros Docker (*registries*), como Docker Hub, que permiten almacenar, compartir y recuperar imágenes fácilmente.

Referencias

10 cloud security best practices Every organization should follow | DigitalOcean. (2024, 4 abril). https://www.digitalocean.com/resources/articles/cloud-security-best-practices

Ansible vs. Terraform, clarified. (s. f.). https://www.redhat.com/en/topics/automation/ansible-vs-terraform

Bhatnagar, P., & Kudrati, A. (2025, 18 junio). *11 best practices for securing data in cloud services*. Microsoft Security Blog. https://www.microsoft.com/en-us/security/blog/2023/07/05/11-best-practices-for-securing-data-in-cloud-services/

Chkadmin. (2022, 31 marzo). *Cloud security best practices*. Check Point Software. https://www.checkpoint.com/cyber-hub/cloud-security/what-is-cloud-security/cloud-security-best-practices/

GeeksforGeeks. (2025a, julio 23). *Introduction to Docker for System Design*. GeeksforGeeks. https://www.geeksforgeeks.org/system-design/introduction-to-docker-for-system-design/

GeeksforGeeks. (2025b, julio 23). *Public Cloud vs Private Cloud vs Hybrid Cloud*. GeeksforGeeks. https://www.geeksforgeeks.org/devops/public-cloud-vs-private-cloud-vs-hybrid-cloud/

Public Cloud vs Private Cloud vs Hybrid Cloud | Microsoft Azure. (s. f.). https://azure.microsoft.com/en-gb/resources/cloud-computing-dictionary/what-are-private-public-hybrid-clouds/

qodo. (2025, 14 febrero). *What is Infrastructure as Code? Benefits & How it Works*. Qodo. https://www.qodo.ai/glossary/infrastructure-as-code/

Straub, E. (2024, 14 octubre). *What is Infrastructure as Code? Benefits and Key Concepts — Infrastructure System*. Infrastructure System. https://www.infrastructure-system.com/tech-journal/what-is-infrastructure-as-code-benefits-and-key-concepts

Susnjara, S. (2025, 22 julio). Public cloud vs private cloud vs hybrid cloud. *IBM*. https://www.ibm.com/think/topics/public-cloud-vs-private-cloud-vs-hybrid-cloud

Weiss, T. R. (2024, 4 noviembre). Exploring Docker for DevOps: What It Is and How It Works | Docker. *Docker*. https://www.docker.com/blog/docker-for-devops/

«What is Docker?» (2024, 10 septiembre). Docker Documentation. https://docs.docker.com/get-started/docker-overview

Caso práctico

Cree un diseño de arquitectura para una aplicación nativa de nube considerando los siguientes componentes:

- Frontend: Una aplicación web que los clientes utilizaran para la navegación.
- Backend: Servicios que se comunican con la base de datos y el frontend.
- Base de datos: Un sistema de gestión de base de datos que almacene información.
- Almacenamiento de objetos: Para gestionar imágenes y contenido estático.

Diseño:

- Seleccione un proveedor de servicios en la nube (AWS, Azure o GCP) y sustente su selección.
- Diseñe una arquitectura de nube. Incluya diagramas que representen la arquitectura y justifique sus decisiones de diseño.

I. Selección del proveedor (Azure)

Escogí Azure porque ofrece un conjunto maduro de servicios PaaS y Kubernetes gestionado que facilitan construir aplicaciones nativas de nube siguiendo las buenas prácticas del *Azure Well-Architected Framework* (seguridad, fiabilidad, eficiencia y optimización de costes). Azure proporciona integración nativa entre red de borde (*Front Door/CDN*), orquestación de contenedores (AKS), almacenamiento de objetos (*Blob Storage*), bases de datos gestionadas y servicios de seguridad/observabilidad (Key Vault, Azure Monitor), lo que acelera el *time-to-market* y reduce la complejidad operativa en producción.

II. Arquitectura propuesta (Resumen funcional)

Los clientes acceden al frontend a través de Azure Front Door, que actúa como gateway global, CDN y WAF en el borde para optimizar latencia, balanceo y protección contra ataques de capa 7. El frontend está servido como una *Single Page Application* estática alojada en Azure Static Web Apps para despliegues rápidos y coste eficiente; el contenido estático se sirve desde Blob Storage y queda cacheado en el CDN/Front Door. Las APIs del backend corren como microservicios en un clúster gestionado AKS (contenedores) para obtener observabilidad, autoscaling y control de despliegues; si el equipo es pequeño y busca simplicidad, se puede optar por Azure App Service como alternativa PaaS menos operativa.

Para datos relacionales utilizo Azure SQL Database (o Azure Database for PostgreSQL/MySQL según stack), y para casos que requieren baja latencia global o modelo NoSQL recomiendo Azure Cosmos DB: la elección entre SQL relacional y Cosmos depende del modelo de datos, consistencia y distribución geográfica de la aplicación.

Todo acceso a secretos y certificados se realiza mediante Azure Key Vault, y la telemetría (métricas, logs, traces) se centraliza en Azure Monitor con alertas y dashboards.

III. Justificación técnica

Azure Front Door reduce latencia y ofrece protección WAF y failover global, por lo que mejora experiencia y resiliencia del frontend. Servir la SPA desde Blob Storage / Static Web Apps es económico y permite cacheo en el borde; Blob Storage es la solución de objetos nativa optimizada para contenido no estructurado.

Para el backend propongo AKS si se buscan microservicios, control sobre el runtime y escalado horizontal dinámico; App Service[1] sería la opción preferida cuando la simplicidad operacional y menor overhead de Kubernetes son prioritarios. En la capa de datos, Azure SQL Database ofrece servicios PaaS robustos y familiares para cargas transaccionales, mientras que Cosmos DB es recomendado para escenarios que requieren replicación global, latencia sub-10ms y escalado masivo de throughput. Key Vault centraliza la gestión de secretos y reduce el riesgo de exposición; Azure Monitor permite observabilidad completa y respuesta ante incidentes.

a. Alta disponibilidad, seguridad y escalabilidad (breve)

La solución obtiene alta disponibilidad distribuyendo puntos de presencia en el borde con Front Door y replicando bases de datos (cuando se use Cosmos o SQL con geo-replicas). AKS con Horizontal Pod Autoscaler y Cluster Autoscaler permite escalar servicios según CPU/latencia/colas.

La seguridad se refuerza con políticas RBAC, redes privadas (AKS en VNet), subredes, NSGs, WAF en Front Door, y secretos en Key Vault (aquí se incluirían Access policies para el acceso y gestión de los secretos, y dichas políticas irían atachadas a las *system-assigned identities* de los recursos que necesiten acceder a estos); cifrado en reposo/tránsito es aplicado por los servicios gestionados.

[1] Siendo honesto, prefiero trabajar con Web Apps antes que con AKS, pero todo dependerá del objetivo final del proyecto y de los requisitos del cliente.

Para backups y recuperación planifica snapshots automatizados de Azure SQL y políticas de versionado + lifecycle en Blob Storage. El objetivo final de este diseño es seguir los pilares del Well-Architected Framework para equilibrar coste, fiabilidad y seguridad.

b. Siguientes pasos prácticos (roadmap corto)

1. Implementar el infrastructure as code (Bicep o Terraform) que cree: Storage Account (Blobs), Front Door + CDN, AKS + ACR, Azure SQL/Cosmos con backups, Key Vault y roles.
2. Definir pipeline CI/CD (build → push a ACR → helm/kustomize deploy a AKS).
3. Configurar Azure Monitor + alertas y pruebas de carga para ajustar autoscaling y caching.
4. Validar requisitos de consistencia y latencia para seleccionar SQL vs Cosmos y correr un proof-of-concept con datos reales para estimación de costes.

IV. Diagrama de infraestructura

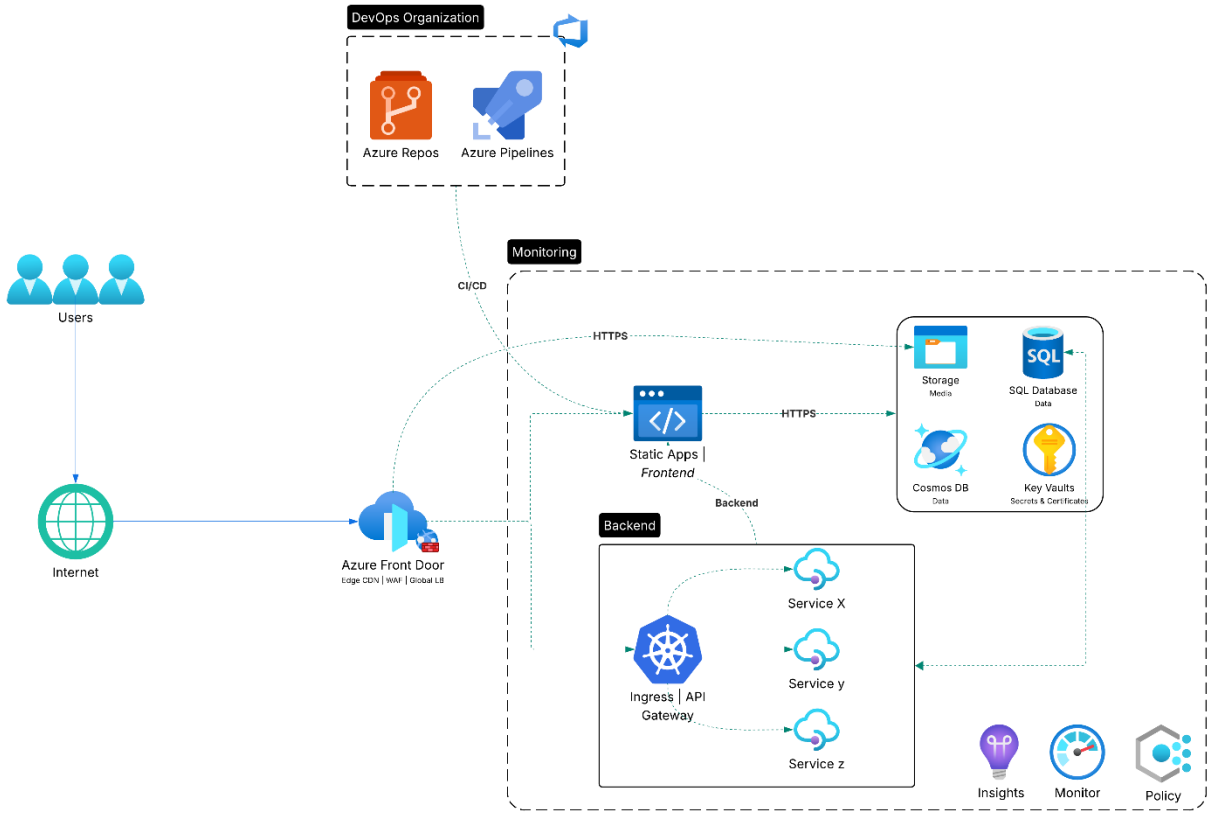

***Figura 1**. Diagrama de infraestructura. Fuente: Elaboración propia (utilizando Lucidchart).*