Proceso de Pensamiento Paso a Paso:

1. **Paso 1: Entender los Flujos y Separar Dominios**
    - **Flujo Público:** Usuario anónimo -> Accede al sitio web -> Lee noticias. Características: Alto volumen, lectura intensiva, sensible a la latencia, necesita caché.
    - **Flujo Privado (Admin):** Editor -> Se loguea -> Accede a portal admin -> Crea/Edita/Elimina noticias. Características: Bajo volumen, escritura/modificación, requiere seguridad robusta (autenticación/autorización).
    - **Conclusión Inicial:** Necesitamos separar claramente la infraestructura y la lógica que sirve a estos dos flujos tan diferentes en cuanto a escala, seguridad y patrón de acceso (lectura vs escritura).
2. **Paso 2: Definir la Experiencia de Usuario y el Acceso (Frontend)**
    - *Requisito:* Sitios web estáticos.
    - *Tecnología Frontend:* Para construir interfaces modernas e interactivas, **React Native** para reutilización Web/Mobile es una elección sólida y estratégica (mantenemos un solo código fuente para ambos canales).
    - *Servir Estáticos Escalablemente:* ¿Cómo entregar estos archivos (HTML, CSS, JS compilados) de forma rápida y global? La solución PaaS estándar en AWS es **Amazon S3** para el almacenamiento (prácticamente infinito, barato, duradero) combinado con **Amazon CloudFront** como CDN (red de entrega de contenido). CloudFront cachea los archivos en puntos de presencia globales cerca de los usuarios, reduce latencia y carga en S3, y se integra con WAF para seguridad. Esto cumple los requisitos de PaaS, escalabilidad y rendimiento.
    - *Separación:* Necesitamos dos "paquetes" de código estático: appPublica y appPrivada. Ambos se alojarán en S3 (buckets separados) y se servirán a través de CloudFront.
3. **Paso 3: Definir el Contrato y Punto de Entrada del Backend (API First)**
    - *Requisito:* API REST, enfoque API First. El backend debe exponer una API clara.

- *Punto de Entrada Centralizado:* ¿Cómo acceden los frontends al backend de forma segura, gestionada y escalable? **Amazon API Gateway** es el servicio PaaS ideal. Actúa como fachada, gestiona endpoints REST, enrutamiento, validación, throttling, autorización, y escala automáticamente.
- *Separación Público vs. Privado:* Dado los flujos distintos, tiene sentido separar la exposición de la API. Una API pública para leer noticias y una API privada para la gestión.
    - API Pública: Alto tráfico, necesita caché (potencialmente), seguridad más laxa (quizás API Keys).
    - API Privada: Bajo tráfico, requiere autenticación/autorización fuerte.
- *Decisión:* Implementar **dos API Gateways** (api-Gateway-publico, api-Gateway-privado). Esto permite configurar políticas de seguridad, caché, y throttling de forma independiente y clara.

4. **Paso 4: Abordar la Seguridad**
    - *Autenticación Admin:* ¿Cómo gestionamos el login de los editores? **Amazon Cognito** es el servicio PaaS de AWS para gestión de identidades. Permite crear un directorio de usuarios (User Pool), manejar el flujo de login, y se integra directamente con API Gateway para autorizar las llamadas a la API privada.
    - *Seguridad Perimetral:* ¿Cómo protegemos los puntos de entrada públicos (CloudFront, API Gateway) de ataques comunes (DDoS, SQL Injection, XSS, bots)? **AWS WAF** es el firewall de aplicaciones web que se integra con ambos servicios para filtrar tráfico malicioso en el borde.
    - *Gestión de Secretos:* ¿Dónde guardar de forma segura claves de API, credenciales, etc.? **AWS Secrets Manager** es el servicio PaaS para almacenar y rotar secretos, evitando hardcodearlos.
5. **Paso 5: Elegir la Plataforma de Cómputo del Backend**
    - *Requisito:* Ejecutar la lógica de la API REST. Maximizar PaaS. Escalar masivamente.
    - *Opciones:* EC2 (IaaS, mucha gestión), Contenedores (ECS/EKS/Fargate - PaaS/IaaS híbrido), Serverless (Lambda - PaaS puro).

- *Decisión:* **AWS Lambda**. Se alinea perfectamente con los requisitos: es totalmente PaaS, escala automáticamente por petición (ideal para picos variables), pagas por ejecución (costo-eficiente), y se integra nativamente con API Gateway y otros servicios AWS.

6. **Paso 6: Estructurar la Lógica del Backend (API First, Escalabilidad Lectura)**
    - *Patrón BFF:* Para adaptar las respuestas del backend a las necesidades específicas de cada frontend (appPublica, appPrivada) y seguir el enfoque API-First, introducimos el patrón **Backend for Frontend (BFF)**. Implementamos bff-privado y bff-publico como funciones **Lambda** situadas entre API Gateway y la lógica de negocio principal. Optimizan llamadas y simplifican los frontends.
    - *Lógica de Negocio - Monolito vs. Microservicios:* ¿Cómo organizar el código que realmente interactúa con los datos?
        - *Consideración Clave:* El requisito de manejar **picos masivos de LECTURA** de noticias, que debe ser independiente de las operaciones de ESCRITURA (CRUD) de bajo volumen.
        - *Decisión:* Aplicar un patrón similar a **CQRS (Command Query Responsibility Segregation)** separando la lógica en (al menos) dos **Microservicios** implementados como Lambdas: ms-list-news (optimizado para consultas/listados, escalará masivamente) y ms-news-crud (maneja creación, edición, borrado). Esto asegura que la alta carga de lectura no impacte la administración y viceversa. Es una decisión estructural clave para la escalabilidad. (Dentro de estas Lambdas, aplicaríamos buenas prácticas como Clean Architecture, DDD, TDD).
7. **Paso 7: Seleccionar Almacenamiento de Datos Persistentes**
    - *Requisito:* Guardar noticias e imágenes. Escalable, PaaS.
    - *Datos de Noticias:* Necesitamos acceso rápido (ej: por ID de noticia) y listados eficientes (ej: por fecha). Debe escalar para soportar millones de lecturas desde ms-list-news.
        - *Opciones:* RDS (Relacional, menos elástico/serverless), NoSQL.
        - *Decisión:* **Amazon DynamoDB**. Base de datos NoSQL serverless nativa de AWS, diseñada para escala masiva, baja latencia en

accesos por clave, se integra perfectamente con Lambda y API Gateway. Cumple los requisitos, aunque exige pensar bien el modelado de datos según los patrones de acceso.

- *Imágenes y Archivos Binarios:* Almacenamiento de objetos.
  - *Decisión:* **Amazon S3**. Es el estándar de facto. Escalable, duradero, económico, y se integra perfectamente con CloudFront para la entrega eficiente.

8. **Paso 8: Optimizar Rendimiento y Resiliencia (Caché y Asincronismo)**
   - *Optimización de Lecturas:* DynamoDB puede ser costoso o sufrir throttling bajo lecturas extremadamente intensivas y repetitivas desde ms-list-news.
     - *Decisión:* Introducir una capa de **caché en memoria** en el backend. **Amazon ElastiCache for Redis** es la solución PaaS. Se coloca entre ms-list-news y DynamoDB, usando el patrón cache-aside. Reduce drásticamente la carga en DynamoDB, baja costos y mejora la latencia.
   - *Desacoplamiento de Tareas Secundarias:* Operaciones como invalidar la caché de Redis/CloudFront tras una actualización en ms-news-crud, o futuras notificaciones, no deberían bloquear la respuesta al editor.
     - *Decisión:* Implementar **Procesamiento Asíncrono**. ms-news-crud publica un evento (ej: "NoticiaActualizada") en **Amazon EventBridge** (bus de eventos). Una regla en EventBridge envía el evento a una cola **Amazon SQS**. Lambdas separadas consumen mensajes de SQS para realizar las tareas secundarias (invalidar caché, etc.). Esto desacopla los servicios, mejora la resiliencia (SQS maneja reintentos, DLQ para errores) y la escalabilidad.
9. **Paso 9: Implementar Observabilidad**
   - *Necesidad:* En una arquitectura distribuida y serverless, es vital poder monitorizar, depurar y entender el rendimiento y los costos.
   - *Decisión:* Utilizar las herramientas PaaS estándar de AWS: **Amazon CloudWatch** para Logs, Métricas y Alarmas, y **AWS X-Ray** para Tracing distribuido (fundamental para seguir una petición a través de API Gateway, Lambdas, DynamoDB, entre otros).

10. **Paso 10: Definir Infraestructura y Despliegue**

    - *Gestión de Infraestructura:* Evitar la configuración manual propensa a errores.
        - *Decisión:* **Infraestructura como Código (IaC)** usando **Terraform**. Permite definir toda la arquitectura en código, versionarla y aprovisionarla de forma automatizada y repetible.
    - *Automatización de Despliegues:* Entregar cambios de código e infraestructura de forma fiable.
        - *Decisión:* Un pipeline de **CI/CD** usando **AWS CodePipeline / AWS CodeBuild** (o equivalentes). Se integra con el control de versiones (Git) para automatizar las pruebas y los despliegues (ejecutando Terraform, desplegando código Lambda, subiendo builds de React a S3).

11. **Paso 11: Iteración y Refinamiento (Incluyendo DR)**

    - Revisar la arquitectura completa contra los requisitos. ¿Hay cuellos de botella? ¿Falta algo? ¿Es coherente? Es en esta fase donde se refinan detalles, se consideran alternativas y se incorporan requisitos no funcionales adicionales como la **Recuperación ante Desastres (DR)**, llevándonos a diseñar la estrategia Pilot Light como una capa adicional sobre la arquitectura de producción con RTO y RPO objetivo 1 hora considerando que un sito de noticias debe recuperarse muy rápido para continuar con el servicio a los clientes