Proyecto-ICP

Pablo Gonzálbez Cabo

Febrero 2026

Abstract

Este proyecto presenta el diseño e implementación de un pipeline *serverless* en Amazon Web Services (AWS) orientado a la automatización del proceso de compresión y archivado de ficheros. El sistema se activa automáticamente ante la subida de un objeto a un bucket de Amazon S3 y orquesta, mediante AWS Step Functions, una secuencia de funciones AWS Lambda encargadas de inspeccionar, copiar, comprimir y almacenar el fichero resultante. El proceso finaliza con el envío de una notificación al usuario a través de Amazon SNS.

La solución se ha implementado íntegramente mediante infraestructura como código, utilizando AWS CloudFormation como tecnología principal de despliegue y una réplica equivalente desarrollada en AWS CDK basada en constructos de bajo nivel. Además, se analiza el rendimiento de compresión según el tipo de fichero, destacando una mayor efectividad en datos textuales (por ejemplo, `.txt` y `.csv`) frente a formatos ya comprimidos (por ejemplo, imágenes), y se proponen mejoras y ampliaciones alineadas con buenas prácticas cloud (por ejemplo, políticas de ciclo de vida en S3 para borrado automático y archivado en *Deep Archive*).

1 Introducción

La automatización de procesos de tratamiento de datos es una necesidad recurrente en entornos cloud modernos, especialmente en escenarios donde se manejan grandes volúmenes de ficheros y se requiere una respuesta rápida, escalable y con un coste ajustado. En este contexto, los servicios *serverless* proporcionan un modelo de desarrollo que elimina la gestión directa de infraestructura y permite centrar el diseño en la lógica del proceso.

Amazon Web Services (AWS) ofrece un conjunto de servicios gestionados que facilitan la construcción de este tipo de soluciones, permitiendo implementar flujos de trabajo reactivos basados en eventos, altamente desacoplados y con escalabilidad automática. Entre estos servicios destacan Amazon S3 como sistema de almacenamiento, AWS Lambda para la ejecución de código bajo demanda y AWS Step Functions para la orquestación de procesos complejos.

Este proyecto se enmarca dentro de dicho paradigma y tiene como finalidad el desarrollo de un sistema automatizado para la compresión y archivado de ficheros en AWS. A través de un enfoque basado en eventos, el sistema procesa los objetos subidos a un bucket de entrada y los transforma de manera transparente para el usuario, garantizando un flujo reproducible, mantenible y alineado con buenas prácticas de diseño cloud.

Adicionalmente, durante las pruebas se observa que la efectividad de la compresión depende fuertemente del tipo de fichero: los contenidos con alta redundancia (por ejemplo, ficheros de texto plano `.txt` y datos tabulares `.csv`) presentan ratios de compresión significativamente mejores que formatos que suelen estar previamente comprimidos (por ejemplo, imágenes). Esta observación orienta posibles extensiones del proyecto hacia estrategias adaptativas de compresión por tipo de contenido.

La memoria describe el problema abordado, las decisiones técnicas adoptadas, la arquitectura implementada y el funcionamiento detallado del sistema, incluyendo una demostración práctica y un apartado de mejoras potenciales para enriquecer la solución.

2 Problema a resolver

En numerosos escenarios reales es necesario procesar ficheros de forma automática tras su subida a un sistema de almacenamiento: compresión para reducir costes, archivado, preparación para su distribución o integración con otros sistemas. Implementar este tipo de procesos de forma manual o mediante servidores tradicionales introduce problemas de escalabilidad, mantenimiento, costes fijos y complejidad operativa.

El problema que se pretende resolver en este proyecto es la automatización completa del proceso de compresión y archivado de ficheros en AWS, cumpliendo los siguientes requisitos:

- Activación automática del flujo al subir un fichero.
- Procesamiento sin intervención humana.
- Uso exclusivo de servicios gestionados (*serverless*).
- Separación clara de responsabilidades entre las distintas fases.
- Notificación al usuario al finalizar el proceso.

Además, el sistema debe ser fácilmente desplegable y eliminable, garantizando la reproducibilidad del entorno mediante scripts y plantillas de infraestructura como código.

3 Justificación técnica de la arquitectura

La arquitectura elegida se basa en un enfoque completamente *serverless*, lo cual se justifica por varios motivos técnicos y operativos:

3.1 Escalabilidad y elasticidad

Algunos de los servicios utilizados (AWS Lambda, Step Functions y S3) escalan automáticamente en función de la carga, sin necesidad de planificación previa de capacidad. Esto permite que el sistema pueda manejar desde un número reducido de ficheros hasta cargas elevadas sin modificaciones estructurales.

Adicionalmente, se utiliza SNS para soportar picos de publicación/entrega sin que el usuario gestione infraestructura.

3.2 Modelo de costes

El modelo de pago por uso propio de los servicios serverless evita costes fijos asociados a instancias en ejecución permanente. El sistema únicamente genera costes cuando se producen eventos reales (subida de ficheros y ejecución del pipeline).

3.3 Simplicidad operativa

Al no existir servidores que administrar, se eliminan tareas relacionadas con actualizaciones, parches, monitorización de instancias o gestión de sistemas operativos. La lógica del negocio se encapsula en funciones Lambda pequeñas y especializadas.

3.4 Desacoplo y mantenibilidad

Cada etapa del proceso se implementa como una función independiente, orquestada mediante Step Functions. Este desacoplo facilita el mantenimiento, las pruebas y la evolución futura del sistema, permitiendo modificar o ampliar fases concretas sin afectar al conjunto.

3.5 Infraestructura como código

Toda la infraestructura está definida mediante plantillas de AWS CloudFormation. Adicionalmente, se ha desarrollado una réplica equivalente utilizando AWS CDK en TypeScript, basada exclusivamente en constructos de bajo nivel (*Cfn**), con el objetivo de garantizar una correspondencia exacta con las plantillas de CloudFormation. No obstante, esta versión no pudo ser validada en ejecución debido a limitaciones de permisos de despliegue de Stacks de CloudFormation en la cuenta AWS disponible (únicamente pudimos desplegar Stacks desde la consola), mientras que la versión CloudFormation sí fue desplegada y probada correctamente.

4 Arquitectura y funcionamiento del sistema

4.1 Visión general

El sistema implementa un flujo de procesamiento activado por eventos que comienza con la subida de un fichero a un bucket de entrada en Amazon S3. Este evento dispara una función Lambda que inicia una máquina de estados de AWS Step Functions, la cual orquesta de forma secuencial las distintas etapas del pipeline.

Las fases del proceso son las siguientes:

1. Inspección del objeto subido.
2. Copia del fichero a un bucket de artefactos.
3. Compresión del fichero mediante `gzip`.
4. Almacenamiento del resultado final.
5. Notificación al usuario por correo electrónico.

4.2 Funciones Lambda y responsabilidades

El sistema está compuesto por varias funciones AWS Lambda, cada una con una responsabilidad claramente definida:

- **StartStateMachine**: se activa automáticamente mediante un evento `ObjectCreated` de Amazon S3. Extrae el nombre del bucket y la clave del objeto, e inicia la ejecución de la Step Function pasando estos valores como entrada.
- **InspectObject**: realiza una operación `headObject` sobre el fichero original para obtener metadatos como el tamaño y el tipo de contenido.
- **PrepareArtifact**: copia el objeto original al bucket de artefactos, utilizando el prefijo `artifacts/`, estableciendo un área de trabajo intermedia.
- **ExtremeCompression**: descarga el fichero desde el bucket de artefactos, lo comprime utilizando `zlib.gzip` y sube el resultado con extensión `.gz`, incluyendo metadatos relativos a la compresión.
- **StoreFinalObject**: copia el fichero comprimido al bucket de salida definitivo, bajo el prefijo `final/`.
- **NotifyUser**: publica un mensaje de finalización en un tema de Amazon SNS, que a su vez envía un correo electrónico al usuario suscrito.

4.3 Orquestación con Step Functions

AWS Step Functions actúa como elemento central de coordinación del sistema. La máquina de estados define explícitamente la secuencia de ejecución:

$$\text{InspectObject} \rightarrow \text{PrepareArtifact} \rightarrow \text{ExtremeCompression} \rightarrow \text{StoreFinalObject} \rightarrow \text{NotifyUser}$$

Este enfoque permite:

- Visualizar el estado del proceso en todo momento.
- Gestionar errores y reintentos de forma declarativa.
- Mantener un historial de ejecuciones para auditoría y depuración.

4.4 Intercambio de datos y uso de buckets

Los objetos no se transfieren directamente entre funciones Lambda. En su lugar, se utiliza Amazon S3 como mecanismo de intercambio de datos entre etapas. Cada función escribe o lee objetos desde los buckets correspondientes, mientras que la Step Function únicamente intercambia referencias ligeras (nombre del bucket y clave del objeto). Este diseño reduce el tamaño de los mensajes intercambiados y evita limitaciones asociadas al paso de grandes volúmenes de datos en memoria.

Además, cuando el flujo requiere *mover* objetos entre buckets (por ejemplo, de *input* a *artifacts* y de *artifacts* a *output*), se utiliza una copia *server-side* de S3 (operación `copyObject`). Este enfoque evita descargar el fichero a la Lambda y volver a subirlo, reduciendo tiempo de ejecución, consumo de memoria y tráfico de red. La descarga completa del objeto solo se realiza en la etapa de compresión, donde es estrictamente necesaria para generar el `.gz`.

La utilización de múltiples buckets responde a una separación clara de responsabilidades:

- **Bucket de entrada**: ingestión de ficheros originales.
- **Bucket de artefactos**: almacenamiento intermedio de trabajo.
- **Bucket de salida**: resultados finales comprimidos.

4.4.1 Justificación del bucket de artefactos (buenas prácticas y trade-off)

En este proyecto se incorpora un bucket de *artifacts* como área de trabajo intermedia, lo cual responde a patrones habituales en pipelines de procesamiento en cloud. Aunque para un caso mínimo podría comprimirse directamente desde el bucket de entrada y escribir el resultado en el bucket de salida, el uso de artefactos aporta ventajas operativas que son relevantes como buenas prácticas:

- **Trazabilidad y auditoría**: permite conservar estados intermedios si se necesita inspeccionar el flujo o reproducir incidencias.
- **Reintentos sin tocar el original**: ante fallos, el pipeline puede reintentarse desde el artefacto sin sobrescribir o depender del objeto original.
- **Separación de dominios**: distingue claramente *input* (origen), *work-in-progress* (artefactos) y *output* (resultado), reduciendo riesgos de mezcla accidental.
- **Políticas diferenciadas**: permite aplicar políticas específicas (por ejemplo, borrado rápido de artefactos, retención en output, etc.).

No obstante, esta decisión también introduce un coste adicional (una copia extra y almacenamiento temporal). Por ello, en escenarios de muy baja complejidad o con restricciones estrictas de coste/tiempo, podría considerarse una simplificación eliminando el bucket de artefactos. En este proyecto se mantiene con el objetivo de justificar y practicar un diseño más alineado con arquitecturas escalables y mantenibles.

5 Análisis del comportamiento de compresión según el tipo de fichero

Durante las pruebas realizadas se observa una diferencia clara en el ratio de compresión alcanzado dependiendo del contenido del objeto:

- **Ficheros de texto y datos tabulares (`.txt`, `.csv`)**: suelen presentar alta redundancia (patrones repetidos, separadores, números en texto, etc.). En consecuencia, `gzip` ofrece ratios de compresión elevados, siendo un caso de uso especialmente adecuado para este pipeline.
- **Imágenes y formatos ya comprimidos**: muchos formatos comunes de imagen (y otros binarios) incorporan compresión interna o baja redundancia efectiva, por lo que aplicar `gzip` puede producir mejoras pequeñas o incluso marginales. En estos casos, el coste de computación puede no compensar la ganancia obtenida.

En el flujo actual, el ratio de compresión observado no siempre es especialmente alto, lo cual es coherente cuando el contenido de entrada incluye formatos poco compresibles o ya comprimidos. No obstante, se documenta también una prueba adicional con un ejemplo de compresión extrema (como se puede observar en la **Figura 1**), evidenciando que el pipeline puede ser extremadamente efectivo en datos textuales con alta redundancia:

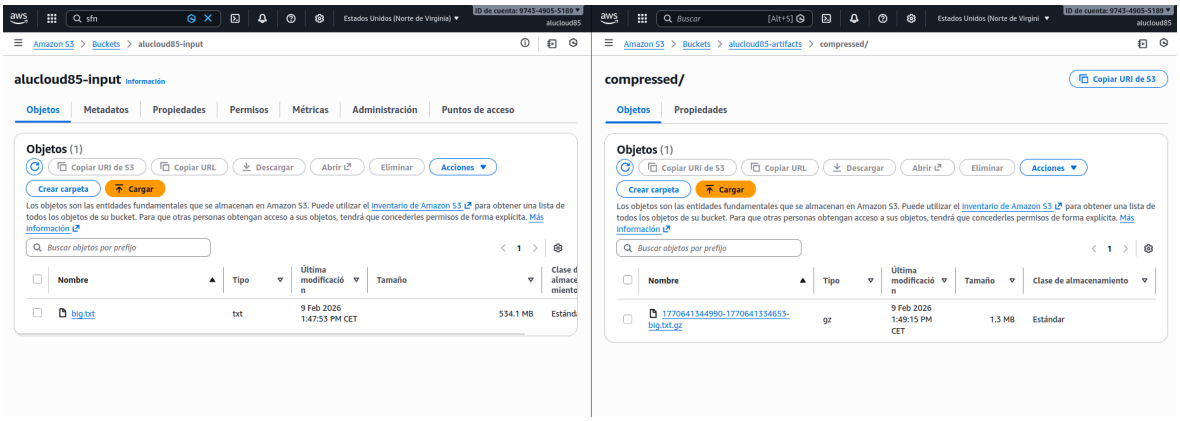

Figure 1: Ejemplo de compresión extrema

Esta observación sugiere que el pipeline está particularmente bien orientado a ficheros textuales (logs, exportaciones CSV, dumps de configuración, etc.) y que, si el objetivo fuese optimizar para imágenes u otros binarios, sería razonable adaptar la estrategia (por ejemplo, compresión distinta, criterios de *skip*, o transformaciones específicas por tipo de contenido).

6 Demostración del sistema

A continuación se describe la demostración funcional del sistema, acompañada de capturas sugeridas para su inclusión en el informe.

6.1 Despliegue del stack

El despliegue se realiza mediante el script `create-stack.sh`, el cual crea el stack `STACK-Proyecto-IPC-85` a partir de la plantilla principal almacenada en Amazon S3 y espera a su correcta finalización (como se puede observar en las **Figuras 2 y 3**).

Figure 2: Ejecución del script **create-stack.sh**

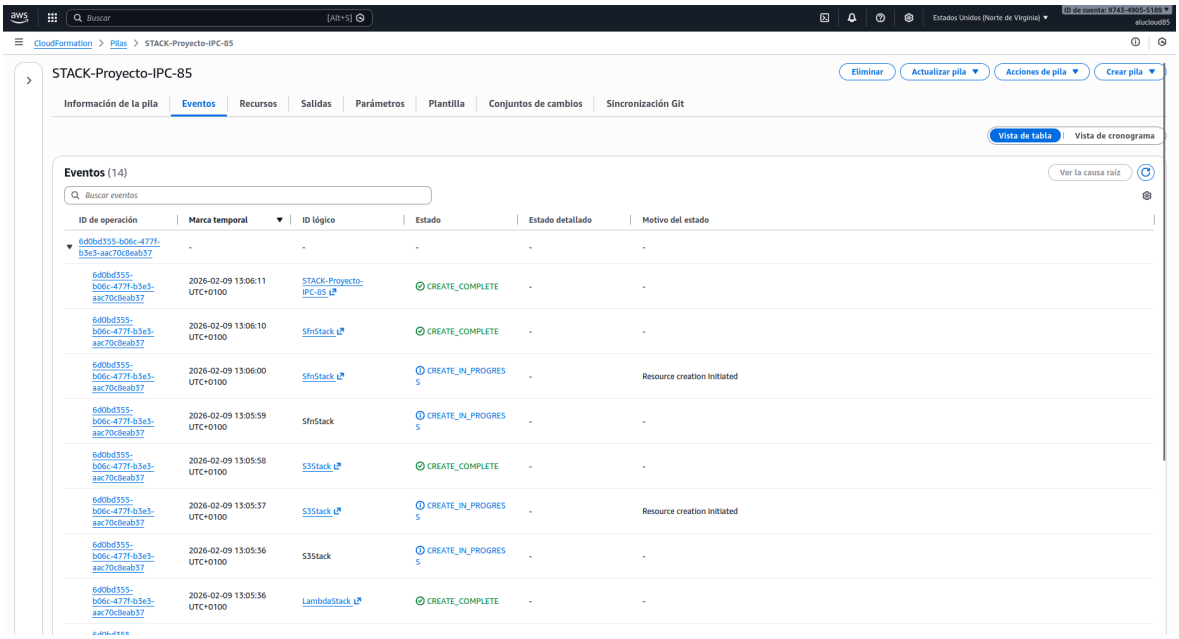

Figure 3: Captura de la consola de AWS con el Stack creado

6.2 Suscripción al Topic de SNS

En las **Figuras 4 y 5** se muestran el correo de la suscripción al Topic y la captura de la consola de AWS donde se comprueba que el resultado de la suscripción ha sido satisfactorio:

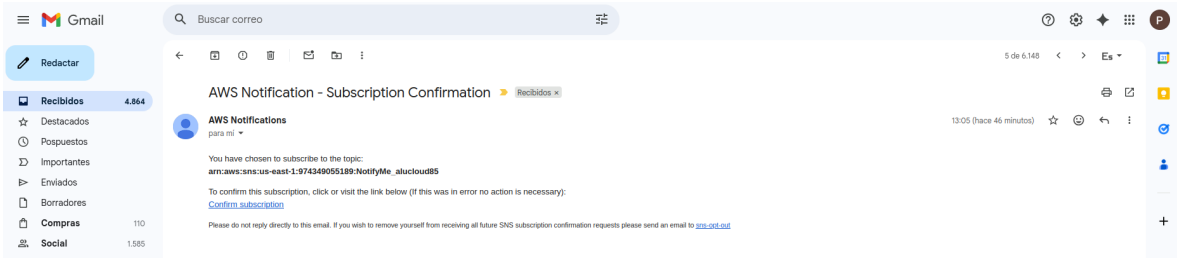

Figure 4: Captura del correo de suscripción al Topic

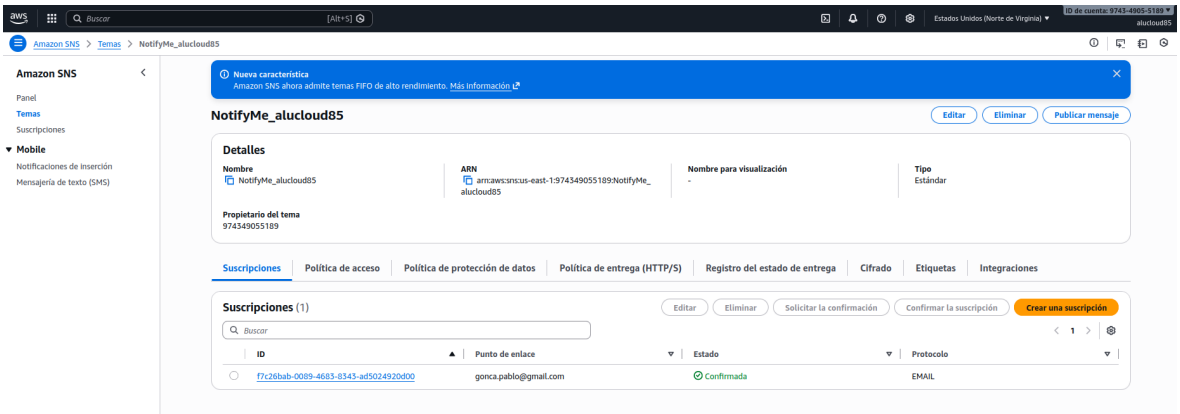

Figure 5: Captura del estado de la suscripción al Topic

6.3 Subida de un fichero al bucket de entrada

Se sube manualmente un fichero de prueba al bucket de entrada `*-input` (como se puede observar en la **Figura 6**). Este evento desencadena automáticamente la ejecución del pipeline.

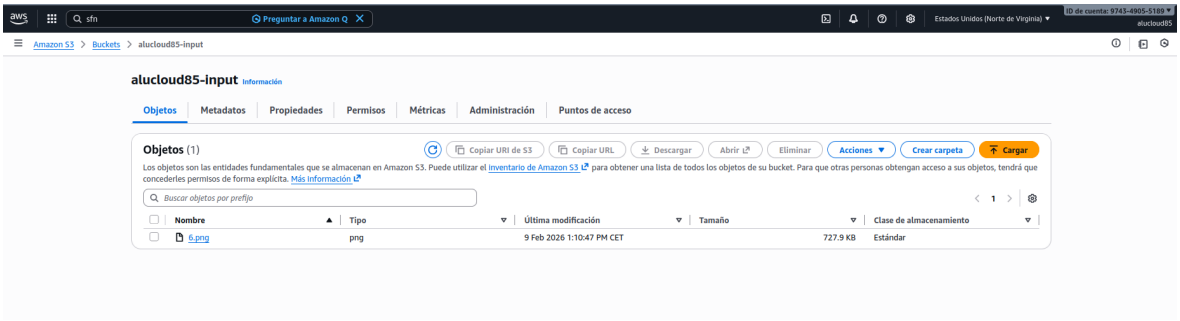

Figure 6: Subida de archivo al bucket de Input

6.4 Ejecución de la Step Function

La Step Function se ejecuta y muestra el avance secuencial por todos los estados definidos, desde la inspección hasta la notificación final (como se muestra en la **Figura 7**).

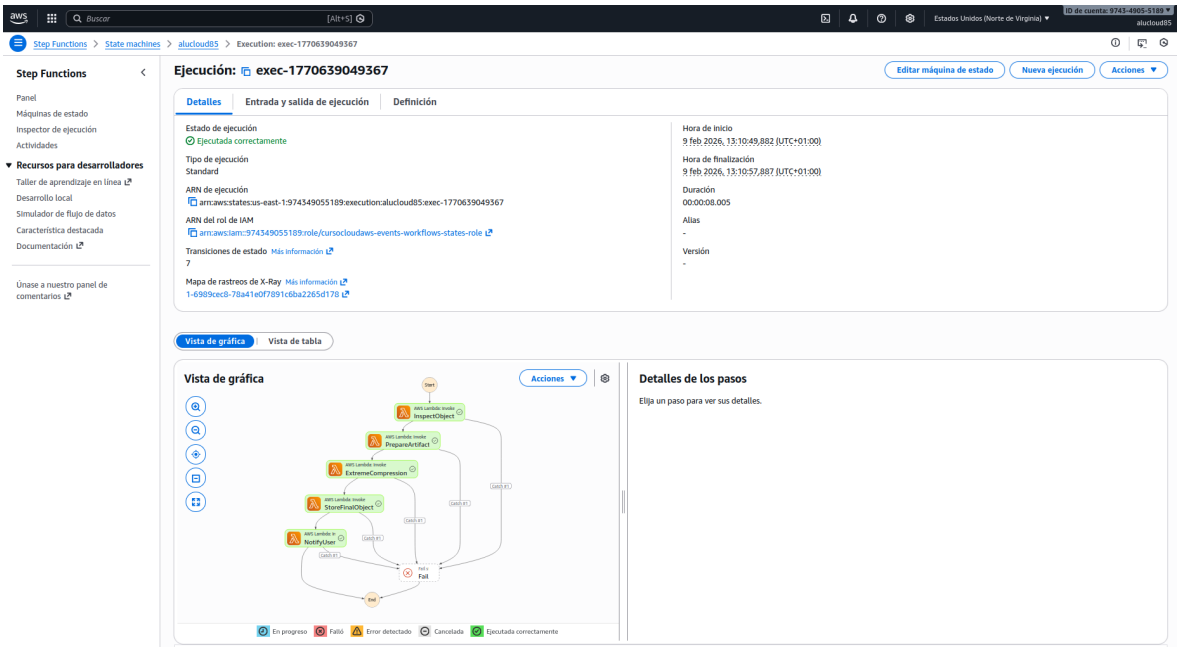

Figure 7: Vista gráfica de AWS Step Functions sin errores en la ejecución

6.5 Objeto comprimido en el bucket de salida

Una vez completado el proceso, el fichero comprimido con extensión `.gz` aparece en el bucket de salida, bajo el prefijo `final/` (como se puede observar en la **Figura 8**).

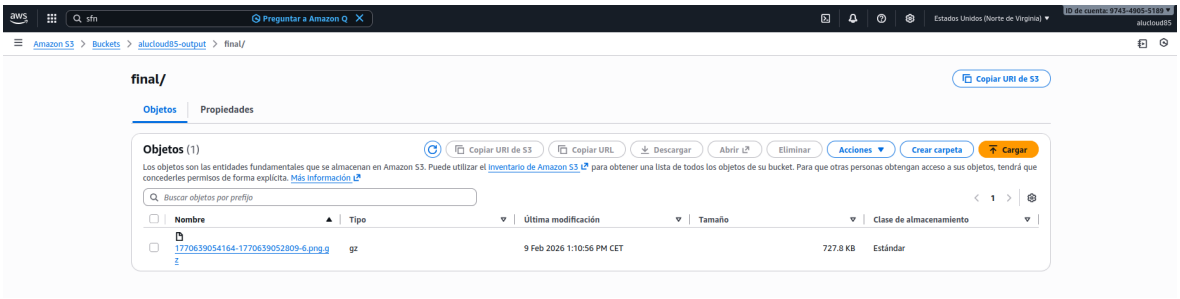

Figure 8: Captura de la consola de S3 mostrando el objeto comprimido y sus metadatos.

6.6 Notificación por correo electrónico

Finalmente, el usuario recibe un correo electrónico enviado a través de Amazon SNS, confirmando la finalización del proceso de compresión (como se muestra en la **Figura 9**).

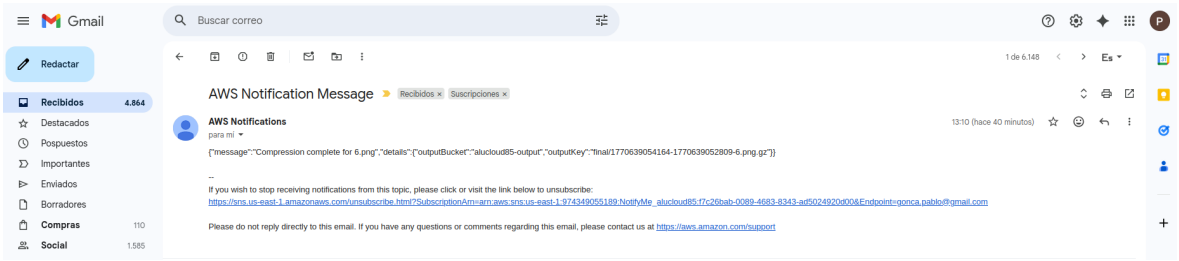

Figure 9: Captura de la bandeja de entrada del correo electrónico con el mensaje de notificación.

6.7 Borrado de recursos en AWS

A continuación, se observa la ejecución del script de borrado de recursos del proyecto (**Figura 10**).

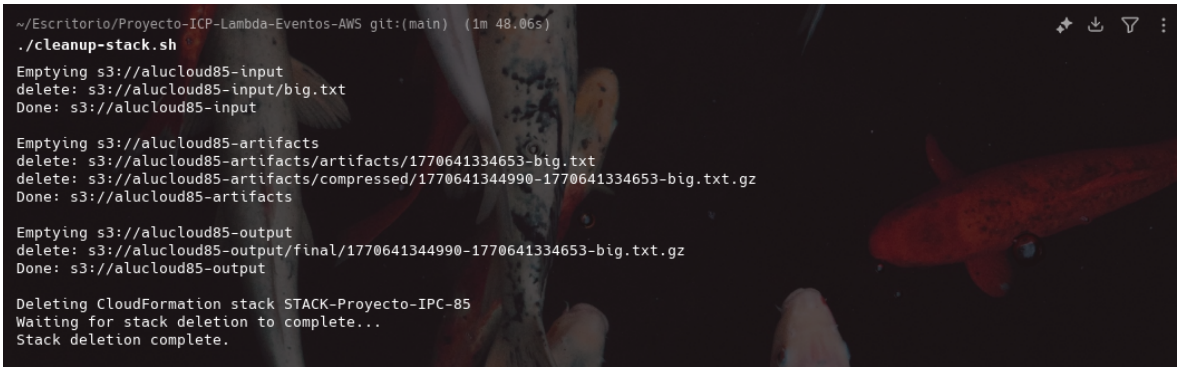

Figure 10: Ejecución del script **cleanup-stack.sh**

7 Posibles mejoras y ampliaciones

Aunque la implementación cumple el objetivo principal de la práctica (pipeline serverless de compresión y archivado), existen mejoras que permitirían elevar la solución hacia un diseño más robusto, eficiente y completo. A continuación se describen las principales propuestas.

7.1 Mejoras en la estrategia de compresión

En el estado actual, el pipeline aplica `gzip` de forma uniforme. Esta decisión simplifica la práctica, pero no necesariamente maximiza la eficiencia para todos los tipos de fichero. Algunas mejoras posibles serían:

- **Compresión adaptativa por tipo de contenido**: usando `Content-Type` (o extensiones) para decidir si conviene comprimir o no. Por ejemplo, priorizar `.txt` y `.csv` y omitir formatos típicamente ya comprimidos.
- **Ajuste de nivel de compresión**: `gzip` permite niveles que cambian el equilibrio entre CPU y ratio. Podría seleccionarse un nivel distinto en función del tamaño del objeto o del SLA de tiempo.
- **Algoritmos alternativos**: existen opciones como `zstd` o `brotli` que, según el caso, pueden mejorar ratio/velocidad. Su uso requeriría incluir binarios o capas adicionales en Lambda, y excede el alcance de la práctica, pero es una ampliación razonable.
- **Medición y trazabilidad del ratio**: almacenar en metadatos el tamaño original, tamaño comprimido y ratio, permitiendo analizar qué tipos de fichero se benefician realmente.

Se destaca que estas mejoras no son imprescindibles para el objetivo de la práctica, pero permiten justificar por qué la compresión actual puede resultar limitada en ciertos casos (por ejemplo, imágenes) y cómo podría evolucionarse el sistema.

7.2 Políticas de ciclo de vida (Lifecycle) en los buckets

Una ampliación directa, alineada con buenas prácticas de almacenamiento, consiste en aplicar políticas de ciclo de vida sobre los buckets para automatizar borrados y optimizar costes:

- **Borrado automático en input**: si el bucket de entrada es solo de ingestión temporal, podrían borrarse objetos tras X días (siempre que sea aceptable según requisitos).
- **Borrado rápido de artifacts**: el bucket de artefactos suele contener datos intermedios; es habitual aplicar retención corta (por ejemplo, 1–7 días) para evitar acumulación innecesaria.
- **Archivado en output**: para resultados finales con retención larga y acceso poco frecuente, se puede mover automáticamente a clases de almacenamiento más baratas. Por ejemplo, transicionar desde *Standard* a *Standard-IA* y finalmente a *Glacier Deep Archive*, reduciendo el coste de almacenamiento a largo plazo.

Estas políticas refuerzan el enfoque *serverless* y *low-ops*: el sistema reduce intervención manual y controla el crecimiento del almacenamiento sin procesos adicionales.

7.3 Robustez y observabilidad

Para un entorno más cercano a producción, serían recomendables mejoras relacionadas con control de errores y visibilidad:

- **Manejo explícito de errores**: añadir *Catch* y rutas de fallo en Step Functions, enviando una notificación diferenciada si el proceso falla.
- **Reintentos configurados**: aplicar políticas de reintento por tipo de error (por ejemplo, throttling de S3/SNS, fallos temporales).
- **Métricas y alarmas**: métricas de duración, errores y ratio de compresión; alarmas en CloudWatch para fallos recurrentes o tiempos anómalos.
- **Autorización de compresión**: notificación al usuario y estado de espera en Step Functions antes de ejecutar la compresión, representando una aprobación manual simplificada.

7.4 Mejoras de seguridad y control de acceso

Aunque el proyecto ya emplea cifrado y bloqueo de acceso público en S3, podrían añadirse:

- **Principio de mínimo privilegio**: roles IAM específicos por Lambda con permisos mínimos necesarios (en lugar de reutilización de roles genéricos).
- **Validación de entrada**: restringir el pipeline a prefijos concretos o tipos de objetos permitidos.
- **Integridad**: cálculo de hash (por ejemplo, SHA-256) antes y después para verificar consistencia del fichero procesado.

7.5 Mejoras funcionales del pipeline

- **Procesamiento por lotes**: disparar el pipeline con agrupación (batching) si se suben muchos objetos o una carpeta entera, reduciendo ejecuciones individuales.
- **Paralelización**: si se amplía a múltiples transformaciones, Step Functions permite ramas paralelas (por ejemplo, compresión + generación de metadatos).
- **Idempotencia**: garantizar que reejecutar el flujo con la misma entrada no genera duplicados (por ejemplo, claves deterministas y verificación previa).

8 Conclusiones

El proyecto demuestra la viabilidad y las ventajas de una arquitectura serverless para la automatización de pipelines de procesamiento en AWS. El uso combinado de S3, Lambda, Step Functions y SNS permite construir un sistema escalable, desacoplado y de bajo mantenimiento, alineado con buenas prácticas actuales de computación en la nube.

Las pruebas sugieren que el pipeline resulta especialmente efectivo en ficheros con alta redundancia (por ejemplo, `.txt` y `.csv`), mientras que en formatos previamente comprimidos (por ejemplo, imágenes) la ganancia puede ser limitada. Esta conclusión refuerza la idea de que, en escenarios reales, conviene adaptar la estrategia de compresión al tipo de contenido para maximizar eficiencia.

La definición completa de la infraestructura mediante CloudFormation, junto con su réplica equivalente en AWS CDK, refuerza la reproducibilidad y portabilidad del sistema. Aunque la versión CDK no pudo validarse por restricciones de permisos, su implementación en constructos de bajo nivel garantiza la equivalencia con la solución desplegada.

Finalmente, se identifican ampliaciones claras y alineadas con buenas prácticas, destacando el uso de políticas de ciclo de vida en S3 para controlar retención y costes (borrado automático y transición a *Deep Archive*), así como mejoras de robustez, observabilidad y seguridad que permitirían evolucionar el pipeline hacia un diseño más completo.