Adopción de nuevas tecnologías en la nube:
Evaluación comparativa de utilización de memoria y tiempos de ejecución para tareas automatizadas utilizando MyFEPS

Universidad de Belgrano
Facultad de Tecnología e Información
Año 2021

Carrera: Licenciatura en Informática

Autor:

- *Jinkus, Ivan*
  - *Matrícula 501-11220*
  - *jinkus.ivan@comunidad.ub.edu.ar*

Tutora: *Mg. Piccin, Ana*
ana.piccin@comunidad.ub.edu.ar

Agradecimientos

A la Mg. Ana Piccin, esta investigación no podría haberse desarrollado sin su ayuda, su valioso aporte, sus consejos, sus recomendaciones y sus críticas.

Al Mg. Sergio Aguilera, por su apoyo brindado a lo largo de la carrera.

A la Mg. Paula Angeleri, por su constante ayuda y aporte en las metodologías de trabajo y calidad.

Al PhD. Alejandro Mitaritonna, por su tiempo y ayuda brindada a lo largo del cuatrimestre, por su aporte crítico al trabajo que motivó la mejora continua del trabajo.

A los profesores, quienes durante estos años me instruyeron y educaron.

A mi familia, por su soporte cotidiano y apoyo incondicional.

A mis compañeros de todos estos años, con quienes compartí el aprendizaje, el tiempo dentro y fuera de las aulas, los logros y las frustraciones, y quienes en su día a día me ayudaron a llegar hasta aquí.

Índice

Índice de Figuras

Índice de Tablas

Resumen

En la investigación que se presenta se analiza la problemática asociada a la migración entre dos formas de implementación de tareas en la nube; una tradicional denominada IaaS (*Infrastructure as a Service*), que requiere la contratación de un servidor en la nube, y una emergente, denominada FaaS (*Function as a Service*), para la que no es necesario contratar la infraestructura de servidor en la nube. Se estudia, en particular, el caso de las tareas automatizadas recurrentes. Al tener infraestructura en la nube asociada, para que la tarea automatizada recurrente pueda ser ejecutada, si el servidor tiene una falla, entonces la tarea automatizada recurrente quedaría sin ser ejecutada. En cambio, si se implementara en un esquema sin servidores, ese problema quedaría eliminado. Para determinar la conveniencia o no de la migración se compararon y analizaron métricas operativas establecidas sobre la implementación de dos prototipos de tareas automatizadas recurrentes; uno con IaaS, con infraestructura asociada, y otro utilizando FaaS. Se consideraron tanto la utilización de memoria RAM como los tiempos de ejecución. Se obtuvieron conclusiones con respecto a alcances y limitaciones de uno y otros prototipos. Las implementaciones que se describen, así como los resultados obtenidos, permiten su réplica en situaciones similares para establecer criterios de adopción de las nuevas tecnologías en escenarios similares.

Introducción

Los productos de software requieren infraestructura subyacente para funcionar, ya sean servidores físicos o virtuales. Los servicios que permiten desplegar infraestructura en la nube reciben el nombre de IaaS (*Infrastructure as a Service*), traducido al español como “Infraestructura como Servicio”.

La evolución de tecnologías para implementaciones en la nube provocó una ampliación de la oferta de este tipo de servicios. Una de estas nuevas tecnologías es la denominada Faas (*Function as a Service*), traducido al español como “Funciones como Servicio”, conocido también como *Serverless* (*Sin Servidor*). En consecuencia, las empresas usuarias se enfrentan al desafío de establecer criterios que les permita decidir entre unas y otras alternativas, ya sea para nuevas implementaciones como para la conveniencia de migrar de unas a otras.

Este trabajo ofrece una comparación técnica entre ambas tecnologías. Se toma como caso de estudio el de las tareas automatizadas recurrentes de una empresa que ejecuta ETL (*Extract Transform Load*) una vez al día con una duración total menor de 15 minutos. Se trata de tareas automatizadas recurrentes que se ejecutan sin necesidad de intervención humana de forma regular; se pueden ejecutar diariamente, por hora, por minuto, o por cualquier otra frecuencia programada.

Para realizar la comparación técnica se implementaron dos prototipos, uno con infraestructura asociada y uno utilizando FaaS. Para evaluar y comparar la utilización de memoria y tiempos de ejecución de ambos prototipos, se utilizó MyFEPS [3]

En este trabajo se estudia la factibilidad y conveniencia de reemplazar infraestructura en la nube para tareas automatizadas recurrentes, por tecnología *serverless* en base a los recursos computacionales utilizados

.

1.1 Formulación del problema

Una tarea automatizada recurrente que se ejecuta cada 15 minutos y tiene una duración menor de 15 minutos, dejará tiempos en donde la infraestructura subyacente no esté en uso. Esta situación desaprovecha los recursos computacionales, ya que no se utiliza la capacidad de cómputo de la infraestructura contratada. Asimismo, existen tareas que pueden requerir altos niveles de potencia para ejecutar procesos cortos, como el caso de la ejecución de un algoritmo de inteligencia artificial sobre una fuente de datos. Considerando que siempre existirá un delta de tiempo ocioso, es necesario invertir en acciones para encontrar una alternativa eficiente que solucione este problema.

Como alternativa se puede adoptar una estrategia de programación de apagado (*Shutdown*) y encendido (*Start up*) de los equipos en la nube bajo demanda. Esta solución llevaría a implementar nuevos controles de infraestructura que aseguren el correcto apagado o encendido de los equipos generando nuevos potenciales puntos de falla.

Otra alternativa es el uso de *serverless*, una nueva tecnología que ofrece la ejecución de procesos a demanda sin necesidad de tener infraestructura asociada. Sin embargo, la planificación de un cambio de paradigma debe ser analizado para cada caso particular para evitar la pérdida de *performance* y asegurar que la utilización de recursos computacionales lo justifique.

De este problema deriva la necesidad de investigar la posibilidad de mejorar la utilización de recursos computacionales y eliminar infraestructura ociosa para tareas automatizadas recurrentes implementando una nueva tecnología que ha ganado adopción en los últimos años, llamada *serverless.*

Existen casos en donde empresas mencionan haber implementado este tipo de alternativa en la nube con éxito. No obstante, existen excepciones a la regla, como el caso de Bank of America [1]. Ninguna de las publicaciones ofrece resultados concretos producidos u observados en circunstancias comparables a tareas automatizadas recurrentes. Asimismo, estos casos no están respaldados por documentación publicada en sitios académicos que describan los detalles de la implementación y los resultados obtenidos.

1.2 Justificación del trabajo

Del problema planteado resulta la necesidad de analizar la alternativa *serverless* mencionada anteriormente para el caso de tareas automatizadas recurrentes implementados utilizando *IaaS*, con el fin de verificar si es posible mejorar la utilización de recursos computacionales sin impactar la funcionalidad del sistema.

La bibliografía consultada, tanto de referencia como auxiliar, desarrolla conceptos teoricos y practicos de *serverless* pero no permite asegurar que no haya

una alteración de la *performance* al adoptar esta tecnología ya que ninguna ofrece resultados producidos u observados en circunstancias comparables a las tareas automatizadas recurrentes.

Los casos mencionados no poseen documentación publicada donde se explique la implementación ni se describa su alcance, ni tampoco un respaldo académico que permita generalizar los resultados obtenidos [2].

Por estas razones se desea verificar que, para este tipo de escenarios, los recursos computacionales no son afectados por la implementación de esta tecnología y que el sistema cumple con los mismos requisitos funcionales que aquellos conseguidos en una arquitectura tradicional en la nube.

1.3 Objetivo general y específicos

El objetivo general de esta investigación es medir y cuantificar las diferencias en la utilización de recursos computacionales entre una implementación de tareas automatizadas recurrentes en un esquema tradicional en la nube comparado a un esquema *serverless*.

Se propone un trabajo de investigación que analice la implementación de un proceso automatizado recurrente dentro de dos arquitecturas en la nube, una con infraestructura tradicional y una *serverless*, midiendo y comparando la *performance* mediante la recolección de métricas de utilización de recursos computacionales.

Para lograr esto, se proponen los siguientes objetivos específicos:

- Verificar que la implantación del *serverless* mejore la utilización de los recursos computacionales.
- Verificar que la implantación del *serverless* no afecta la *performance* en la ejecución de la tarea.

1.4 Elaboración de hipótesis

La idea central de esta investigación es que las tareas automatizadas recurrentes subutilizan la infraestructura de instancias contratadas en la nube, ya que existe la posibilidad de que haya un intervalo en donde la tarea no se encuentra en ejecución.

Las tareas automatizadas recurrentes son comunes en el esquema de datos que mantienen las grandes empresas en la actualidad. Estos procesos utilizan recursos computacionales y requieren de infraestructura asociada. Como alternativa existen servicios *FaaS* para ejecutar las tareas sin necesidad de contratar infraestructura asociada.

Con la aparición de una nueva tecnología que no requiere infraestructura asociada para la ejecución de tareas, llamada *serverless*, surge la posibilidad de

migrar desde un esquema tradicional con infraestructura en la nube hacia arquitecturas sin servidores. Por esta razón se plantearon las siguientes preguntas:

En caso de migrar las tareas automatizadas recurrentes desde una arquitectura tradicional en la nube con infraestructura asociada hacia una arquitectura *serverless* que no tenga infraestructura asociada, ¿Se pierde *performance* en el sistema? ¿ Es necesario utilizar más memoria RAM al no poseer infraestructura asociada? ¿ Cómo se ven afectados los tiempos de ejecución en un esquema *serverless*?

Con respecto a estas preguntas, la posible mejora sobre la utilización de recursos computacionales resultante de la implementación de una alternativa más eficiente debería ser considerada.

Alex Yavorskiy, CTO (*chief technology officer)* en Financial Engines, asegura que “la *performance* de AWS *Lambda* es más predecible y estable que correr los procesos en servidores *on premises*” [2]. Si bien no está siendo comparado con una arquitectura en la nube, resulta interesante verificar que la afirmación de Yavorskiy se cumple para el escenario analizado en este trabajo, y que en efecto implementar *serverless* no genera un impacto sobre la utilización de recursos computacionales del sistema.

Teniendo esto en consideración, se formuló la siguiente hipótesis:

1.4.1 Hipótesis

La adopción de *serverless* para ejecutar tareas automatizadas recurrentes es más eficiente en la utilización de memoria y reduce los tiempos de ejecución en comparación a una arquitectura tradicional en la nube con infraestructura subyacente.

Mediante la implementación de una misma tarea automatizada recurrente tanto en una arquitectura tradicional en la nube con infraestructura subyacente como en un esquema *serverless,* podrá ser posible comparar de forma objetiva el resultado. Verificando métricas operativas, como la utilización de memoria y el tiempo de ejecución, será posible analizar si existe una pérdida de *performance* o si esta alternativa mejora la utilización de los recursos mencionados.

1.5 Método

Para cumplir con los objetivos previamente descritos se realizaron los siguientes pasos:

1. Implementar un prototipo en un esquema tradicional con infraestructura en la nube asociada para ejecutar una tarea automatizada recurrente.
2. Implementar un prototipo en un esquema *serverless* utilizando FaaS para ejecutar una tarea automatizada recurrente.
3. Analizar y comparar la utilización de memoria RAM y tiempos de ejecución de ambos prototipos.

4. Producir una tabla comparativa de utilización de memoria RAM y tiempos de ejecución.
5. Describir los resultados que surgen del análisis de la tabla comparativa entre ambas implementaciones.

Se implementaron en AWS dos prototipos para ejecutar una tarea automatizada recurrente: uno dentro de un esquema tradicional en la nube con infraestructura asociada y el otro en un esquema *serverless.* Ambos tienen los mismos parámetros de configuración para hacerlos equiparables.

En base a estos prototipos se establecieron parámetros de comparación que fueron utilizados para obtener resultados comparativos con respecto a la calidad computacional de las implementaciones. Luego, se produjeron tablas comparativas de métricas operativas, como utilización de memoria y tiempo de ejecución.

Tomando como referencia estos prototipos se realizaron mediciones que fueron utilizadas para comparar ambas soluciones y determinar ventajas y desventajas entre ellas, utilizando MyFEPS como métrica de eficiencia en la utilización de memoria interna (código **07.3.1.U**) [3]. Las referencias bibliográficas fueron realizadas de acuerdo a lo establecido por la IEEE, utilizando la documentación brindada por ellos en su sitio web oficial. [4]

1.6 Organización del documento

Este informe de investigación incluye las siguientes secciones:

En la sección 2, Marco teórico, se muestra el resultado del estudio bibliográfico para fundamentar la investigación de los prototipos desarrollados en la nube. En la sección 3 se describe el estado del arte. Esto incluye cómo se utilizan en la actualidad las tecnologías descritas. En la sección 4, Desarrollo, se describen los procesos realizados para cumplir con los objetivos de investigación y alcanzar la demostración de la hipótesis. Esta sección está dedicada a la implementación de los prototipos. La subsección 4.1 incluye el diseño de los prototipos implementados que conducen a la corroboración de la hipótesis; en la subsección 4.2 se encuentra la implantación de dichos prototipos; en la subsección 4.3 se desarrolla el análisis comparativo de las métricas recolectadas sobre la ejecución de los prototipos implementados. En la sección 5 se detallan los resultados obtenidos. El informe se completa con la sección 6, Conclusiones.

1.7 Alcance del trabajo

Este trabajo consiste en una investigación de implementación del concepto *serverless* como alternativa a un esquema tradicional con infraestructura en la nube para tareas automatizadas recurrentes.

El límite operativo de esta investigación está dado por una implementación práctica de *serverless* para simular una tarea automatizada recurrente que extrae información de un archivo y la transforma en base a cierta lógica computacional. Asimismo se implementó la misma tarea en un esquema tradicional. Ambos prototipos implementados se encuentran limitados por los servicios ofrecidos dentro de la capa gratuita del proveedor seleccionado.

El análisis y comparación de los prototipos implementados abarca el concepto de *serverless* y su implementación práctica como alternativa para ejecutar tareas automatizadas recurrentes sin necesidad de tener infraestructura asociada y sin percibir una pérdida de *performance.*

Las recomendaciones a futuro están limitadas por los resultados de la implementación operativa. Solo se recomiendan futuras acciones sobre escenarios de tareas automatizadas recurrentes de hasta 15 minutos de ejecución.

1.8 Antecedentes

Esta investigación está inspirada en casos publicados en la web como el de la compañía *Astrazeneca*, que utilizando servicios *serverless* ofrecidos por AWS implementó una solución para ejecutar diariamente 51 mil millones de pruebas de genomas sin necesidad de infraestructura tradicional. [5]

También el caso de *Chelsea*, equipo de fútbol de Inglaterra, que utilizando soluciones *serverless* de *machine learning*, elaboró una aplicación para ayudar en el aprendizaje de futuros futbolistas. [6]

El trabajo de Garett McGrath de la Universidad de Notre Dame, Indiana [7] inspiró esta investigación, y despertó la necesidad de avanzar en el análisis de servicios alternativos en la nube con el fin de mejorar la utilización de recursos computacionales.

Marco teórico

Es necesario conocer en detalle las tecnologías utilizadas en la implementación de *serverless e* indagar documentos oficiales para obtener información verosímil sobre el tema. El siguiente marco teórico desarrolla en profundidad todos los temas involucrados en esta investigación.

En esta sección se desarrollan conceptos críticos de la investigación, desde un abordaje técnico. Es importante entender cada uno de estos conceptos para poder desarrollar la implementación de *serverless* para tareas automatizadas recurrentes y compararla con una implementación en un esquema tradicional en la nube.

2.1 *Serverless*

Serverless es un tipo de arquitectura en la nube que no requiere contratación de infraestructura subyacente, donde el cliente contrata la ejecución de su código como servicio. Algunos ejemplos de estos servicios son: *"Lambda Functions"* de Amazon Web Services, "*Azure Functions"* de Microsoft Azure y *" Cloud Functions"* de Google Cloud Platform entre los más conocidos. Este tipo de arquitecturas tiene foco en dos pilares. El primero, que solo se debe pagar por los recursos que se utilizan, siguiendo un modelo "pay-per-use" [8]. El segundo, que la responsabilidad total de la infraestructura subyacente es del proveedor del servicio *serverless* contratado. [9]

En el artículo publicado por IEEE Internet Computing en 2018, titulado *Serverless is More: From PaaS to Present Cloud Computing,* se define *serverless* como "Un tipo de *cloud computing* que permite a usuarios correr aplicaciones granulares disparadas por eventos sin necesidad de administrar la lógica operacional". [10]

Es decir, mejora la utilización de recursos e infraestructura inactiva, y al mismo tiempo, se delega la responsabilidad de administrar toda la infraestructura subyacente al proveedor del servicio.

Este tipo de arquitectura puede ser utilizada para distintos escenarios, algunos de los cuales son desarrollados en la sección Estado del arte. Esta investigación está enfocada en el escenario de procesamiento de datos, donde los procesos son ejecutados de forma automatizada y programada, en alguna frecuencia de tiempo, para realizar mantenimiento de datos.

2.1.1 Orígenes

Si se tiene en cuenta la historia del despliegue de software, es posible afirmar que en un principio la única opción para correr aplicaciones era poseer servidores físicos. Con el tiempo surgió el concepto de la máquina virtual y los contenedores, que permitieron a los desarrolladores correr sus aplicaciones en la nube, pagando a un proveedor que ofrezca este tipo de servicios.

En la siguiente figura se muestra en resumen la historia de conceptos de informática que llevaron a la creación de *serverless*:

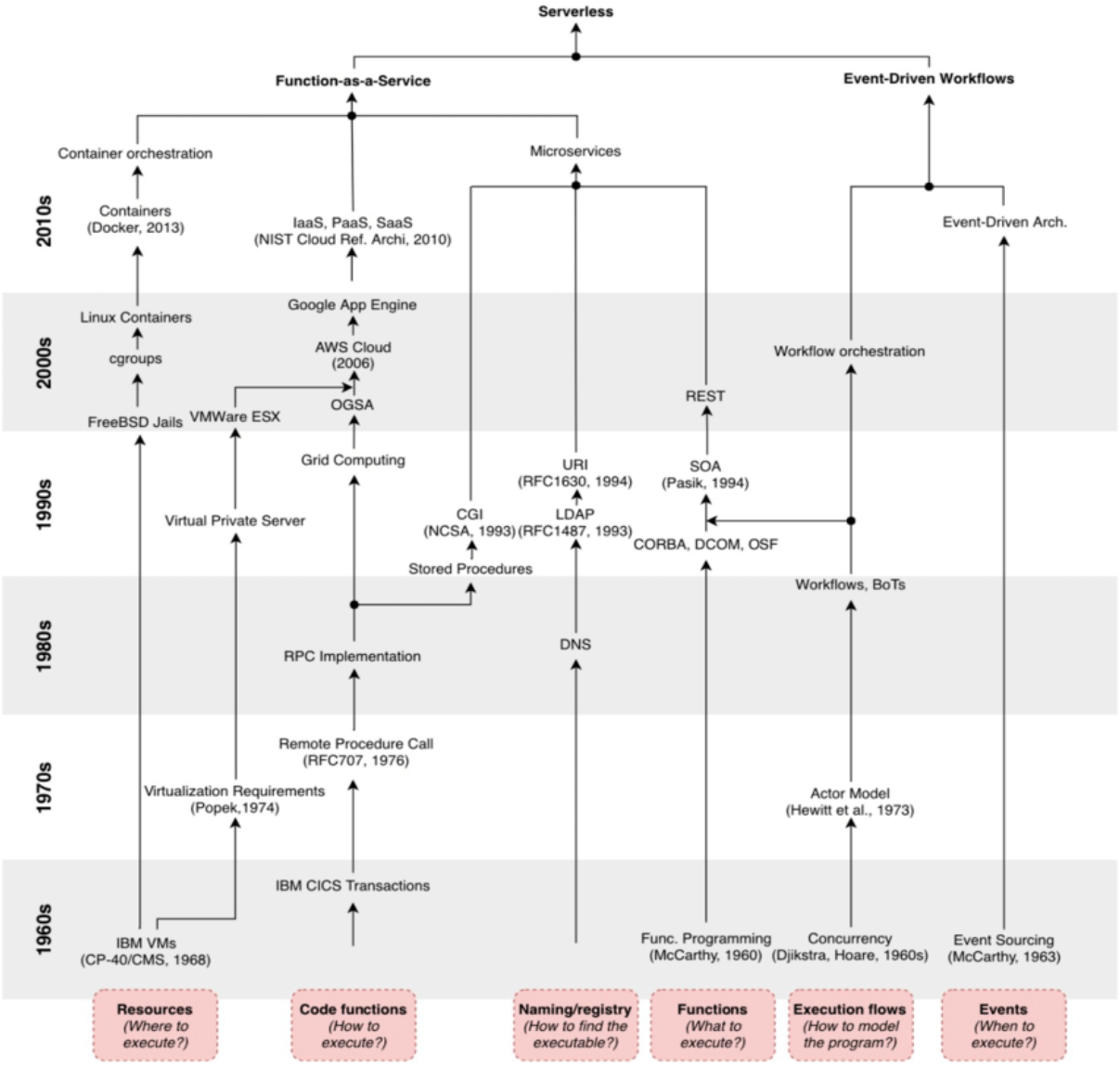

Figura 2.1.1-1: Historia de conceptos de computación que conllevan a serverless.

Fuente: https://www.researchgate.net/figure/A-history-of-computer-science-concepts-leading-to-serverless-computing_fig1_328088482 *el día 10-05-2021 a las 12:37 hs. [10]*

El concepto de *serverless* surgió en 2014 con la aparición de AWS Lambda, un servicio que provee FaaS para ejecutar lógica computacional sin necesidad de infraestructura subyacente. Con este servicio se obtenía una alternativa al problema mencionado anteriormente, eliminando por completo la necesidad de infraestructura. Por primera vez ya no era necesario tener servidores, ya sean físicos o virtuales, para correr aplicaciones.

En un artículo publicado por *451 Research*, en colaboración con *Red Hat*, separa la historia de *serverless* en dos etapas, *serverless* 1.0 y *serverless* 2.0. Segun *Red Hat*, *serverless* 1.0 se caracteriza por el uso de:

- HTTP
- Enfocado en Funciones
- Tiempo de ejecución limitado.
- Sin orquestación.
- Sin estado.

Con la aparición de containers en el mundo serverless, Red Hat considera que se avecina una nueva etapa de *serverless* 2.0, enfocado en:

- Manejo básico de estados.
- Uso de patrones de integración.
- Capacidad de mensajería avanzada.
- Fusión con el PaaS del proveedor.
- Estado e integración. [11]

De esta manera se puede evidenciar que en sus comienzos estaba orientado a la ejecución de funciones y tareas sin estado ni orquestación y con la aparición de servicios como AWS Step Functions se comenzaron a incluir estados en algunos servicios *serverless.*

Para finalizar, una cita textual del artículo previamente mencionado: “La adopción de *serverless* está creciendo rápidamente, y es un componente central de muchas transformaciones digitales en los proyectos de las empresas - algunos creen que toda la computación será *serverless* eventualmente” [11]

2.1.2 Function as a Service

En el artículo publicado por IEEE Internet Computing, titulado *Serverless is More: From PaaS to Present Cloud Computing*, se define Function as a Service (*FaaS*) como “un tipo de *serverless computing* donde los proveedores de servicios en la nube administran los recursos, el ciclo de vida, y el manejo de eventos que disparan la ejecución de funciones desarrolladas por usuarios”.[11]

Utilizando FaaS, los desarrolladores escriben funciones granulares sin estado, y los proveedores cloud son responsables de administrar todos los aspectos operacionales necesarios para su ejecución.[10]

En otras palabras, FaaS es un tipo de servicio en la nube que permite ejecutar código en respuesta a eventos sin necesidad de la infraestructura que está asociada típicamente a la construcción y despliegue de aplicaciones o microservicios. [12]

Para desplegar una aplicación en internet se requiere el aprovisionamiento y mantenimiento de servidores, ya sean físicos o virtuales, y el manejo de sistemas operativos y procesos de conectividad y transferencia de paquetes a través de redes. Utilizando FaaS, se delega la responsabilidad de esto a los proveedores de servicios *cloud*, liberando al desarrollador a enfocar su tiempo y productividad en la lógica computacional requerida y no en la implementación de la aplicación ni la administración de su infraestructura subyacente.

2.2 Servicios en la nube (*Cloud*)

Según el Instituto Nacional de Estandarización y Tecnología (*NIST* por sus siglas en inglés*)* define los servicios *cloud* como "un modelo para brindar de forma conveniente acceso *on-demand* a través de la red a recursos configurables compartidos que pueden rápidamente ser provisionados y desplegados con un mínimo esfuerzo administrativo" [13].

En otras palabras, es una forma de proveer recursos computacionales como servicio de utilidad. Este concepto fue introducido por primera vez en 1962 por John McCarthy, en su discurso dado en la celebración del cumplimiento de 100 años del MIT (*Massachusetts Institute of Technology)*, cuando sugirió que el tiempo compartido de recursos tecnológicos podría convertirse en el futuro en un modelo de negocios donde los recursos computacionales y aplicaciones puedan venderse como utilidades (como el agua o la electricidad) [14].

Las características esenciales que poseen son las siguientes [13] :

- *On-demand self-service:* un consumidor puede provisionar de forma unilateral los recursos computacionales que utiliza, de forma automática, sin la necesidad de interacción humana.
- *Broad network access:* los recursos son accesibles a través de la red desde cualquier punto geográfico.
- *Resource Pooling:* los recursos computacionales provistos son agrupados para servir a múltiples consumidores usando un modelo multi-accesible, donde los recursos físicos y virtuales son dinámicamente alocados y reasignados según la demanda de los distintos clientes.

- *Rapid elasticity:* los recursos pueden ser elásticamente provisionados y desplegados, en algunos casos de forma automatizada, para rápidamente escalar de acuerdo a la demanda.
- *Measured service:* los sistemas en la nube controlan y optimizan los recursos de forma automática utilizando métricas apropiadas para el nivel de abstracción de los distintos tipos de servicios.

Estos servicios están divididos en los siguientes 3 modelos de servicios [13] :

- *Software as a Service (SaaS):* la capacidad provista al consumidor es la usabilidad de las aplicaciones desarrolladas por el proveedor que corren en infraestructura *cloud.* Estas aplicaciones son accesibles desde varios dispositivos, ya sea a través de una interfaz como navegadores web, o bien una interfaz del programa. En este modelo, el consumidor no administra ni controla la infraestructura subyacente, que incluye servidores, conectividad, sistemas operativos, almacenamiento de datos, o incluso las capacidades individuales de la aplicación.
- *Platform as a Service (PaaS):* se provee al consumidor la capacidad de desplegar en la nube infraestructura creada por el consumidor o aplicaciones creadas utilizando herramientas del proveedor. El consumidor no administra ni controla la infraestructura subyacente.
- *Infrastructure as a Service (IaaS):* se provee al consumidor la capacidad de desplegar infraestructura, ya sea para almacenamiento de datos, conectividad, servidores, y cualquier tipo fundamental de recursos computacionales donde el consumidor sea capaz de instalar y ejecutar software de su elección, que puede incluir sistemas operativos y aplicaciones. El consumidor no administra ni controla la infraestructura *cloud* subyacente, pero sí tiene control sobre los sistemas operativos y las aplicaciones instaladas.

La siguiente figura, extraída del artículo publicado por C.N. Höfer · G. Karagiannis, titulado Cloud computing services: taxonomy and comparison, muestra estos modelos con ejemplos de servicios en la nube reales pertenecientes a cada uno de estos:

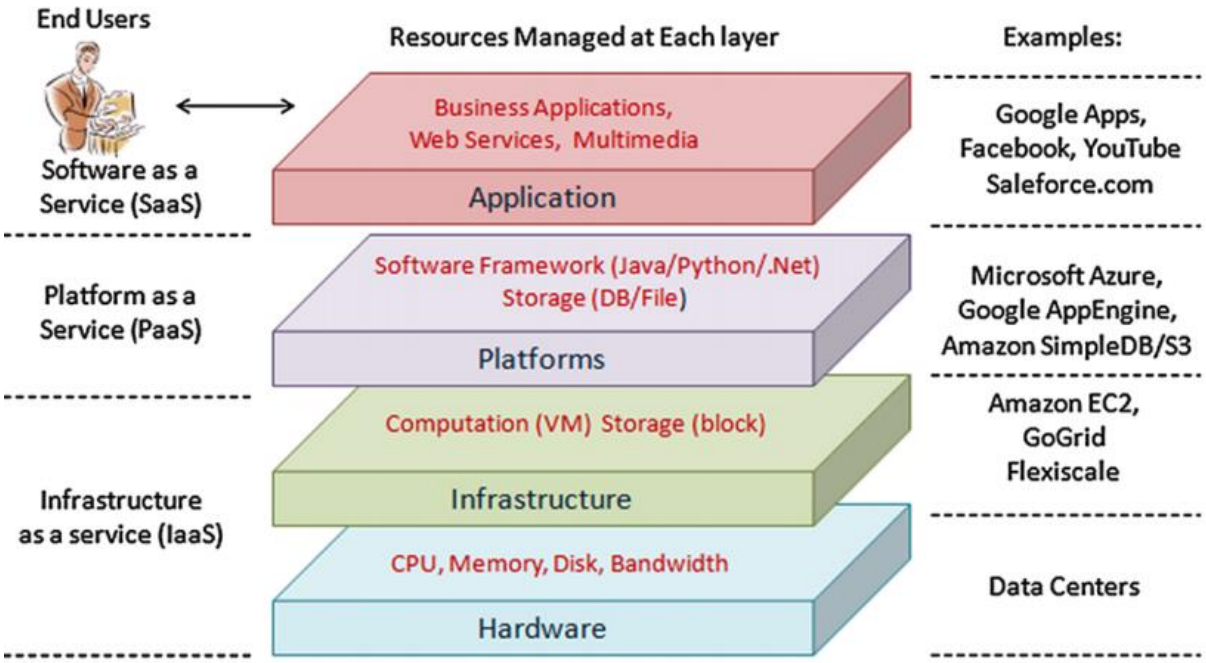

Figura 2.2-1: Modelos de servicios cloud.

Fuente: https://link.springer.com/content/pdf/10.1007/s13174-011-0027-x.pdf [15]

2.2.1 Funciones Lambda

En esta investigación se requirió desarrollar una implementación utilizando servicios en la nube. Se tomó la decisión de utilizar los servicios provistos por Amazon en su plataforma *AWS*.

De esta manera, para la implementación *serverless* fue necesario utilizar servicios de *FaaS* provistos por *AWS.* El servicio elegido para la implementación fue *AWS Lambda [16].*

"Lambda es un servicio de cómputo que permite ejecutar código sin necesidad de provisionar o administrar servidores" *[16].*

Este servicio se encarga de ejecutar código programado por el usuario en una arquitectura de alta disponibilidad y administrar los recursos computacionales necesarios para la correcta ejecución del código, incluyendo el mantenimiento de servidores y sistemas operativos, la escalabilidad automática, el monitoreo de la ejecución y *logging.*

Con *Lambda* se puede ejecutar virtualmente cualquier tipo de aplicación o servicios de *backend*, siempre y cuando el código provisto sea en un lenguaje soportado por el servicio. El código se organiza en funciones Lambda, que son ejecutadas únicamente cuando se necesita, y escalan automáticamente con la demanda o tráfico asociado.

Solo se paga por los recursos computacionales y el tiempo en el cual se los consume. Nunca se paga cuando el código no está siendo ejecutado. [16]

Este servicio es ideal para distintos escenarios, siempre y cuando el código que sea ejecutado cumpla con los límites provistos por el servicio dentro de su *standard runtime environment* [17]. Un límite restrictivo de este servicio es que cualquier función ejecutada no puede tener un tiempo de ejecución mayor a los 15 minutos. Luego de 15 minutos, *AWS* interrumpirá forzosamente la ejecución.

Algunos *features* de este servicio incluyen:

- Concurrencia y control de escalabilidad.
- Funciones definidas como imágenes de *containers.*
- Firma de código.
- Extensiones *Lambda.*
- Acceso de Bases de Datos.
- Acceso a *Filesystem.* [16]

2.2.2 Instancias de computo en la nube

Existen servicios *cloud* para desplegar instancias de cómputo en la nube. Estas funcionan como *Virtual Machines (VMs)*, o máquinas virtuales, que permiten a los usuarios utilizarlas como cualquier computadora sin necesidad de poseer hardware físico.

En el caso de *AWS,* esto es ofrecido dentro del servicio *EC2.* El mismo “provee capacidad de cómputo escalable en *AWS*”. Asimismo, elimina la necesidad de invertir en hardware por adelantado, brindando la posibilidad de desarrollar y desplegar aplicaciones rápidamente. [18]

Es posible utilizar *EC2* para crear cuantas máquinas virtuales sean necesarias, configurar su acceso y seguridad, su configuración de conectividad, y manejar su almacenamiento. También permite escalar automáticamente y manejar los cambios en el tráfico sin necesidad de estimar por adelantado la cantidad de requerimientos que tendrá que responder o clientes que servir.

Los principales *features* de este servicio son:

- Ambientes de cómputo virtuales, conocidos como instancias.
- *Templates* preconfigurados para las instancias, conocidas como imágenes (*AMIs).*
- Configuraciones varias de CPU, memoria, almacenamiento y capacidad de conectividad.
- Login securizado.
- Volúmenes temporales y persistentes de almacenamiento.

- Múltiples ubicaciones físicas para los recursos, conocidas como regiones y zonas de disponibilidad.
- Firewalls.
- IPv4 estática.
- Metadata.
- Redes virtuales privadas, conocidas como *Virtual Private Clouds (VPCs).* [18]

2.2.3 Almacenamiento

Asimismo, existen servicios para manejar el almacenamiento de datos. Uno de los provistos por *AWS* para esta tarea es *Amazon Simple Storage Service*, también conocido como S3, que está pensado como almacenamiento para internet, diseñado para que la escalabilidad de la web sea más sencilla para los desarrolladores [19].

Este servicio permite crear un *filesystem* compartido, denominado *bucket*, en donde se pueden crear carpetas y realizar carga y descarga de archivos con accesos compartidos.

Las ventajas de utilizarlo son:

- La posibilidad de crear *buckets,* que funcionan como contenedores fundamentales para almacenamiento de datos.
- Almacenamiento de datos, donde la cantidad de archivos dentro del contenedor no está limitada. Esto quiere decir que se pueden cargar todos los archivos que se desee. Sin embargo, la cantidad total de datos almacenados tendrá un impacto directo en el costo del servicio. Asimismo, ningún archivo puede tener un tamaño mayor a 5 TB de datos.
- Descarga de archivos, permitiendo que cualquier usuario con acceso al *bucket* pueda descargar cualquier archivo.
- Manejo de permisos, donde se puede otorgar o denegar acceso a la información almacenada a los distintos usuarios. También se puede otorgar permisos de carga o descarga a los usuarios, asegurando que usuarios sin permisos no puedan sobrescribir el archivo creando una nueva versión, pero que sí tengan acceso a descargarlo cuantas veces quieran [19].

2.3 Automatización de procesos

“Un proceso define una descripción y un orden de actividades en un tiempo y espacio que están diseñados para brindar productos o servicios específicos y a su vez asegurar que se cumplan los objetivos de la organización”. Es la base conceptual para la integración y coordinación de tareas automatizadas. Es por esto que la administración efectiva de los procesos es crítica para el desarrollo de las organizaciones. [20]

Los procesos se ejecutan en flujos de trabajo, o *workflows*, que especifican cómo está coordinada la ejecución de múltiples tareas llevadas a cabo por diferentes entidades dentro de la organización. Una tarea "define el trabajo que debe ser realizado y puede ser especificada de muchas maneras, incluyendo programas de computadoras" [20]. Las tareas ejecutadas dentro de un flujo de trabajo suelen ser ejecutadas por un sistema automatizado.

Las tareas recurrentes son aquellas que son ejecutadas periódicamente dentro de una frecuencia preestablecida. Por ejemplo, un programa es ejecutado una vez al día actualizando archivos de logs en un *filesystem* compartido. En otras palabras, son tareas repetitivas ejecutadas en flujos de trabajo cada determinado tiempo.

En *Red Hat* definen la automatización como "el uso de software para crear instrucciones repetibles y procesos que reemplazan o reduzcan la interacción humana con los sistemas de tecnología" [21].

Teniendo en cuenta estas definiciones y conceptos, podemos pensar en la automatización de procesos recurrentes como la ejecución de tareas cada determinada frecuencia de tiempo sin necesidad de intervención humana.

2.3.1 ETL (*Extract Transform Load*)

Uno de los casos más comunes de procesos automatizados son los procesos ETL (*Extract Transform Load*). Estos se pueden definir como "software responsable de la extracción de datos de múltiples fuentes, su limpieza, transformación, e inserción en un *data warehouse*" [22]. Es decir, son procesos encargados de extraer información, modificarla de acuerdo a las necesidades, y luego cargarla de forma permanente en una fuente de datos final. Estas fuentes suelen ser bases de datos, aunque también pueden ser archivos y otras estructuras de datos *NoSQL*, como por ejemplo, un *Key Value Store*.

Estos procesos suelen ser utilizados para generar reportes y gráficos visualizables para mejorar la toma de decisiones de las empresas. Son críticos en áreas como *Business Intelligence*, y pueden ser utilizados para cargar fuentes de datos explotadas en procesos de *Machine Learning.*

Debido a que el análisis de datos ha tomado un rol de suma importancia en la toma de decisiones de las empresas y en múltiples procesos, es cada vez más frecuente el uso de procesos ETL para asegurar la calidad e integridad de los datos utilizados. Por estas razones, la necesidad de las compañías de automatizar dichos procesos ha visto un incremento en los últimos años.

A esto se le suma la creciente utilización de *Big Data*, que se basa en analizar y explotar múltiples datos sobre las operaciones de las compañías. Dado que esta tecnología requiere de datos para mejorar la toma de decisiones de las empresas, resulta imprescindible utilizar procesos ETL para transformar la data utilizada.

2.4 Metodologías y *framework* para la evaluación de productos de software

MyFEPS es un proyecto que brinda métricas para la evaluación de software [3]. En este trabajo se estudia en específico la aplicación del punto 3.7 de MyFEPS titulado Eficiencia, en específico la eficiencia en la utilización de memoria interna, sección 3.7.3, según se muestra en la Tabla 4.1.3-1.

Código MyFEPS [3]	**Atributo**	**Métrica**
07.3.1.U.	Memoria Interna Utilizada para ejecutar la Función Tipo X	Con una Aplicación Adhoc, Para todas las Funciones 1. Medir CMF(x) cantidad de Memoria usada por la Función en Carga Alta. 2. Obtener una Cantidad de Memoria del sistema: CMS. 3. Valoración = 1- (CMF/CMS) Valor Final= Media de Valoración Parcial.

Tabla 4.1.3-1: Métrica: Eficiencia en la utilización de memoria interna, extraída del proyecto MyFEPS.

- Código MyFEPS: número identificador de la métrica dentro del proyecto.
- Atributo: objeto a ser evaluado.
- Métrica: fórmula brindada por MyFEPS para evaluar el atributo.

Estado del arte

Existen múltiples casos de uso de la tecnología *serverless*. Si bien en esta investigación solo se diseña e implementa una solución de tareas automatizadas recurrentes para casos de ETLs y mantenimientos de logs, es importante conocer las distintas aplicaciones de *serverless* para entender más en detalle las posibilidades que esta tecnología ofrece, considerando que estos casos de uso son sugeridos por el proveedor para motivar a potenciales clientes a migrar sus sistemas hacia estos servicios.

3.1 Aplicaciones web

Se pueden implementar aplicaciones web que consistan de RESTful APIs utilizando *serverless*. Pese a que existen diferentes formas de lograr esto, Amazon ofrece una documentación con figuras que muestran una arquitectura básica utilizando servicios *serverless* de los cuales son propietarios.

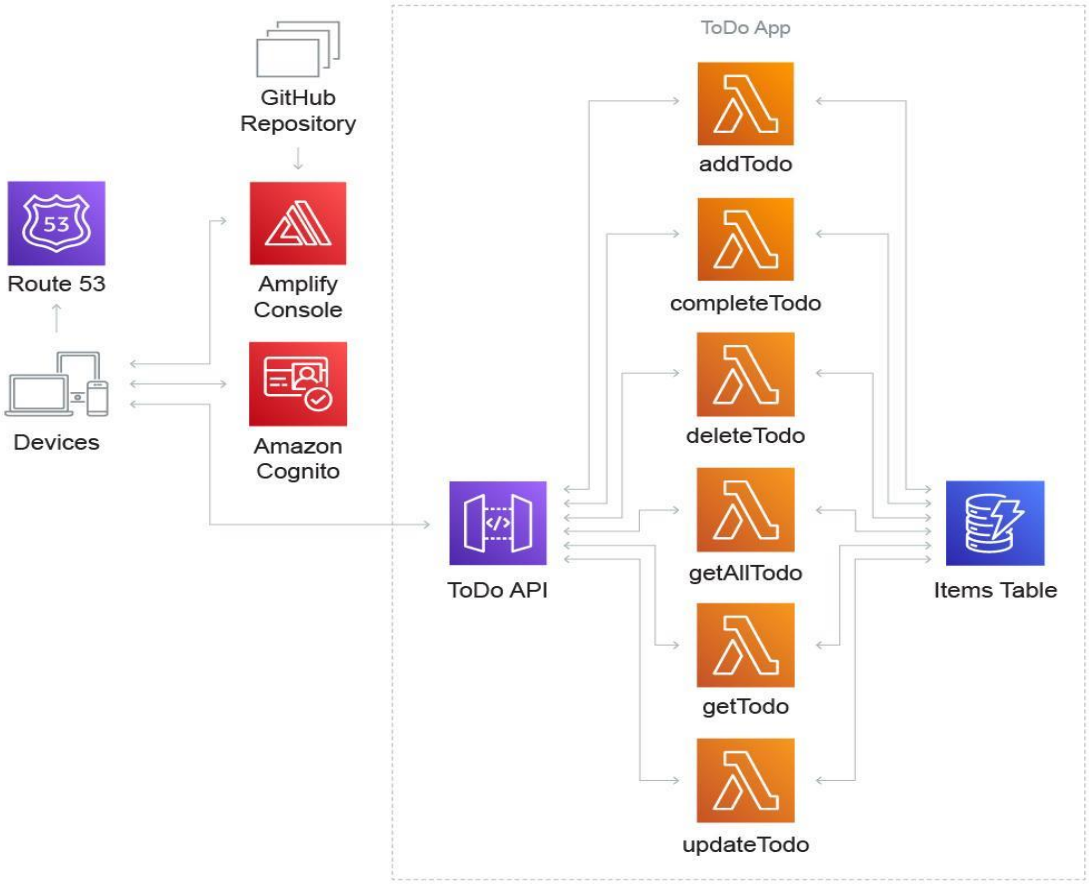

Figura 3.1-1: implementación de una API de items de un producto de software.

Fuente: https://aws.amazon.com/serverless/?nc2=h_ql_sol_use_servc visitada el 18-04-2021 a las 20:30 hs. [23]

En esta figura es posible observar cómo los clientes se conectan a la aplicación *ToDo* mediante una tabla de ruteo, y este evento es manejado por ToDo API para ejecutar las funciones lambda programadas por el desarrollador de la aplicación.

Esta arquitectura reemplaza los servidores físicos utilizando FaaS para funcionar de *backend*, siendo el proveedor *cloud* quien tiene la responsabilidad de manejar la ejecución de las funciones en base a los eventos configurados por el usuario.

Debido a la forma de facturación que tiene lambda, es posible que en algunos casos esta solución no aplique a la aplicación, ya sea por el tráfico que recibe o por la cantidad de datos que manejara. Cada caso debe ser analizado por separado, considerando todos los valores de entrada y salida que se estiman.

3.2 Procesamiento de datos

AWS ofrece un diagrama con una arquitectura para un modelo simple de procesos automatizados que se ejecuta frente a la creación de archivos en un *bucket* de S3, enviando notificaciones por email, y modificando el archivo *markdown* recibido como entrada para cargarlo en otro *bucket* de S3. También se ejecuta una *Lambda Function* para analizar el tono del archivo cargado y cargar dicho análisis en un DynamoDB.

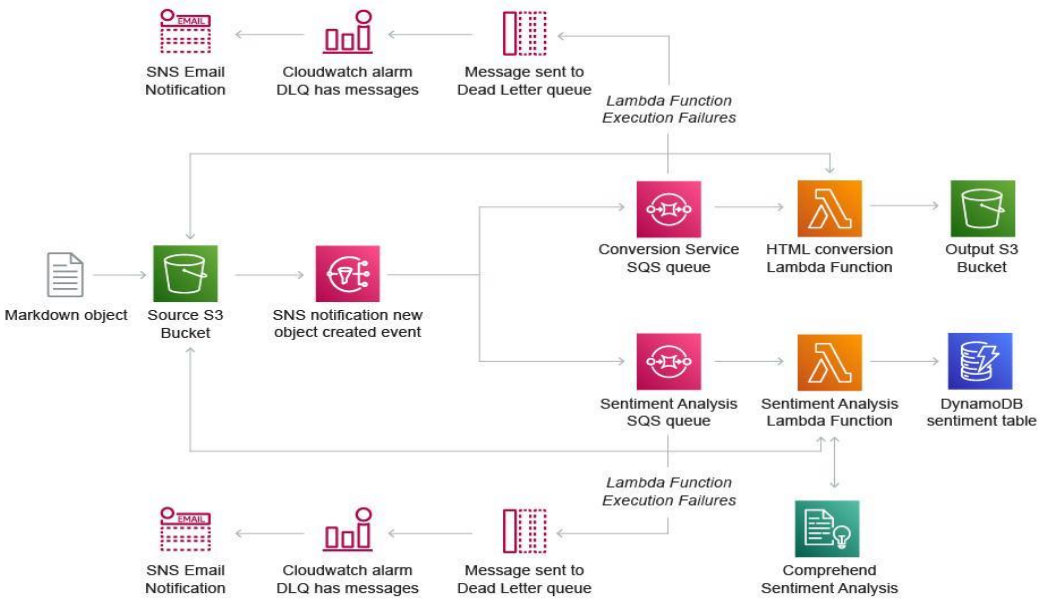

Figura 3.2-1: arquitectura de proceso automatizado que incluye notificación por email ante la creación de un nuevo objeto.

Fuente: https://aws.amazon.com/serverless/?nc2=h_ql_sol_use_servc visitada el 18-04-2021 a las 20:30 hs. [23]

Este es un caso de uso para los múltiples escenarios que existen de tareas automatizadas recurrentes. Si bien este diagrama ejecuta un ETL, incluye la automatización de notificaciones por email que no es propio de un ETL (consultar la sección 2.3.1 ETL).

Esta investigación se enfocó en comparar el funcionamiento de una solución *serverless* en comparación a una arquitectura tradicional para tareas automatizadas recurrentes sin la complejidad de notificaciones, manejo de permisos, u otro tipo de agregados a un proceso ETL.

Por ende, se extrae de esta arquitectura la ejecución de Funciones Lambda para realizar la transformación de los datos, que luego son cargados en un *bucket* de S3, al igual que en el caso implementado en este trabajo.

3.3 Procesamiento ETL

Este es el escenario analizado en este trabajo, donde se requiere ejecutar un programa con el fin de procesar datos. Ya sea para realizar mantenimiento, limpieza, o bien explotación de datos, se suelen requerir tareas o procesos que se ejecuten frecuentemente.

Si bien la arquitectura de estas tareas puede variar dependiendo del caso de uso, la idea siempre es similar. Se ejecuta un proceso de forma automática para realizar una serie de tareas que tienen como fin la carga de datos limpios en una tabla u otra estructura de permanencia de información.

La siguiente figura muestra una arquitectura de procesos de ETL utilizando en su totalidad servicios *serverless* ofrecidos por AWS.

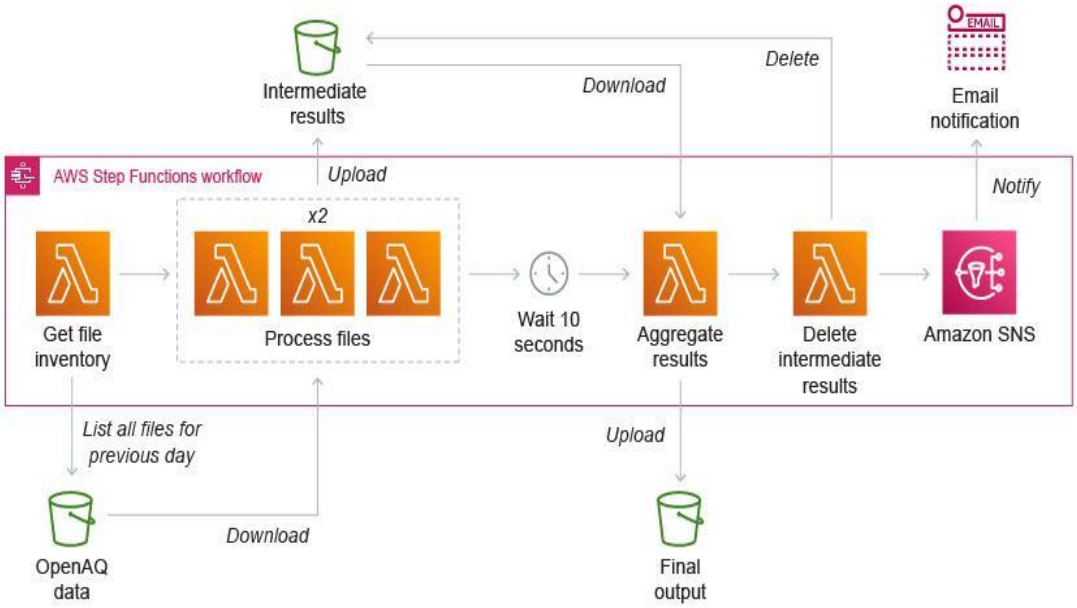

Figura 3.3-1: implementación de ETL utilizando Lambda Functions de AWS.

Fuente: https://aws.amazon.com/serverless/?nc2=h_ql_sol_use_servc visitada el 18-04-2021 a las 20:30 hs.[23]

Esta arquitectura implementa el servicio de AWS llamado *Step Functions*, que permite ejecutar una serie de funciones Lambda siguiendo cierta lógica previamente configurada. En esta investigación se utilizará un modelo similar, conteniendo una menor cantidad de funciones Lambda y sin utilizar Amazon SNS, servicio para enviar notificaciones disparadas por eventos (en el caso de la figura 3.3-1, cuando la función *Delete intermediate results* finaliza su ejecución).

Esta arquitectura es el principal modelo teórico de este trabajo, en el cual está basado el prototipo *serverless* de esta investigación. Del mismo modo, se utilizó *Lambda Functions* junto con S3 para implementar un ETL.

También es posible ejecutar un proceso ETL usando tablas de bases de datos transaccionales como fuentes para la extracción y carga. En este caso será más sencillo unir registros de distintas tablas utilizando un lenguaje SQL. Asimismo es posible ejecutar un proceso ETL utilizando bases de datos NoSQL, como MongoDB, como fuente de extracción y carga de datos.

Por ende, esta arquitectura permite realizar cualquier tipo de proceso ETL sin necesidad de servidores físicos o digitales, independientemente de la fuente de datos y su implementación técnica.

3.4 Aplicación FaaS para Machine learning

En última instancia, se tiene el caso de uso de implementación de *serverless* para procesos relacionados con la inteligencia artificial y el *Machine Learning.* Si bien esta implementación no es utilizada en esta investigación, resulta interesante estudiar cómo la implementación de *serverless* no está limitada a aplicaciones web y procesos o tareas automatizadas que requieren de servidores o infraestructura, sino que también puede ser usada en el campo de la inteligencia artificial.

La siguiente figura, extraída del sitio oficial de AWS, describe una arquitectura *serverless* que utiliza servicios de *Machine Learning,* como *Comprehend* y *Rekognition*, para indexar documentos e imágenes y enviar los resultados a un *ElasticSearch*. [23]

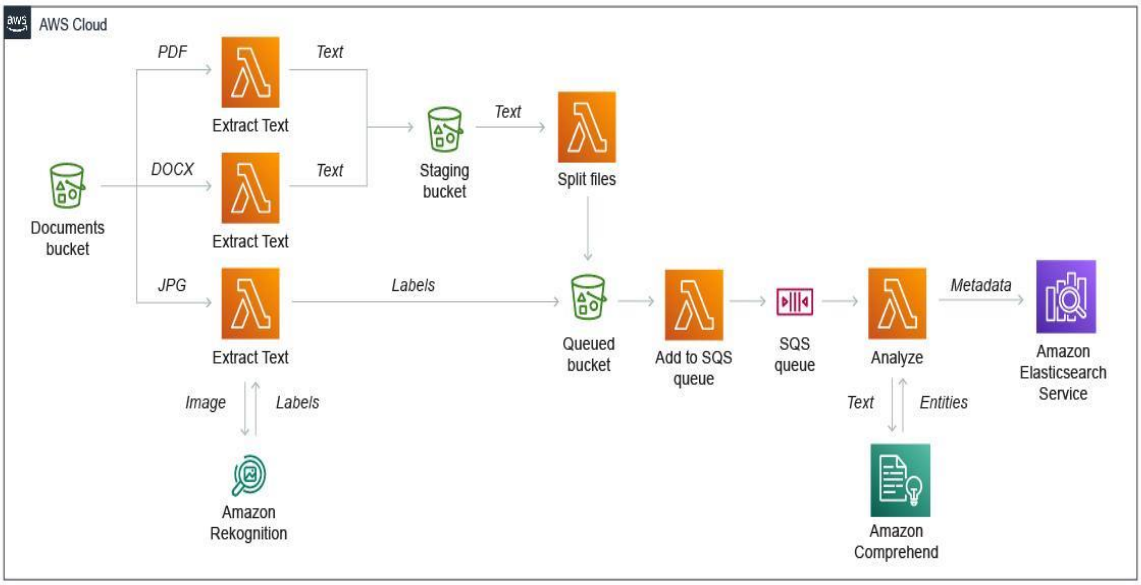

Figura 3.4-1: Implementación de una API de items de un producto de software.

Fuente: https://aws.amazon.com/serverless/?nc2=h_ql_sol_use_servc visitada el 18-04-2021 a las 20:30 hs. [23]

Esto permite que sea posible buscar distintas métricas y resultados sobre los datos obtenidos sin necesidad de utilizar servidores o infraestructura subyacente de ningún tipo.

En esta implementación *serverless*, la responsabilidad de manejar la infraestructura, de administrar los eventos, y de encargarse de todos los costos operativos es del proveedor *cloud* (en este caso, AWS).

Esto permite que los desarrolladores de *Machine Learning* puedan enfocarse en su algoritmo y modelo, y no en cómo será la implementación de la arquitectura y la aplicación en su totalidad.

Desarrollo

En esta sección se describen dos prototipos a implementar y el análisis realizado, así como los resultados obtenidos de la ejecución de los prototipos y las métricas de utilización de recursos computacionales recolectadas.

Los prototipos hacen alusión a una tarea automatizada recurrente, que fue implementada tanto dentro de un servidor en la nube como en una función Lambda, obteniendo una solución *serverless*. Ambos prototipos son comparados, verificando las métricas de utilización de memoria y tiempos de ejecución resultantes. Esta comparación fue realizada utilizando tablas para evaluar las métricas recolectadas, cuyos resultados son expuestos al final de este trabajo.

4.1 Diseño de los prototipos

El sistema a implementar en este trabajo es un prototipo de ETL que extrae datos de un archivo JSON almacenado en S3, realiza una transformación lógica de los datos y luego carga el resultado en un nuevo archivo. Dicho ETL simula un proceso de transformación de datos que podría ser utilizado para mandar anuncios específicos de forma automatizada dependiendo del tipo de consumidor. Este ETL se ejecutará diariamente como una tarea automatizada recurrente. Se analizará la memoria RAM utilizada por el prototipo, así como el tiempo total de ejecución.

Se asume que la información extraída fue generada por algún proceso de la compañía, como por ejemplo un modelo de machine learning con un algoritmo de clasificación que pone etiquetas a los distintos usuarios para luego saber qué tipo de consumidor son. El caso del Machine learning es un gran ejemplo ya que requiere de múltiples procesos que constantemente transformen la información y la almacenen en nuevas fuentes de datos.

En base a este escenario se desarrollará una tarea automatizada recurrente a modo de experimento para demostrar que puede ser implementado tanto en una arquitectura tradicional cómo utilizando servicios *serverless*.

El experimento consiste en implementar la misma tarea automatizada recurrente dentro de un esquema *serverless* utilizando funciones Lambda, así como dentro de un esquema tradicional en la nube con infraestructura subyacente asociada. Ambas utilizan un *bucket* de S3 para almacenar los archivos generados por la ejecución de los prototipos.

La idea de este experimento es que ambas transformaciones obtenidas de la ejecución de los prototipos sean idénticas. Asimismo, comparar ambas soluciones y corroborar que las métricas operativas obtenidas en el prototipo *serverless* son mejores en comparación a la arquitectura tradicional. Por este motivo se desarrollarán tablas comparativas en base a este experimento. Se comparan también similitudes o diferencias de los archivos finales cargados en el almacenamiento de datos.

El sistema a desarrollar actuará como ETL, extrayendo información de un archivo guardado en la nube, modificando los datos en base a cierta lógica computacional, y cargando un nuevo archivo final en un directorio de S3. Este sistema se ejecuta tanto desde un servidor como desde una *lambda function* obteniendo el mismo resultado almacenado en diferentes carpetas con el fin de comparar las soluciones.

Ambas implementaciones se ejecutaran diariamente a la medianoche. En el caso de la tradicional, se ejecuta una tarea en *scheduler* que administra el sistema operativo del servidor. En cuanto a la solución *serverless*, se ejecutará un evento disparado por un servicio de AWS llamado *event bridge*, cuyo funcionamiento es similar al del *cron table* de un sistema operativo Linux.

El servidor es una máquina virtual corriendo Linux de AWS, un sistema operativo provisto por Amazon que se encuentra dentro de la capa gratuita. El mismo será configurado para aceptar tráfico proveniente de ssh para que los administradores puedan conectarse a la instancia siempre y cuando tengan la clave de acceso, encriptada dentro de un *.pem*. Dichas configuraciones serán realizadas de forma manual por los administradores, así como la configuración del *cron table*, utilizando comandos a través de la terminal del sistema operativo Linux.

Se creará una *lambda function* de forma manual, ejecutando el mismo algoritmo que el servidor. El evento disparador de la función será creado de forma manual utilizando la consola de AWS. Ambos servicios componen la arquitectura serverless.

Para almacenar los datos ambas soluciones utilizan un *bucket* de S3 que cuenta con tres directorios. El primero, llamado *Data,* contiene el archivo que es extraído por ambas soluciones, también conocido como fuente de datos.

La segunda carpeta, llamada *serverless*, contiene el archivo final generado por la ejecución de la función Lambda, que se encuentra versionado por su fecha de ejecución, utilizando la convencion de nombres *final_data_mm_dd_yyyyZhh:mm:ssT.json*.

La última carpeta, llamada *traditional*, contiene el archivo final generado por la ejecución de la tarea en *cron table* dentro del servidor, utilizando la misma convención de nombres y versionado de archivos que utiliza la solución *serverless*. Incluyendo la fecha y hora de creación del archivo se puede mantener históricos en caso de que requieran ser utilizados en otros procesos o auditorías que exceden los límites de esta investigación.

4.1.1 Prototipo tradicional

El prototipo tradicional cuenta con un servidor con sistema operativo Linux, cuya tarea es correr en *scheduler* un *Script* desarrollado en *python* que realiza un proceso de ETL, descargando de S3 un archivo conteniendo los datos de consumidores, transformando esos datos según una lógica computacional programada, y cargando un nuevo archivo en el mismo *bucket* de S3.

Este prototipo simula una tarea automatizada recurrente para procesos ETL dentro de un esquema tradicional con infraestructura asociada. La misma deberá ser mantenida y monitoreada para asegurar el correcto funcionamiento del proceso ETL en caso de ser implementado en un ambiente productivo de una empresa real.

El script es ejecutado por el *cron* del sistema operativo Linux instalado en el servidor. En la siguiente figura se puede ver cómo interactúa el servidor con el almacén de datos dentro de una arquitectura tradicional en la nube, utilizando los servicios de AWS:

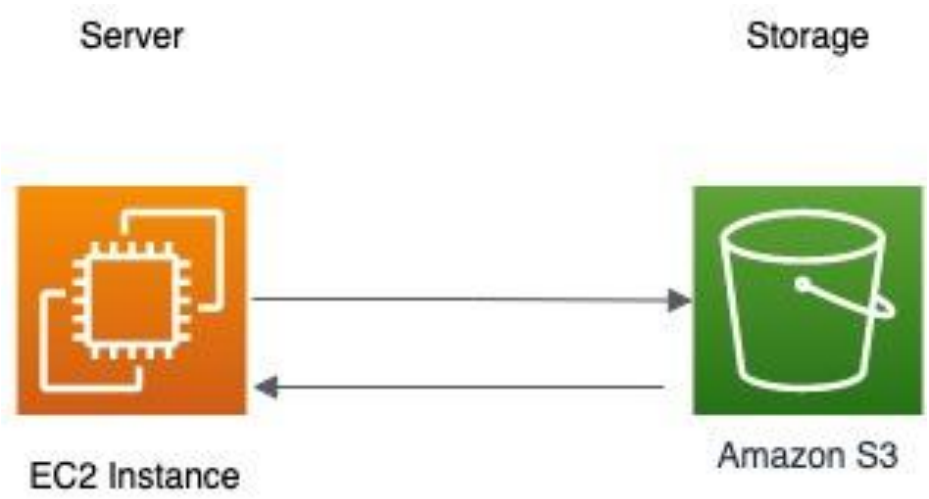

Figura 4.1.1-1: arquitectura tradicional para proceso automatizado ETL.

El servidor será configurado para aceptar todo el tráfico entrante del internet que contenga la llave encriptada de acceso, ubicada dentro del archivo *.pem*, que no puede ser replicada ni alterada. Esto permitirá a los administradores acceder al servidor a realizar la configuración del *cron table* y de los directorios conteniendo el código que ejecuta el proceso ETL. A su vez, el *hardware* del servidor se compone por un CPU virtual de 2.5 GHz y 1GB de memoria RAM. Esto se debe a que el tipo de instancia utilizada es una *t2.micro,* que cuenta con dicho *hardware.* Este tipo de instancia es elegida debido a que pertenece a la capa gratuita de AWS y puede ser utilizada sin costo para los efectos de esta investigación.

Este prototipo sigue un esquema IaaS, *Infrastructure as a Service*, donde se proporciona infraestructura como servicio computacional para poder correr los programas dentro de instancias físicas o virtuales.

4.1.2 Prototipo serverless

El prototipo serverless se compone de una *Lambda function* que ejecuta un script para realizar un proceso ETL similar al de la implementación tradicional. La diferencia es que este prototipo no tiene infraestructura asociada. Es decir, no existe el servidor utilizado en el esquema tradicional que cuenta con un *cron table* para ejecutar la tarea de forma recurrente. Este esquema requiere de un evento disparador encargado de ejecutar la función lambda para replicar la funcionalidad del *scheduler* del servidor. Esta arquitectura también cuenta con un servicio latente de las funciones lambdas para manejar el *logging* de las ejecuciones de la función, eliminando la necesidad de configurar una herramienta externa o crear una instancia de monitoreo y *logs*. (Ver figura 4.1.2-1)

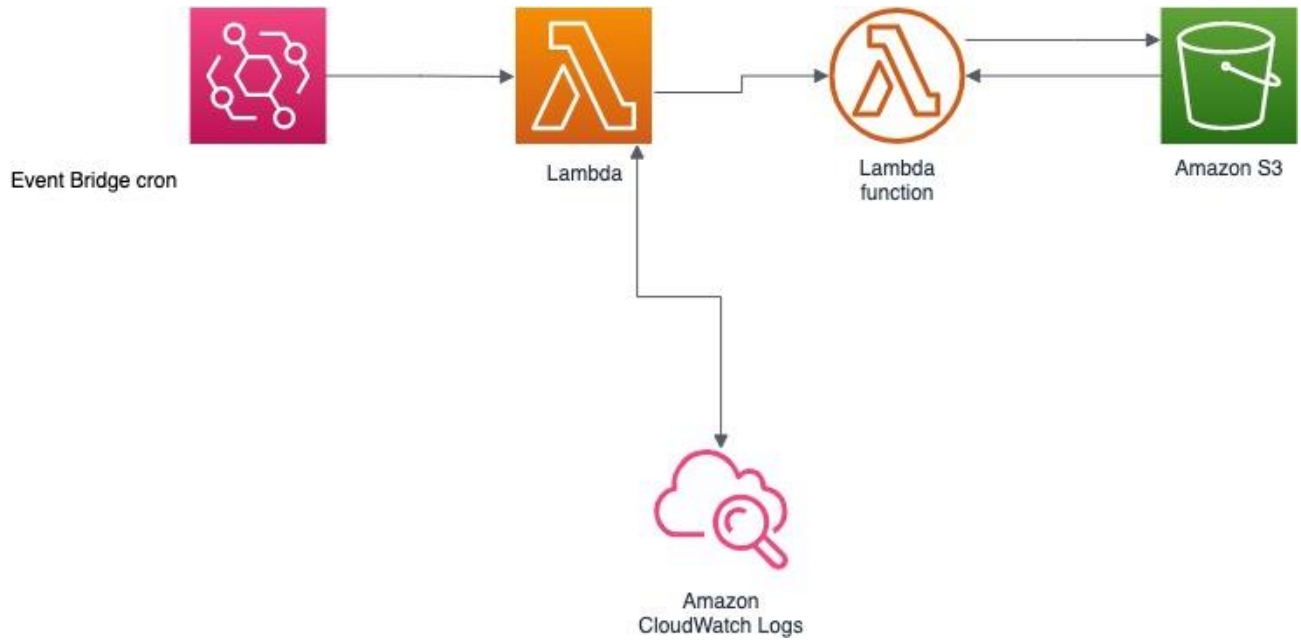

Figura 4.1.2-1: arquitectura serverless para proceso automatizado ETL.

Este prototipo sigue un esquema FaaS, *Function as a Service*, donde no existe infraestructura subyacente y el código es ejecutado por *triggers* externos, también conocidos como eventos, eliminando la necesidad de mantener infraestructura asociada.

4.1.3 Comparación de métricas operativas

Para determinar si la utilización de recursos computacionales del prototipo *serverless* es mejor que la del prototipo tradicional, es necesario recolectar métricas operativas, como la utilización de memoria y tiempo de ejecución, y producir una tabla comparativa.

Como se observa en la figura 4.1.2-1 , en la solución *serverless* es necesario incluir de forma independiente al evento disparador debido a la falta de un servidor con *scheduler*. En comparación la solución tradicional solo cuenta con un servidor en la nube (ver figura 4.1.1-1).

La ventaja técnica de utilizar *serverless* es la posibilidad de desacoplar en pequeños módulos o subrutinas la funcionalidad particular de cada proceso, reduciendo arquitecturas muy complejas en pequeños módulos, facilitando el mantenimiento de cada uno de ellos en el futuro.

Por otro lado, el diagrama de la arquitectura tradicional es más simple en comparación al de la implementación serverless, ya que el servidor será responsable de ejecutar la tarea programada dentro del *scheduler*. Esto reduce la modularidad que se obtiene utilizando *serverless*, siendo el servidor el único punto de falla dentro de la arquitectura. El problema de mantener esta arquitectura es que frente a una falla del servidor no se sabrá exactamente cuál módulo o subrutina del programa es el que ha fallado, siendo la única alternativa entrar al servidor y hacer un *troubleshooting* del problema.

Ambas soluciones deben obtener el mismo resultado, siendo el único cambio la arquitectura, y por consecuente, la infraestructura subyacente. Estas implementaciones evidencian que el mismo problema puede ser resuelto desde un esquema con servidores en la nube tanto como un esquema *serverless*.

Resta entonces comparar el funcionamiento de ambos prototipos y verificar las métricas operativas, comparándolas para elegir el CPU, tiempo de ejecución y tamaño de archivo resultante para evaluar cual es el prototipo más eficiente. Dicha comparativa será realizada sobre tablas para visualizar rápidamente los números resultantes como evidencia empírica.

Tomando como parámetros las métricas de utilización de memoria recolectadas de la ejecución de los prototipos implementados, se utilizó MyFEPS para medir la eficiencia en la utilización de memoria de ambos prototipos (ver 2.4 Metodologías y framework para la evaluación de productos de software (MyFEPS))

4.2 Implantación de los prototipos

Los prototipos descritos anteriormente fueron implementados utilizando los servicios en la nube de Amazon dentro de la herramienta AWS. Ambas soluciones se desarrollaron e implementaron dentro del marco del diseño del sistema. La implantación fue exitosa dado que ambos sistemas funcionaron de forma correcta dentro de los parámetros esperados, como un tiempo total de ejecución menor a los 2 segundos y una utilización de memoria y CPU dentro de los límites provistos por el servicio en la nube utilizado. Gracias a esto se pudo analizar las métricas resultantes de ambas implementaciones, que permiten compararlas con el fin de demostrar la hipótesis. Debido a que el producto es el mismo en ambas arquitecturas es posible extrapolar los hallazgos del resultado de la implementación a otros escenarios que posean arquitecturas de software similares para escenarios de tareas automatizadas recurrentes.

Desde el punto de vista técnico, el prototipo implementado simula un proceso ETL cuya tarea es extraer información de usuarios almacenada en un *bucket* de S3, transformarla según una lógica computacional y cargarla en un *bucket* de S3, de la manera que fue descrito en la sección 4.1 Diseño de los prototipos

Este proceso ETL es ejecutado desde dos prototipos diferentes, uno desarrollado en un esquema tradicional en la nube y otro desarrollado en un esquema *serverless.* Ambos prototipos extraen los datos desde la misma carpeta dentro del *bucket* de S3, llamada *data/.* En la figura 4.2-1 se pueden ver las carpetas creadas con los nombres descritos en el diseño del sistema.

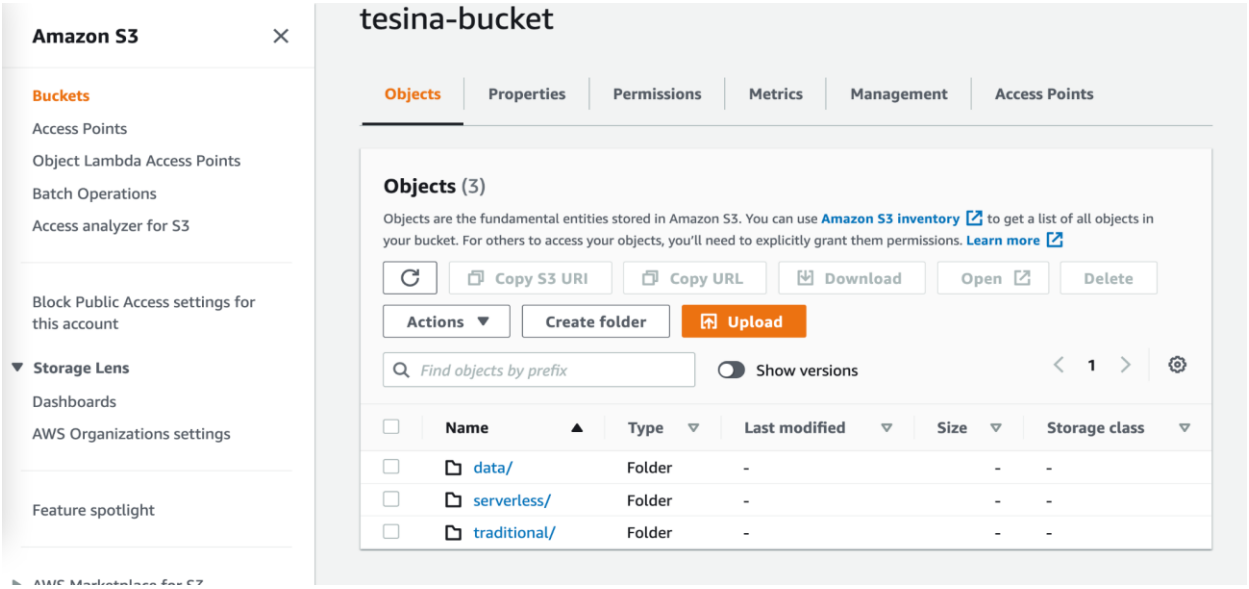

Figura 4.2-1: bucket de S3 llamado tesina-bucket utilizado para almacenar archivos relacionados al proceso ETL.

Ambos prototipos se encuentran funcionales y no presentan ningún problema o error en el código o la ejecución de la tarea. Los archivos obtenidos resultantes de la ejecución de estos sistemas son los esperados, demostrando que el sistema funciona y que sólo resta comparar ambas implementaciones para determinar si la hipótesis es correcta.

4.2.1 Prototipo Tradicional

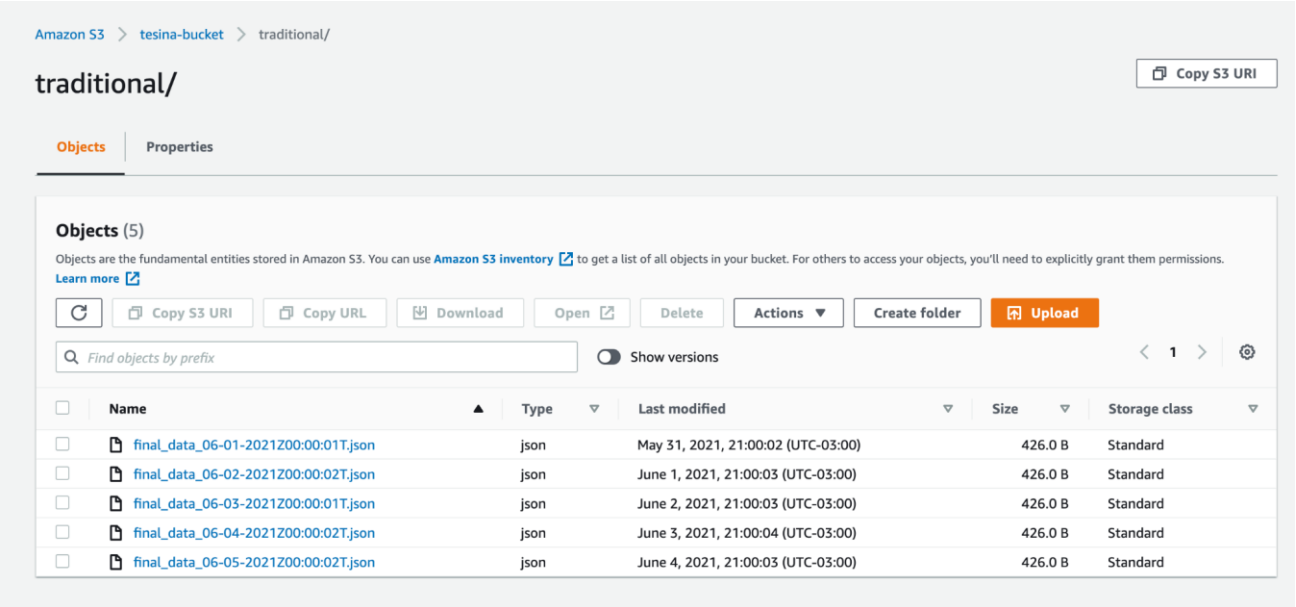

Figura 4.2.1-1: directorio traditional/ dentro de tesina-bucket contiene todos los archivos resultantes de la ejecución del ETL tradicional.

Cómo se puede ver en la figura 4.2.1-1, la implementación del ETL en el esquema tradicional logró crear exitosamente un archivo diariamente durante las últimas semanas. Esto demuestra que puede ser ejecutado dentro de un esquema tradicional resultando todas sus ejecuciones en el mismo archivo con el mismo tamaño, y por ende la misma información.

La implementación del sistema fue realizada de acuerdo a las arquitecturas diseñadas, obteniendo un *script* desarrollado en *Python* 3.8 (ver 9.1 Script desarrollado en Python 3.8 para ETL en esquema tradicional) que es ejecutado diariamente a la medianoche por un proceso en *scheduler* (ver 9.2 Script desarrollado en Bash para ejecutar ETL en esquema tradicional y 9.4 Configuración del cron table en el servidor). Dicho *script* se encarga de ejecutar el ETL, que se compone por 3 funciones ejecutadas secuencialmente.

La primera función, llamada *extract*, extrae el archivo con la fuente de datos de la carpeta *data/* ubicada en el bucket de S3. Esto es posible gracias a la librería *boto3*,

que *Python* brinda para utilizar la funcionalidad del AWS CLI encapsulado dentro de la misma.

La segunda función, llamada *transform*, se encarga de ejecutar una serie de secuencias lógicas para computar los datos extraídos por la función anterior y transformar la información en base a ciertos parámetros de entrada. Específicamente, se realizan iteraciones sobre la fuente de datos, que tiene un formato .json, y por cada iteración, se evalúa el tipo de consumidor y en base a eso se genera un objeto consumidor transformado, que permite determinar que tipo de anuncios deben enviarse al usuario de forma automatizada.

Por último, la función *load,* es responsable de cargar el archivo conteniendo los datos transformados. Al igual que la función *extract*, utiliza *boto3* para hacer uso del AWS CLI encapsulado por la librería.

4.2.2 Prototipo Serverless

El prototipo implementado en esquema *serverless* realiza exactamente las mismas tareas que la implementación dentro del esquema tradicional en la nube. Como se puede observar en la figura 4.2.2-1, los archivos creados por el *lambda function* contienen los mismos datos transformados, evidenciado en el tamaño total del archivo de 426.0 *KiloBytes* (ver figura 4.1.2-1 para observar que todos los archivos tienen un tamaño total de 426.0 *KiloBytes*)

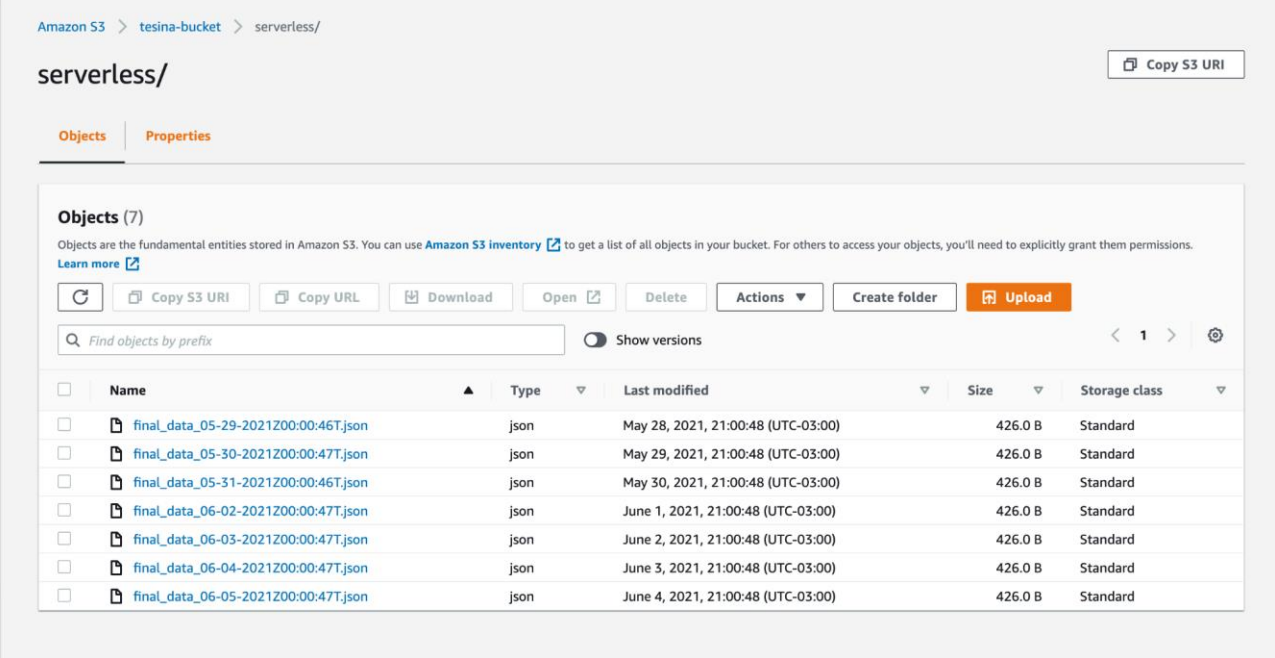

Figura 4.2.2-1: directorio serverless/ dentro de tesina-bucket contiene todos los archivos resultantes de la ejecución del ETL tradicional.

La diferencia entre el script ejecutado por el esquema tradicional y aquel ejecutado por la función lambda (ver 9.3 Script desarrollado en Python 3.8 para ETL en esquema serverless) es que en el último se deben cambiar algunos detalles para poder ser ejecutado.

Para empezar, debido a la falta de un *scheduler* frente a la ausencia de un servidor, debe existir una función dentro del script llamada *lambda_handler.* Esta actúa como punto de entrada al *script* cuando la función lambda es ejecutada. En el caso del prototipo implementado, el *han*dler ejecuta la función *run_etl* (llamada de la misma forma en el esquema tradicional 9.1 Script desarrollado en Python 3.8 para ETL en esquema tradicional)

Asimismo, frente a la ausencia de directorios donde descargar los archivos extraídos que contienen la fuente de datos que alimentan el ETL, se debe utilizar el directorio temporal llamado *tmp/* que provee el servicio de AWS Lambda para almacenar archivos temporales que sean utilizados dentro de la ejecución de la función. Este directorio no persiste datos más allá de la ejecución de la función lambda, y posee un tamaño máximo de 512 MB [24]. Por este motivo, se modificó el script original para hacer uso de este directorio y corregir los *path* referenciados en el script del servidor. Dichas modificaciones fueron mínimas y llevaron un tiempo total de completitud menor de 15 minutos. Estos tiempos resultan despreciables en una escala de horas semanales, con lo cual no es justificable contemplarlos en el análisis de la migración de un esquema tradicional en la nube a un esquema *serverless.*

A continuación, en la figura 4.2.2-2, se puede observar los *logs* de ejecución de la función *Lambda*, donde la duración total de ejecución fue de 716,46 ms (milisegundos) y la memoria utilizada fue de 79 MB (*MegaBytes)*, menor a la memoria total provista por el servicio de 128 MB. Esta memoria total puede ser incrementada dentro de los límites del servicio [24], pero para poder utilizarlo dentro de la capa gratuita no puede exceder los 128 MB de utilización de memoria RAM.

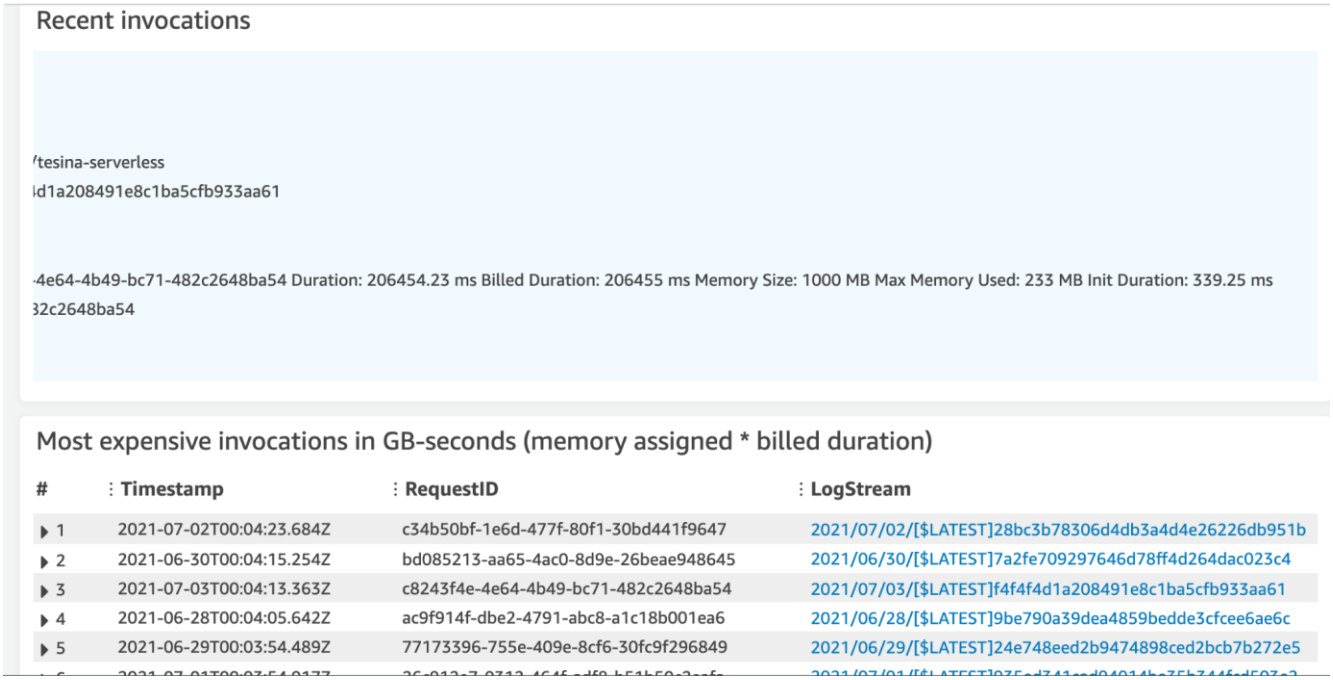

Figura 4.2.2-2: Logs con detalles de la ejecución de la función lambda. observables desde la consola de AWS.

En algunas ejecuciones el proceso tuvo un tiempo de ejecución menor, de 444 milisegundos, como se puede observar en la siguiente figura 4.2.2-3.

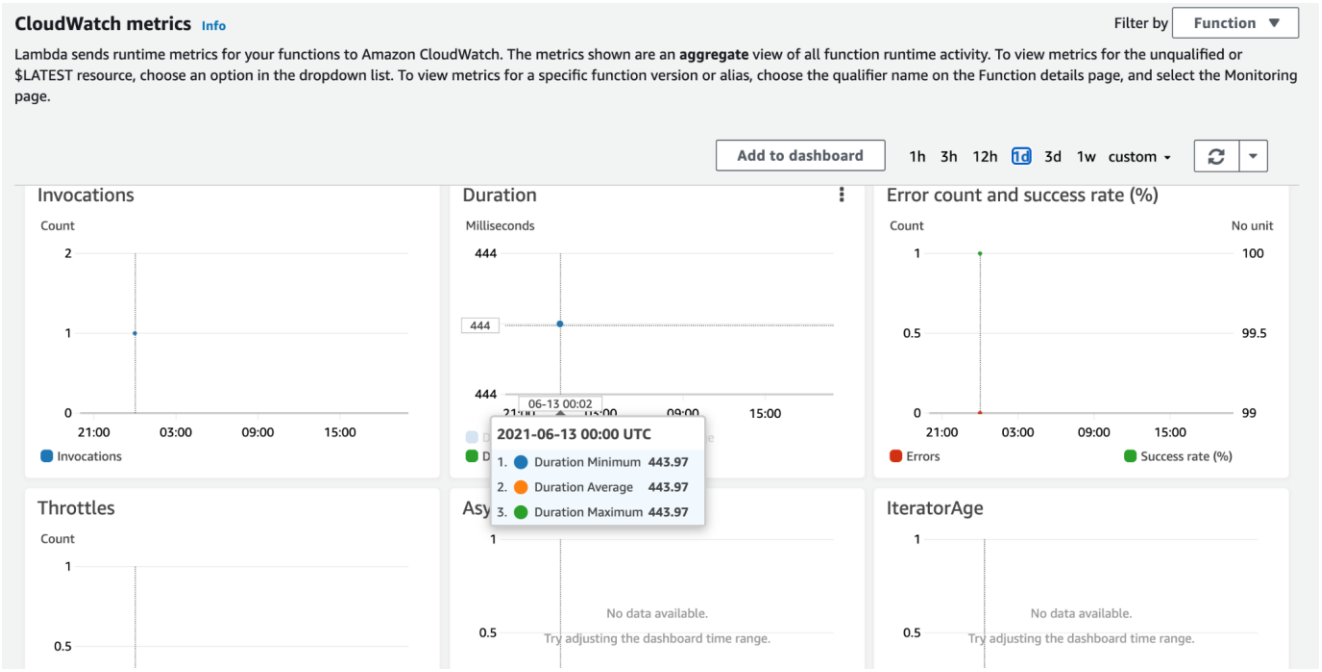

Figura 4.2.2-3: métricas operativas de la función Lambda observables desde la consola de AWS.

4.2.3 Análisis Comparativo

Dado que ambos prototipos fueron implementados exitosamente y se obtuvieron métricas operativas de su ejecución, resta únicamente comparar ambas implementaciones y verificar en base a las métricas operativas recolectadas si un prototipo es más eficiente que su contraparte.

Es importante tener en cuenta que la comparación, así como los resultados de la misma, se encuentran limitados por el alcance de esta investigación, y por ende, no deben extrapolarse y generalizarse a situaciones que no estén contenidas dentro de los límites de este trabajo.

El experimento de esta investigación y la comparación de los resultados obtenidos sirven únicamente para evaluar la posibilidad de migrar tareas automatizadas recurrentes desde esquemas tradicionales en la nube hacia una arquitectura *serverless* sin modificar la funcionalidad ni perder *performance*.

Las figuras 4.2.3-1 y 4.2.3-2 muestran la utilización de recursos computacionales, como uso de CPU o memoria RAM, dentro del servidor.

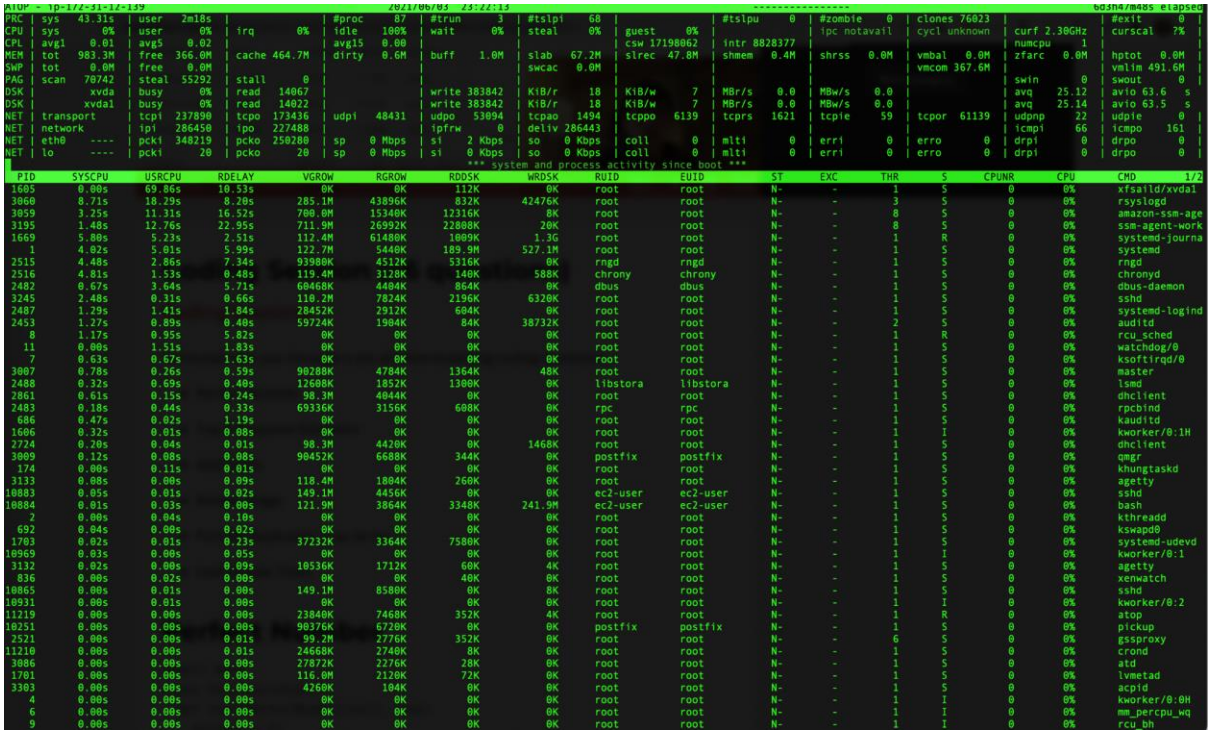

Figura 4.2.3-1: Métricas operativas y de utilización de recursos dentro del servidor.

Utilizando esta información junto con los datos obtenidos del producto *serverless* (observables en la figura 4.2.3-1 y 4.2.3-2 dentro de la sección anterior) es posible evaluar y comparar la utilización de cada uno de los recursos computacionales, con

el fin de verificar que el tiempo de respuesta o el uso de CPU o memoria no incrementa con la solución *serverless.*

Debido a que se ha verificado que los archivos resultantes de la ejecución de ambas implementaciones son idénticos, no será necesario incluirlos dentro de la tabla comparativa.

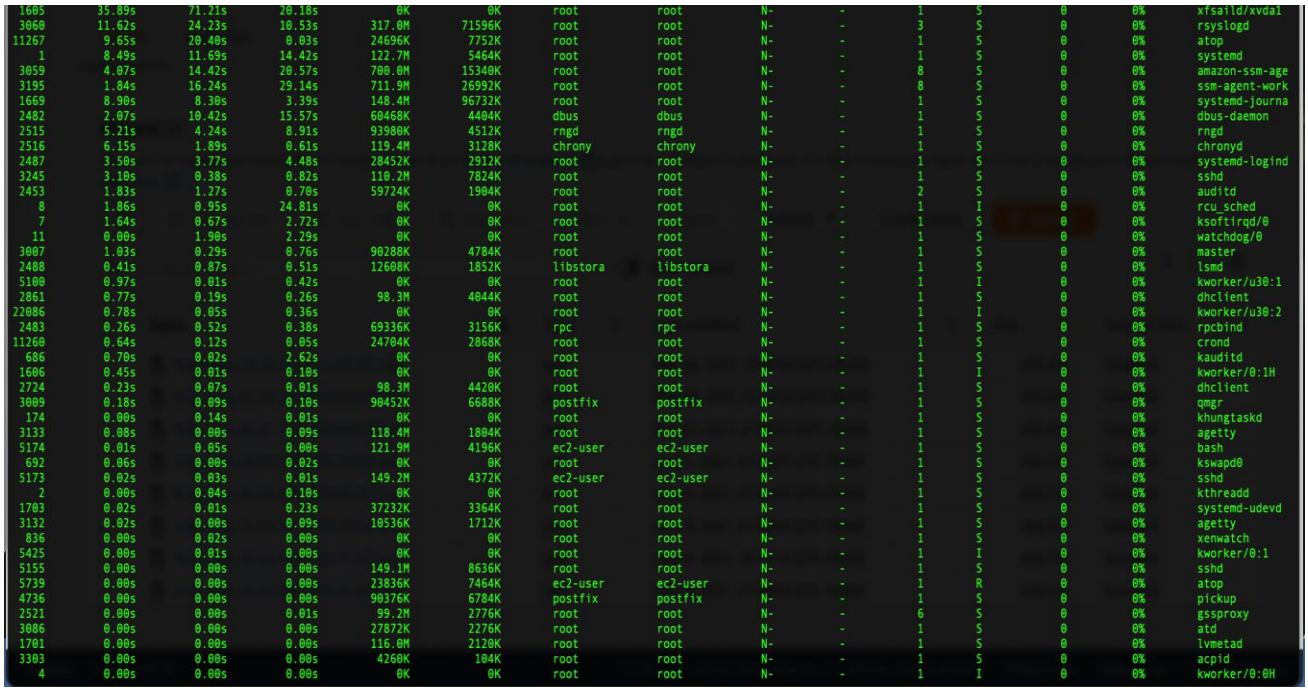

Figura 4.2.3-2: Métricas operativas y de utilización de recursos dentro del servidor.

Por ende, se realizó una tabla comparativa para analizar los datos obtenidos de las métricas de ambas implementaciones:

	Prototipo Tradicional	***Prototipo Serverless***
Tamaño de archivo	426 KiloBytes	426 KiloBytes
CPU	1 vCPU 2.5 GHz	1 vCPU 2.5 GHz
Memoria RAM	120 MB	79 MB
Tiempo de ejecución	120 ms	444 - 716 ms

Tabla 4.2.3-1: Tabla comparativa de utilización de recursos computacionales en ambas implementaciones.

En la tabla 4.2.3-1 se puede observar los resultados obtenidos de ambas implementaciones. En cuanto al uso de CPU, en ambas implementaciones AWS ofrece un vCPU (*Virtual Central Processing Unit)* con un único *core* para ejecutar el

procesamiento computacional. Esto se debe a que es la máxima cantidad de vCPUs que se pueden utilizar dentro de la capa gratuita.

En el caso de la memoria RAM, el resultado demuestra que la solución *serverless* utiliza aproximadamente un 30% menos, lo que sugiere una ligera mejora en la utilización de los recursos computacionales.

Sin embargo, es posible visualizar que el tiempo de ejecución fue menor en el servidor que en la *lambda function.* Esto se debe a que la segunda debe ejecutar procesos de arranque e invocaciones que son externas al desarrollo del producto y no son necesarias dentro de un servidor, ya que este ejecuta procesos de arranque cuando se enciende el hardware. Como los servidores suelen estar encendidos, estos procesos de arranque ya son ejecutados previamente, y por ende, los tiempos de respuesta de las ejecuciones del script son menores.

Asimismo resulta interesante analizar que el tiempo de ejecución del *lambda function* tuvo una variación en diferentes ejecuciones, y no fue estable en comparación al del servidor (que tuvo un delta despreciable de 30 ms). El delta del tiempo de ejecución en el caso de la implementación *serverless* fue de cerca de 300 ms, 10 veces más que la solución tradicional.

Debido a que AWS ofrece su capa gratuita con una serie de limitaciones que no pudieron ser evadidas, el experimento cuenta con un tamaño muy pequeño de datos. En consecuencia, se aprovechó el pequeño flujo de datos para analizar en detalle las variaciones de las métricas operativas.

Suponiendo un ETL cuya extracción de datos tiene una carga total de 17 MB, por ejemplo la descarga de 500 mil registros con 6 atributos cada uno en formato JSON, se ejecutaron ambas soluciones para evaluar su *performance* con tamaños más cercanos a la realidad.

Asimismo, se intentó reproducir el éxito obtenido con un archivo con tamaño total de 34 MB. Sin embargo, si bien el prototipo tradicional pudo soportar la carga del archivo y el tiempo de ejecución de media hora, el prototipo *serverless* devolvió *TIMEOUT ERROR* al alcanzar el límite de tiempo provisto por el servicio *lambda functions* de 15 minutos.

Esto permitió evaluar ambos prototipos y su escalabilidad frente a una fuente de datos mayor. Si bien es posible utilizar funciones Lambda y otros servicios *serverless* para ETLs que procesen fuentes de datos de este tamaño o mayores, se debe considerar que será necesario otro algoritmo y otra arquitectura para la ejecución de procesos que divida la carga operativa en múltiples funciones, evitando alcanzar los límites de los servicios utilizados.

Por consecuente, es posible extrapolar los resultados obtenidos y compararlos con el fin de evaluar las métricas operativas y verificar si se cumple la hipótesis en diferentes escenarios:

	Prototipo Tradicional		*Prototipo Serverless*	
	Archivo 1	***Archivo 2***	***Archivo 1***	***Archivo 2***
Cantidad de registros	800 mil	1.6 millones	800 mil	1.6 millones
Tamaño de archivo	17 MB	34 MB	17 MB	34 MB
Memoria RAM	483 MB	980 MB	232 MB	*Time Out*
Tiempo de ejecución	237 segundos	29 minutos	211 segundos	*Time Out*

Tabla 4.2.3-2: Tabla comparativa de utilización de recursos computacionales en ambas implementaciones.

En la tabla 4.2.3-2 se puede observar las distintas métricas operativas al ejecutar el mismo proceso ETL diseñado en la investigación pero extrayendo datos de mayor tamaño, aumentando la carga operativa de ambas implementaciones.

Analizando los resultados se observa que el prototipo tradicional pudo soportar ambos archivos de distinto peso ya que no existe una limitante para el tiempo de ejecución. En el peor de los casos si los recursos utilizados no alcanzan, la ejecución tardará más tiempo pero el servidor seguirá operando. Esto no es el caso con las funciones Lambda, que tienen un límite de ejecución de 15 minutos por función. Es por esto que extrayendo un archivo de 34 MB el prototipo implementado en serverless fallo con TIMEOUT ERROR. Sin embargo, como fue mencionado anteriormente, esto puede ser posible implementando otro algoritmo y otros servicios serverless.

Tomando la métrica de MyFEPS expuesta en la Tabla 4.1.3-1 para determinar la eficiencia en la utilización de memoria interna de los prototipos, se desarrolló la siguiente tabla comparativa:

	Prototipo Tradicional		*Prototipo Serverless*	
	Archivo 1	***Archivo 2***	***Archivo 1***	***Archivo 2***
	800 mil	1.6 millones	800 mil	1.6 millones
Tamaño de archivo	17 MB	34 MB	17 MB	34 MB
CMF	483 MB	980 MB	232 MB	*Time Out*

CMS	1 GB	1 GB	1 GB	1 GB
Valoración = 1 - CMF/CMS	0.517	0.020	0.768	*N/A*
Valoración Final		0.268		0.768

Tabla 4.2.3-3: Tabla comparativa de métricas obtenidas de la utilización de MyFEPS.

Siendo mayor la valoración final del prototipo *serverless*, resulta evidente que bajo la implementación de MyFEPS como métrica de evaluación de eficiencia en la utilización de memoria interna el prototipo *serverless* es más eficiente que el prototipo tradicional.

Resultados

Como se detalla en la sección 4.2.3 Análisis Comparativo, ambos prototipos implementados fueron comparados en base a su utilización de memoria y tiempos de ejecución. Como se puede observar en la Tabla 4.2.3-1 y Tabla 4.2.3-2, el prototipo *serverless* posee un menor tiempo de ejecución utilizando una menor cantidad de recursos computacionales.

En la tabla 5.1-1 se exponen los resultados obtenidos de la ejecución de ambos prototipos:

Tamaño de archivo	**426 KB**		**17MB**	
	Prototipo Tradicional	***Prototipo Serverless***	***Prototipo Tradicional***	***Prototipo Serverless***
CPU	1 vCPU	1 vCPU	1 vCPU	1 vCPU
Memoria RAM	120 MB	79 MB	484 MB	232 MB
Tiempo de ejecución	120 ms	444 ms	237 segundos	211 segundos

Tabla 5.1-1: Resultados de los prototipos implementados

Esto ocurrió para ambos casos de prueba, en donde se cambió el tamaño total de la fuente de datos procesada. En porcentaje, el prototipo *serverless* utilizó entre 35% y 50% menos memoria que el prototipo tradicional. Con respecto al tiempo de ejecución, para procesar un archivo de 426 *KiloBytes* de información el prototipo tradicional fue más rápido en comparación con el prototipo *serverless*, tardando un cuarto del tiempo en finalizar la ejecución. Sin embargo, al procesar un archivo de 17 MB de información, la ejecución del prototipo *serverless* fue 10% más rápida que la del prototipo tradicional. Esto se debe a que las ejecuciones de arranque que debe ejecutar *Lambda* no existen en el servidor, que ya se encuentra encendido y operando cuando se ejecuta la tarea. Esto es notable en archivos con poco tamaño, pero se torna despreciable cuando se alcanza una carga operativa de MegaBytes.

Con estos valores es posible asegurar que la hipótesis 1 se cumple independientemente de la carga operativa que se le de al prototipo implementado, siempre y cuando se respeten los límites del servicio *serverless* utilizado (para el caso de funciones Lambda ver sección 4.2.3 Análisis Comparativo).

En otras palabras, migrar tareas automatizadas recurrentes desde un esquema tradicional en la nube hacia un esquema *serverless* sin infraestructura asociada mejora la utilización de recursos computacionales de las tareas y reduce los tiempos de ejecución.

Los resultados obtenidos dentro de esta investigación están limitados por el alcance del trabajo y no se deben extrapolar sin considerar los límites de la implementación.

Conclusiones

Se implementaron y compararon dos prototipos, uno con infraestructura en la nube y el otro serverless. Con los resultados expuestos en la sección anterior se logró demostrar que la adopción de *serverless* para ejecutar tareas automatizadas recurrentes es más eficiente en la utilización de memoria y reduce los tiempos de ejecución en comparación a una arquitectura tradicional en la nube con infraestructura subyacente.

El prototipo *serverless* utilizó los recursos computacionales de forma más eficiente, siendo menor su utilización de memoria RAM y su tiempo de ejecución en comparación al prototipo implementado dentro de un esquema tradicional.

Se observa que la implementación de *serverless* en esquemas que cuentan con infraestructura en desuso o con sub utilización de recursos elimina la necesidad de infraestructura sin pérdida de *performance*. Por esta razón adoptar esta tecnología para realizar tareas automatizadas recurrentes es una manera factible de optimizar la utilización de recursos en la nube.

Es posible utilizar funciones Lambda (AWS *Lambda Functions*) para realizar múltiples tipos de tareas automatizadas recurrentes, siempre y cuando se logre limitar el tiempo de ejecución dentro de los 15 minutos. Asimismo, es posible que en ejecuciones que utilizan mayor cantidad de recursos computacionales que los usados en esta investigación logren reducir el tiempo de ejecución, obteniendo una implementación de *serverless* que soporte un incremento en la carga operativa.

6.1 Futuras investigaciones

Con respecto a investigaciones futuras en base a ésta se proponen las siguientes:

- Realizar un análisis de costos sobre el caso estudiado, verificando que dentro de los límites de este trabajo la implementación de *serverless* reduce los costos derivados de infraestructura en desuso.
- Desarrollar una fórmula para extrapolar y aplicar resultados de forma proporcional a la utilización de los recursos computacionales. Como hipótesis posible para esta investigación se sugiere: “Existe una relación matemática para la utilización de recursos computacionales y los tiempos de respuesta de las ejecuciones de procesos en esquemas *serverless*”.
- Estudiar la posibilidad de procesar archivos de mayor tamaño utilizando otros servicios en la nube que no fueron abordados en este trabajo, como por ejemplo, *step functions* de AWS, que permite ejecutar secuencias de funciones en forma concurrente.
- Comprobar que los resultados obtenidos pueden ser replicados fuera del alcance y límites que definieron este trabajo.

Referencias bibliográficas

[1] Alex Morrell, *Bank of America Says It Saves $2 Billion Per Year By Ignoring Amazon and Microsoft and Building Its Own Cloud Instead*. Business Insider. Oct. 19, 2017. Visitada en: Apr. 13, 2021 [online]. Accesible: https://www.businessinsider.com/bank-of-americas-350-million-internal-cloud-bet-striking-payoff-2019-10

[2] *Financial Engines Cuts Costs 90% Using AWS Lambda and Serverless Computing*. Amazon Web Services. Visitada en: Apr. 18, 2021 [online]. Accesible: https://aws.amazon.com/solutions/case-studies/financial-engines/

[3] Amos Sorgen, Rolando Titiosky, Martìn Santi, Paula Angeleri. *MyFEPS-Descripción de Atributos y Métricas.* Facultad de Tecnología Informática, Universidad de Belgrano. 2018. Visitada en: Jul, 11, 2021 [online]. Accesible: https://sites.google.com/a/comunidad.ub.edu.ar/myfeps/

[4] *IEEE Reference Guide*. IEEE Periodicals. 2018. Visitada en: Mar. 15, 2021. [online]. Accesible: https://ieeeauthorcenter.ieee.org/wp-content/uploads/IEEE-Reference-Guide.pdf

[5] *AstraZeneca's Genomics Data Processing Solution Runs 51 Billion Tests in 1 Day on AWS*. Amazon Web Services. 2021. Visitada en: Apr. 29, 2021. [online]. Accesible: https://aws.amazon.com/solutions/case-studies/astrazeneca/?did=cr_card&trk=cr_card.

[6] *Chelsea Builds Success off the Field with AWS*. Amazon Web Services. 2021. Visitada en: Apr. 29, 2021. [online]. Accesible: https://aws.amazon.com/solutions/case-studies/chelsea-case-study/?did=cr_card&trk=cr_card.

[7] *Garett McGrath, Serverless Computing: Applications, Implementation, and Performance,* Graduate Program in Computer Science and Engineering Notre Dame, Indiana April 2017.

[8] *What is serverless computing?* | *Serverless definition*. Cloudflare. Visitada en: Mar. 22, 2021. [online]. Accesible: https://www.cloudflare.com/learning/serverless/what-is-serverless/

[9] *Scott Carey. What is serverless computing and which enterprises are adopting it?*. Computerworld. Aug 13, 2019. Visitada en: Mar. 22, 2021. [online]. Accesible: https://www.computerworld.com/article/3427298/what-is-serverless-computing-and-which-enterprises-are-

[10] Erwin van Eyk, Laurens Versluis, Lucian Toader, Sacheendra Talluri. *Serverless is More: From PaaS to Present Cloud Computing*. IEEE Internet Computing. Sept. 10, 2018. Visitada en: Apr. 29, 2021. [online]. Accesible: https://www.researchgate.net/publication/328088482_Serverless_is_More_From_PaaS_to_Present_Cloud_Computing.

[11] William Fellows. *Red Hat's OpenShift Serverless for hybrid, legacy and greenfield.* 451 Research. Feb. 12, 2020.
[12] IBM Cloud Education. *FaaS (Function-as-a-Service)*. IBM. Jul 30, 2019. Visitada en: May. 10, 2021. [online]. Accesible: https://www.ibm.com/cloud/learn/faas.
[13] Peter Mell, Timothy Grance. *The NIST Definition of Cloud Computing.* NIST Special Publication 800-145. Sept. 20, 2011. Visitada en : May. 16, 2021 [online]. Accesible: http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf
[14] Paulo Neto. *Demystifying Cloud Computing.* Faculdade de Engenharia da Universidade do Porto Rua Dr Roberto Frias, PORTO, Portugal. Visitada en : May. 16, 2021 [online]. Accesible: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.458.9370&rep=rep1&type=pdf
[15] C.N. Höfer · G. Karagiannis. *Cloud computing services: taxonomy and comparison.* Publicado online: Jun. 19, 2011. Visitada el : May. 16, 2021 [online]. Accesible: https://link.springer.com/content/pdf/10.1007/s13174-011-0027-x.pdf
[16] *AWS Lambda*. Amazon Web Services. Visitada en: Mar. 22, 2021. [online]. Accesible: https://aws.amazon.com/lambda/
[17] *AWS Lambda execution environment*. Amazon Web Services. Visitada en : May. 16, 2021 [online]. Accesible: https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html
[18] *AWS Elastic Compute Service.* Amazon Web Services. Visitada en : May. 16, 2021 [online]. Accesible: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
[19] *AWS Simple Storage Service.* Amazon Web Services. Visitada en : May. 16, 2021 [online]. Accesible: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html
[20] Andrzej Cichocki, Helal A. Ansari, Marek Rusinkiewicz, Darrell Woelk. *Workflow and Process Automation: Concepts and Technology.* Springer Science+Business Media, LLC. 1998.
[21] What's IT automation? . Red Hat. Visitada en: May. 22, 2021. [online] Accesible: https://www.redhat.com/en/topics/automation/whats-it-automation
[22] Panos Vassiliadis, Alkis Simitsis, Spiros Skiadopoulos. *Conceptual modeling for ETL processes*. DOLAP '02: Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP. Nov. 2002. Visitada en: May. 22, 2021. [online] Accesible: https://dl.acm.org/doi/abs/10.1145/583890.583893
[23] *Serverless Computing*. Amazon Web Services. Visitada en: Mar. 22, 2021 [online]. Accesible: https://aws.amazon.com/serverless/
[24] *Lambda quotas*. Amazon Web Services. Visitada en: Jun. 12, 2021. [online]. Accesible: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

Bibliografias auxiliares consultadas

1. Peter Sbarski. *Serverless Architectures on AWS With examples using AWS Lambda*. Pag. 2. 2017
2. Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, David A. Patterson. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. 2019.
3. Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, Aleksander Slominski. *Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research*. 2017.
4. Paul Castro, Vatche Ishakian, Vinod Muthusamy, Aleksander Slominski. *The rise of serverless computing*. 2019.
5. Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, Chenggang Wu. *Serverless Computing: One Step Forward, Two Steps Back*. 2018.

Glosario

- Alta disponibilidad: es una arquitectura de software que minimiza los tiempos de caída (downtime), tendiendo a 0%. Tiene como característica el uso de Load Balancers para distribuir la carga entre los servidores y escalado automático para que la cantidad de servidores y Load Balancers crezcan con el tráfico creciente.
- API: Application Programming Interface. Es un conjunto de funciones, subrutinas y procedimientos que ofrecen una capa de abstracción que es utilizada por otro software. Ejemplo, una API que maneja la carga, búsqueda, y modificación de ítems de un ecommerce.
- ASG: sus siglas significan Auto Scaling Group (grupo de auto escalado en inglés). Es un servicio ofrecido por los proveedores cloud para manejar de forma automática el escalado de servidores frente al incremento en el tráfico entrante. Cuando el tráfico aumenta, se registra un incremento en el tráfico entrante a los servidores (usualmente medido en porcentajes) y al llegar al límite establecido, el ASG despliega automáticamente servidores para aligerar la carga y restablecer el tráfico entrante a las métricas deseadas.
- AWS: Amazon Web Service es la plataforma cloud de Amazon.
- AWS CLI: *Amazon Web Services Command Line Interface* es un paquete de comandos que pueden ser ejecutados dentro de una terminal, que permiten realizar acciones sobre recursos de AWS. Pueden ser utilizados para realizar tareas automatizadas que no requieran interacción manual a través de la interfaz gráfica de AWS, conocida como *AWS management console.*
- CDN: sus siglas significan *Content Delivery Network* (Red de Distribución de Contenidos en español). Es una red de computadoras ubicadas cerca del cliente que contienen copias de datos de los servidores con el fin de maximizar el ancho de banda de la red y reducir el impacto en los tiempos de respuesta generado por los cuellos de botella del internet.
- *Cloud*: servicios en internet que resuelven problemas recurrentes para los desarrolladores, como por ejemplo, creación de instancias o máquinas virtuales, almacenamiento de archivos en directorios compartidos, bases de datos, logs, etc.
- Cron Table: en los sistemas operativos Unix, Cron es un administrador de procesos en segundo plano que son ejecutados en intervalos de tiempo predefinidos.
- Data Warehouse: son bases de datos desnormalizadas e históricas, donde se guardan datos para ser explotados por herramientas de análisis y visualización de datos, como Qlik, Tableau, PowerBI, etc.

- *Downtime*: es el porcentaje medido sobre la cantidad de consultas al servidor que el mismo es incapaz de responder o servir. En otras palabras, el tiempo que un servidor se encuentra inaccesible.
- ETL: las siglas significan Extract (extraccion), Transform (transformación) y Load (carga). Es un proceso de transferencia de datos, donde se descargan de alguna base o archivo estructurado, se transforma con alguna lógica, y se carga en una base final. Son especialmente utilizados para cargar datos en data warehouses.
- FaaS: Function as a Service es un paradigma de tecnología en donde se ofrece la ejecución de código como servicio sin necesidad de infraestructura subyacente. Se ofrece como servicio la ejecución de cierta lógica programada.
- GCP: Google Cloud Platform es la plataforma cloud de Google.
- LB: los balanceadores de carga se encargan de distribuir el tráfico de los clientes a los distintos servidores para que los tiempos de respuesta a los clientes sean lo más rápido posible y no se generen cuellos de botella.
- REST: Representational State Transfer. Es un estilo de arquitectura utilizado para conectar sistemas basados en el protocolo HTTP y devolver datos en formatos XML o JSON.
- RESTful: un servicio web que utiliza la arquitectura REST.
- SLA: *Service level agreement*, Acuerdo de Nivel de Servicio, especifica las condiciones mínimas de servicio, tiempos de caída y penalidades por incumplimento, acompaña a un contrato de servicio y lo imita en tiempos y responsabilidades.
- SRE: *Site Reliability Engineer* es un puesto de trabajo dentro de las empresas de tecnología. El SRE tiene la responsabilidad de asegurar el correcto aprovisionamiento y configuración de la infraestructura y proveer servicios de *DevOps* para la compañía, así como controlar y monitorear los recursos existentes.

Anexos

9.1 Script desarrollado en Python 3.8 para ETL en esquema tradicional

```
import                                                    logging
import                                                      boto3
import                                                         os
import                                                       json
import                                                     shutil
from          botocore.exceptions          import          ClientError
from               datetime               import               datetime

now                           =                          datetime.now()
timestamp             =             now.strftime("%m-%d-%YZ%H:%M:%ST")

AWS_ACCESS_KEY                    =                   'AWS_ACCESS_KEY'
AWS_SECRET_KEY                    =                   'AWS_SECRET_KEY'
AWS_REGION                         =                        'us-east-2'

BUCKET                         =                        'tesina-bucket'

S3_CLIENT                         =                    boto3.client('s3',
                        aws_access_key_id=AWS_ACCESS_KEY,
                        aws_secret_access_key=AWS_SECRET_KEY,
                        region_name=AWS_REGION
                        )
BASE_PATH          =          f"{os.path.realpath(os.getcwd())}/tmp"


def                                                      run_etl():
    os.mkdir(BASE_PATH)
    logging.info("running              extract              process")
    extract()
    logging.info("running            transform              process")
    transform()
    logging.info("running                load               process")
    success                          =                        load()
```

```
    if success:
        logging.info("Successfully performed ETL")
    else:
        logging.error("There was an error ")
    cleanup()


def extract():
    try:
        S3_CLIENT.download_file(BUCKET, 'data/data.json',
f"{BASE_PATH}/data.json")
    except ClientError as e:
        logging.error("There was an error downloading file:
"+e.__str__())
    return


def transform():
    transformedData = []
    try:
        logging.info("Opening extracted data")
        # transform extracted data
        with open(f"{BASE_PATH}/data.json") as file:
            logging.info("Successfully opened extracted data file")
            data = json.load(file)
            logging.info("Successfully loaded json data")
            for element in data:
                logging.info(f"Iterating element
{list(data).index(element)}")
                if element['consumer_type'] == 'random':
                    tData = {
                        "name": element['name'],
                        "surname": element['surname'],
                        "email": element['email'],
                        "send_add": True,
                        "ad_type": 'digital',
                        "aggressive": True,
                        "reason": "random consumer needs aggressive
advertising on holiday to stimulate consumption"
```

```
                }
            if element['consumer_type'] == 'consistent':
                tData = {
                    "name": element['name'],
                    "surname": element['surname'],
                    "email": element['email'],
                    "send_add": True,
                    "ad_type": 'digital',
                    "aggressive": False,
                    "reason": "consistent consumer found. Basic
ad should suffice"
                }
            transformedData.append(tData)
            logging.info(f"appended element to array")
        # dump data in new json file
        with open(f"{BASE_PATH}/transformed_data.json", 'w') as
file:
            logging.info("opened transformed_data file")
            json.dump(transformedData, file)
            logging.info("written transformed_data file")
    except Exception as e:
        logging.error("There was an error transforming data:
"+e.__str__())

    return


def load():
    """Upload a file to an S3 bucket used for experiment

        :return: True if file was uploaded, else False
    """
    path = f"{BASE_PATH}/transformed_data.json"
    if not os.path.exists(path):
        logging.error("Missing transformed_data.json, possible
failure in previous step")
        return False
    try:
        logging.info("uploading final_data file to S3")
```

```
        response = S3_CLIENT.upload_file(path, BUCKET,
f"traditional/final_data_{timestamp}.json")
    except ClientError as e:
        logging.error(e)
        return False
    return True

def cleanup():
    shutil.rmtree(BASE_PATH)

if __name__ == "__main__":
    run_etl()
```

9.2 Script desarrollado en Bash para ejecutar ETL en esquema tradicional

```
#!/bin/bash
python3 /home/ec2-user/app/__main__.py
```

9.3 Script desarrollado en Python 3.8 para ETL en esquema serverless

```
import logging
import boto3
import os
import json
import shutil
from botocore.exceptions import ClientError
from datetime import datetime

now = datetime.now()
timestamp = now.strftime("%m-%d-%YZ%H:%M:%ST")

AWS_ACCESS_KEY = 'AWS_ACCESS_KEY'
AWS_SECRET_KEY = 'AWS_SECRET_KEY'
AWS_REGION = 'us-east-2'
```

```
BUCKET                              =                                'tesina-bucket'

S3_CLIENT                           =                           boto3.client('s3',
                        aws_access_key_id=AWS_ACCESS_KEY,
                        aws_secret_access_key=AWS_SECRET_KEY,
                        region_name=AWS_REGION
                        )
BASE_PATH                            =                           "/tmp/tmp_dir"


def                                                                    run_etl():
    #               if              not              os.path.exists(BASE_PATH):
    os.mkdir(BASE_PATH)
    logging.info("running                    extract                    process")
    extract()
    logging.info("running                   transform                   process")
    transform()
    logging.info("running                      load                     process")
    success                                =                                load()
    cleanup()
    if                                                                    success:
        logging.info("Successfully                performed                ETL")
        return                                                               {
            'statusCode':                                                  200,
            'body':           json.dumps('Hello           from           Lambda!')
        }
    else:
        logging.error("There          was          an          error          ")
        return                                                               {
            'statusCode':                                                  500,
            'body':         json.dumps('There        was        an        error')
        }

    return


def                                                                  extract():
    try:
        S3_CLIENT.download_file(BUCKET,                        'data/data.json',
f"{BASE_PATH}/data.json")
    except                   ClientError                  as                  e:
        logging.error("There  was  an  error  downloading  file:  "  +  e.__str__())
    return


def                                                                transform():
    transformedData                              =                          []
```

```
    try:
        logging.info("Opening extracted data")
        # transform extracted data
        with open(f"{BASE_PATH}/data.json") as file:
            logging.info("Successfully opened extracted data file")
            data = json.load(file)
            logging.info("Successfully loaded json data")
            for element in data:
                logging.info(f"Iterating element {list(data).index(element)}")
                if element['consumer_type'] == 'random':
                    tData = {
                        "name": element['name'],
                        "surname": element['surname'],
                        "email": element['email'],
                        "send_add": True,
                        "ad_type": 'digital',
                        "aggressive": True,
                        "reason": "random consumer needs aggressive advertising on holiday to stimulate consumption"
                    }
                if element['consumer_type'] == 'consistent':
                    tData = {
                        "name": element['name'],
                        "surname": element['surname'],
                        "email": element['email'],
                        "send_add": True,
                        "ad_type": 'digital',
                        "aggressive": False,
                        "reason": "consistent consumer found. Basic ad should suffice"
                    }
                transformedData.append(tData)
                logging.info(f"appended element to array")
        # dump data in new json file
        with open(f"{BASE_PATH}/transformed_data.json", 'w') as file:
            logging.info("opened transformed_data file")
            json.dump(transformedData, file)
            logging.info("written transformed_data file")
    except Exception as e:
        logging.error("There was an error transforming data: " + e.__str__())

    return


def load():
    """Upload a file to an S3 bucket used for experiment
```

```
    :return: True if file was uploaded, else False
    """
    path = f"{BASE_PATH}/transformed_data.json"
    if not os.path.exists(path):
        logging.error("Missing transformed_data.json, possible failure in previous step")
        return False
    try:
        logging.info("uploading final_data file to S3")
        response = S3_CLIENT.upload_file(path, BUCKET, f"serverless/final_data_{timestamp}.json")
    except ClientError as e:
        logging.error(e)
        return False
    return True


def cleanup():
    shutil.rmtree(BASE_PATH)
    return


def lambda_handler(event, context):
    run_etl()
```

9.4 Configuración del cron table en el servidor

```
0 0 * * * /home/ec2-user/app/execute.sh
```