Universidad Nacional de San Juan

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES

Departamento de Informática

Licenciatura en Ciencias de la Computación

Evaluación del comportamiento de los lenguajes de programación en una plataforma serverless. Amazon Web Services como caso de estudio.

Autor: Matías Rodríguez

Registro: 18049

Asesor: Lic. Nelson R. Rodríguez

Tabla de Contenido

Agradecimientos

Me gustaría expresar mi gratitud hacia mi director de trabajo final Lic. Nelson Rodríguez quien me guió y apoyó desde el principio en este trabajo para poder llegar al esquema adecuado. Su paciencia y dedicación fueron fundamentales para concluir este trabajo.

Asimismo, agradezco a la institución y docentes por su trabajo en esta hermosa carrera y transmitir sus conocimientos que permiten a tantos alumnos adaptarse rápidamente a los cambios abruptos de este rubro tan lindo y apasionante como es el informático.

Un trabajo de investigación es siempre fruto de ideas, proyectos y esfuerzos previos que corresponden a otras personas. En este caso mi más sincero agradecimiento a Fernando Castillo ex líder y compañero de trabajo quien siempre me brindó una mano ofreciendo sus amplios conocimientos sobre la plataforma de servicios web de Amazon.

Gracias a mi madre, mi novia, mi tía y mi hermano. Todo este proyecto no hubiera sido posible sin el apoyo vital de las personas que me rodean en mi día a día y que me brindaban el apoyo y la energía para poder crecer como persona y como profesional.

A todos, muchas gracias.

CAPÍTULO 1

Presentación y Descripción

1.1 Introducción

Serverless Computing es una arquitectura o modelo de ejecución en el cloud, alternativo al modelo tradicional. Ofrece numerosas ventajas sobre una arquitectura monolítica, como aportar agilidad, innovación, un mejor escalado automático, flexibilidad en el desarrollo y una mejor evaluación y control de los costos. Surgió como una evolución de microservicios corriendo en contenedores e implementando funciones, por lo cual a veces se lo denomina función como servicio (FaaS) (CNCF, 2018).

En el presente trabajo se evalúa el comportamiento de distintos lenguajes de programación, bajo un enfoque serverless, en la plataforma Amazon Web Services. Los lenguajes de programación a considerar son Python, Java y JavaScript, los cuales se encuentran entre los más populares. Debido a que la comparación de los lenguajes se puede llevar a cabo en diferentes aspectos o tomando diferentes tipos de métricas, se estimó oportuno realizar las pruebas mediante operaciones CRUD (Create, Read, Update, Delete).

Las operaciones CRUD son base para muchos software de aplicación empresarial, tal como sistemas de ventas, de recepción de pedidos, software para hoteles, colegios, de seguridad, entre otros. Mediante estas operaciones se puede agregar un cliente (alumno, paciente, producto, etc.), leer datos (leer la tabla completa, una o varias columnas), modificarlos o eliminarlos.

AWS está en permanente crecimiento y facilita el trabajo a muchos desarrolladores. Los proveedores de la nube ofrecen integrar todas las funciones necesarias requeridas para desarrollar un sistema, además de tener la capacidad de procesar enormes volúmenes de datos y dispositivos.

Los servicios de AWS a emplear son: Lambda, DynamoDB, CloudWatch, CloudWatch Metric, Logs y Log Insight. Para seleccionar y evaluar los lenguajes se implementan test con la herramienta Artillery.

La adopción de plataformas de función como servicio (FaaS) requiere abordar una serie de interrogantes como: ¿El cold start o arranque en frío, es equivalente en los distintos lenguajes? ¿El impacto del mismo es relevante cuando la cantidad de solicitudes es elevada? ¿El tamaño de memoria asignada a una función es relevante? ¿Qué lenguaje tiene mejor escalabilidad? ¿Cuál es el lenguaje o combinación de lenguajes más conveniente para operaciones CRUD?

Es importante mencionar que el uso del servicio Lambda se debe a que éste permite desarrollar y gestionar funciones independientes. Estas funciones se definen como pequeñas unidades de código que se ejecutan en respuesta a un evento.

1.2 Antecedentes

Cloud computing es un modelo de arquitectura, pero también de negocios, que permite brindar recursos bajo demanda tanto de infraestructura, como de plataforma y de software, además de nuevos servicios que han surgido recientemente.

Como una evolución de Cloud Computing, surgieron primero los microservicios y los contenedores, para luego aparecer el término serverless computing, el cual puede dar lugar a equívocos, dando la impresión de que se habla de computación sin necesidad de disponer de servidor. Esto no es exactamente así. En serverless computing existen por supuesto servidores, pero en vez de ser administrados por una persona, están gestionados de forma automatizada por un proveedor cloud.

La idea de este modelo bajo demanda o de "pago por uso" había aparecido anteriormente, más precisamente en 2006 con una plataforma llamada Zimki. Uno de los primeros usos del término serverless nace de Iron, en 2012 con su producto llamado IronWorker, una plataforma de trabajo bajo demanda basada en contenedores.

Desde ese entonces hay nuevas implementaciones de serverless tanto en la nube privada como en la pública. Primero hubo ofertas de BaaS, como Parse en 2011 y Firebase en 2012 (adquirido por Facebook y Google, respectivamente). En noviembre de 2014 se lanzó AWS Lambda, y a principios de 2016 se vieron anuncios para IBM OpenWhisk en Bluemix (ahora IBM Cloud Functions, con el proyecto principal de código abierto Apache OpenWhisk), Google Cloud Functions y Microsoft Azure Cloud Functions. Huawei Function Stage se lanzó en 2017. También hay numerosos frameworks de código abierto. Cada uno de estos, tanto públicos como privados, tiene conjuntos únicos de tiempos de ejecución de lenguaje y servicios para manejar eventos y procesar datos (CNCF, 2018).

Un mapeo sistemático realizado por Al-Ameen y Spillner en "A systematic and open exploration of FaaS research indica que en 2016, que es cuando surge serverless, solo se publicaron 7 trabajos de investigación y además del total de trabajos solo 4 tienen que ver con FaaS (Al-Ameen & Spillner, 2018).

Debido a que la migración de aplicaciones a las arquitecturas serverless no es una tarea sencilla, algunos autores proponen el modelo de actor, que es un método computacional con un patrón muy popular para crear aplicaciones concurrentes.

El modelo simplifica el trabajo de composición paralela y ejecuciones distribuidas mediante el uso de una unidad básica de cálculo: el actor (Pons et al., 2018).

Un actor es una unidad de cálculo aislada e independiente, con ejecución de un solo subproceso. Muchos actores se pueden ejecutar de forma simultánea e independiente entre sí para crear aplicaciones complejas. Esta propuesta puede ser una solución para la migración mediante estos patrones, aunque no hay otros trabajos similares.

Otro trabajo similar, publicado por Yussupov en el 2019, desarrolla un análisis sistemático sobre plataformas y herramientas para FaaS. En dicha publicación se afirma que teniendo en cuenta el ítem modelos de programación solo existen 3 trabajos. Por otro lado también afirma que la mayoría de las publicaciones (36%) se basan en AWS Lambda para realizar los desarrollos y pruebas. De las tres publicaciones citadas ninguna trata el tema de programación específicamente. El trabajo de Abhinav Jangda, presenta una formalización de la computación serverless, la publicación de Meissner presenta una plataforma para aplicaciones serverless con soporte de computación reactiva, y por último los investigadores Moczurad y Malawski en 2018 desarrollan un framework para

computación serverless (Abhinav Jangda et al., 2019; Meissner et al.. 2018; Moczurad & Malawski, 2018; Yussupov et al., 2019).

Debido a que es una tecnología reciente, no existen gran cantidad de trabajos de investigación que tengan en cuenta los diversos aspectos que se pueden analizar de la misma, sumado al hecho de que está en constante evolución mediante el surgimiento de nuevas funcionalidades tanto para FaaS como para BaaS.. Según un survey de Hassan del 2021, se afirma que: “Sin embargo, hasta la fecha hay una falta de estudios en profundidad que ayuden a los desarrolladores e investigadores a mejorar y comprender la importancia de la informática sin servidor en diferentes contextos” (Hassan et al., 2021).

1.3 Justificación

En los últimos años se puede notar una clara tendencia hacia el desarrollo de aplicaciones web, debido a las ventajas que estas ofrecen y a la utilización de servicios cloud para alojarlas. Cada vez crece más la demanda de estas aplicaciones, por lo que las empresas suelen querer reducir los tiempos de desarrollo. Con la aparición de la computación serverless se han logrado reducir notablemente las operaciones de mantenimiento y configuración de los servidores.

Administrar servicios en la nube no es una tarea fácil en absoluto. Los autores de Cloud Programming Simplified: A Berkeley View on Serverless Computing han abordado varios desafíos al administrar un entorno de Cloud por parte de un usuario, como la disponibilidad, el balanceo de carga, el escalado automático, la seguridad, la supervisión, etc. Por ejemplo, el usuario de Cloud tiene que garantizar la disponibilidad de los servicios en los que si se produce un fallo de un solo equipo, no afecta a todos los servicios (Jonas et al., 2021).

Los programadores, gracias a serverless, pueden concentrarse en el código y no pierden tiempo con temas correspondientes a la configuración de los servidores. Otra ventaja de utilizar los servicios de este modelo, es que se reducen los costos de infraestructura comparados con el Cloud tradicional, dado que no se necesita tener un servidor que esté activo permanentemente y generando costos incluso en los tiempos donde no se está ejecutando código.

Sin embargo, la tarea de programación no resulta sencilla, dado que además de la falta de modelos de programación se suman la falta de herramientas de depuración. Además, se requieren herramientas de monitoreo, ya que los desarrolladores deben supervisar la aplicación y observar cómo trabajan las funciones. Entornos de desarrollo integrados más avanzados (IDE) son necesarios, para que los desarrolladores puedan realizar funciones de refactorización, como fusionar o dividir funciones, y revertir funciones a la versión anterior, también hay que considerar que a la fecha no existen el equivalente de stack trace para serverless (Fox et al., 2017).

La idea de este trabajo es evaluar en diversos aspectos a los lenguajes más populares que se utilizan hoy en día en el desarrollo web bajo un enfoque serverless. Teniendo en cuenta no solo parámetros de performance y uso de recursos, sino además por todo lo mencionado en el párrafo anterior. Para llevar a cabo este análisis, se implementarán operaciones CRUD, que permitirán concluir en

resultados interesantes a tener en cuenta a la hora de elegir un lenguaje de programación sobre plataformas serverless.

Teniendo en cuenta que serverless permite que se puedan utilizar de forma combinada los distintos lenguajes de programación, por ejemplo invocando una función Java desde JavaScript con NodeJS, es que los resultados de este trabajo final, puede resultar de sumo interés no solo para la academia, sino también para los desarrolladores

1.4 Objetivos

1.4.1 Objetivo General

Evaluar el comportamiento de los distintos lenguajes de programación utilizados en la plataforma AWS con un enfoque serverless mediante la ejecución de operaciones CRUD.

1.4.2 Objetivos específicos

- Analizar el servicio Lambda de la plataforma AWS.
- Examinar los servicios de bases de datos en la plataforma AWS.
- Determinar y categorizar los distintos lenguajes de programación que ofrece el servicio Lambda.
- Determinar los tests necesarios para comparar la eficiencia y otros parámetros de los lenguajes de programación, utilizando operaciones CRUD.
- Implementar los casos de test y contrastar los resultados obtenidos.

CAPÍTULO 2

Marco Referencial

2.1 Introducción

La computación serverless surge para poder solucionar todos esos problemas tediosos con los que un desarrollador tiene que lidiar cuando se habla de servidores. Aunque su significado no sea realmente la desaparición de los servidores en absoluto. Cuando se habla de serverless se pueden dejar todas las preocupaciones con respecto a la gestión de infraestructura hacia un tercero. Un desarrollador ya no debe capacitarse o por lo menos no de la misma forma que antes para poder lidiar con la infraestructura de un proyecto. Gracias a esto se puede centrar todo el esfuerzo en la funcionalidad del sistema o aplicación.

2.2 Serverless

En el artículo “The rise of serverless computing” se considera que la computación serverless significa ocuparse menos de los servidores. Los desarrolladores no necesitan preocuparse por los detalles de bajo nivel de la administración y el escalado de servidores y solo pagan cuando procesan solicitudes o eventos. También lo definen de la siguiente manera:

“Es una plataforma que oculta el uso del servidor a los desarrolladores y ejecuta código bajo demanda, escalado automáticamente y facturado solo por el tiempo que el código se está ejecutando”. Esta definición captura las dos características claves de serverless; costo y elasticidad.

Costo: Solo se factura por aquello que se está ejecutando (pago por uso). Como los servidores y su uso no forman parte del modelo de computación sin servidor, es natural pagar solo cuando el código se está ejecutando y no por servidores inactivos. Como el tiempo de ejecución puede ser corto, entonces se debe cobrar en unidades de tiempo detalladas como cientos de milisegundos y los desarrolladores no necesitan pagar los gastos generales de creación o destrucción de servidores (como el tiempo de arranque de la máquina virtual). Este modelo de costos es muy atractivo para cargas de trabajo que deben ejecutarse ocasionalmente. Serverless esencialmente admite "escalado a cero" y evita la necesidad de pagar por servidores inactivos. El gran desafío para los proveedores de la nube es la necesidad de programar y optimizar los recursos de la nube.

Elasticidad: escala de cero a "infinito". Dado que los desarrolladores no tienen control sobre los servidores que ejecutan su código, ni conocen la cantidad de servidores en los que se ejecuta su código, las decisiones sobre el escalado se dejan a los proveedores de la nube. Los desarrolladores no necesitan escribir políticas de escalado automático ni definir el uso a nivel de máquina (CPU, memoria, etc.). Depende del proveedor de la nube iniciar automáticamente más ejecuciones paralelas cuando haya más demanda. Los desarrolladores también pueden asumir que el proveedor

de la nube se encargará del mantenimiento, las actualizaciones de seguridad, la disponibilidad y el monitoreo de confiabilidad de los servidores (Castro et al. (2019).

2.2.1 Arquitectura serverless: Servicios

La computación serverless puede dividirse en dos modelos de servicios de computación en la nube: Function as a Service (FaaS) y Backend as a Service (BaaS).

2.2.1.1 Function as a Service - FaaS

Function as a Service, conocidas por sus siglas FaaS, son plataformas donde corren pequeñas unidades de código que se ejecutan en respuesta a un evento. El evento puede ser, entre otros, una solicitud HTTP, una operación en una base de datos o un mensaje de una cola de mensajes. Los usuarios despliegan código a las plataformas FaaS y éste es ejecutado bajo demanda en función de los eventos. Como este servicio provee manejo de escalado e infraestructura parece que no existe un servidor para el usuario (Paakkunainen, 2019).

La figura 1, muestra un ambiente FaaS como una composición de tres componentes: Un controlador FaaS, una fuente de eventos y por último, instancias de funciones. Se puede ver cómo estos componentes interactúan, siendo el controlador FaaS el que se ocupa de administrar el despliegue de las funciones FaaS, monitorear y escalar las instancias y de controlar las funciones y las fuentes de eventos. Las fuentes de eventos permiten que se construyan instancias nuevas a partir de estos y una instancia de función es el entorno de ejecución para una función simple o microservicio.

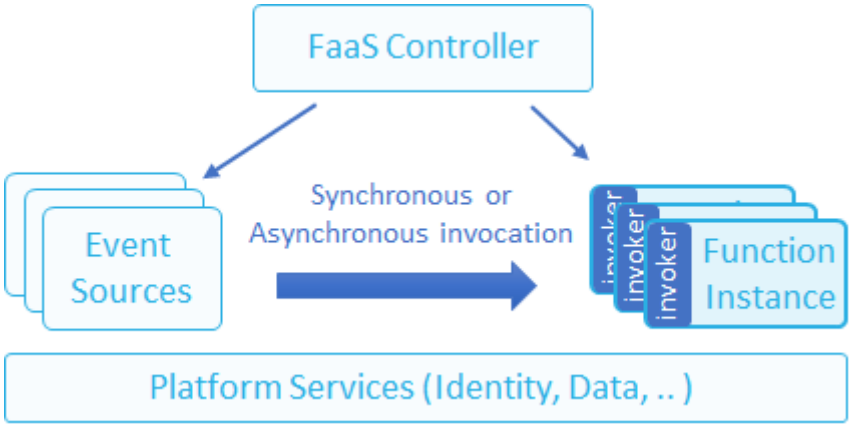


Figura 1. Modelo de procesamiento de funciones FaaS. Recuperado de (CNCF, 2018)

2.2.1.2 Backend as a Service - BaaS

Se refiere a servicios de terceros, los cuales proveen una API para reemplazar o complementar un subconjunto de funcionalidades en una aplicación. Estos servicios son auto escalables y operan de forma transparente al usuario dando la impresión de que no hay servidores. BaaS es comúnmente

usado por las aplicaciones web SPA (single-page applications) o aplicaciones móviles, con el fin de aprovechar las bases de datos accesibles por la nube o los servicios de autenticación (CNCF, 2018).

2.3 Casos de uso

Para los proveedores de nube, la computación serverless promueve el crecimiento porque hace que sea más fácil de programar. Esto ayuda a incorporar nuevos clientes y a que los antiguos puedan hacer uso de más de los servicios ofertados. Debido a los beneficios de serverless, los proveedores de la nube también pueden encontrar de forma sencilla aquellos recursos no utilizados en los cuales poner a ejecutar tareas.

La figura 2 muestra los resultados obtenidos sobre los casos de usos más populares, según una encuesta realizada durante el 2018 en la comunidad serverless.

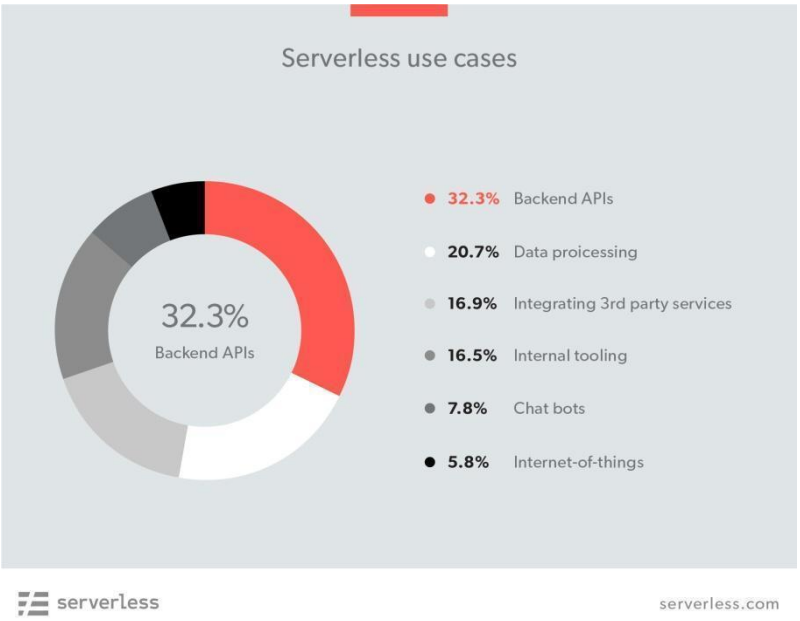


Figura 2. Porcentaje de uso de serverless, en informática, según una encuesta de 2018. Recuperado de (Passwater, 2018)

2.4 AWS Lambda

Amazon Web Services (AWS) es una colección de servicios de computación en la nube pública, ofrecidas a través de Internet por Amazon.com.

El contenido sobre estos servicios de AWS se encuentra en línea[1], donde se describe muy detalladamente AWS Lambda. Éste es un servicio informático sin servidor que ejecuta código como

[1] https://aws.amazon.com/es/lambda/features/

respuesta a eventos y administra automáticamente los recursos informáticos subyacentes. Permite ampliar otros servicios con lógica personalizada o crear sus propios servicios backend que funcionen a escala. AWS Lambda ejecuta de forma automática el código en respuesta a varios eventos, como las solicitudes HTTP a través de Amazon API Gateway, las modificaciones de objetos en los buckets de Amazon Simple Storage Service (Amazon S3), las actualizaciones de tablas en Amazon DynamoDB y las transiciones de estado en AWS Step Functions.

Lambda, entre otras cosas, se encarga del mantenimiento del servidor y del sistema operativo. AWS Lambda permite agregar buckets de Amazon S3 y tablas de Amazon DynamoDB, lo que permite aplicar fácilmente la informática a los datos a medida que entran o transitan por la nube.

Lambda procesa eventos personalizados en lugar de atenderlos del lado del cliente, lo que le ayuda a evitar las variaciones de la plataforma del cliente, reducir el consumo de batería y permitir actualizaciones más sencillas.

También puede empaquetar cualquier código (marcos, SDK, bibliotecas, etc.) como una capa de Lambda para administrarlo y compartirlo fácilmente a través de múltiples funciones. Lambda es compatible de forma nativa con código Java, Go, PowerShell, Node.js, C#, Python y Ruby- Proporciona una API de tiempo de ejecución que le permite utilizar cualquier lenguaje de programación adicional para crear sus funciones.

AWS Lambda se encarga de todas las tareas relacionadas con la administración, el mantenimiento y los parches de seguridad. Ofrece monitoreo y registros integrados a través de Amazon CloudWatch.

2.5 CloudWatch

Los servicios en la nube de Amazon Web Services, como Amazon CloudWatch, se ocupan de monitorear los datos de registro. Mediante estos servicios, se puede recopilar y agregar datos de diversas fuentes para su análisis, visualización y alertas. CloudWatch Logs brinda la salida de los contenedores en los que se ejecuta el código. Para ver este resultado, se deben hacer algunos ajustes al código y luego usar CloudWatch Logs.

Para acceder a Amazon CloudWatch Logs y ver los registros mediante la consola de Lambda, se deben seguir los siguientes pasos:

1. Abrir Functions de la consola Lambda.

2. Elegir la función.

3. Seleccionar Monitoring.

4. Ir a logs en CloudWatch

La figura 3 muestra la arquitectura de cómo funciona CloudWatch.

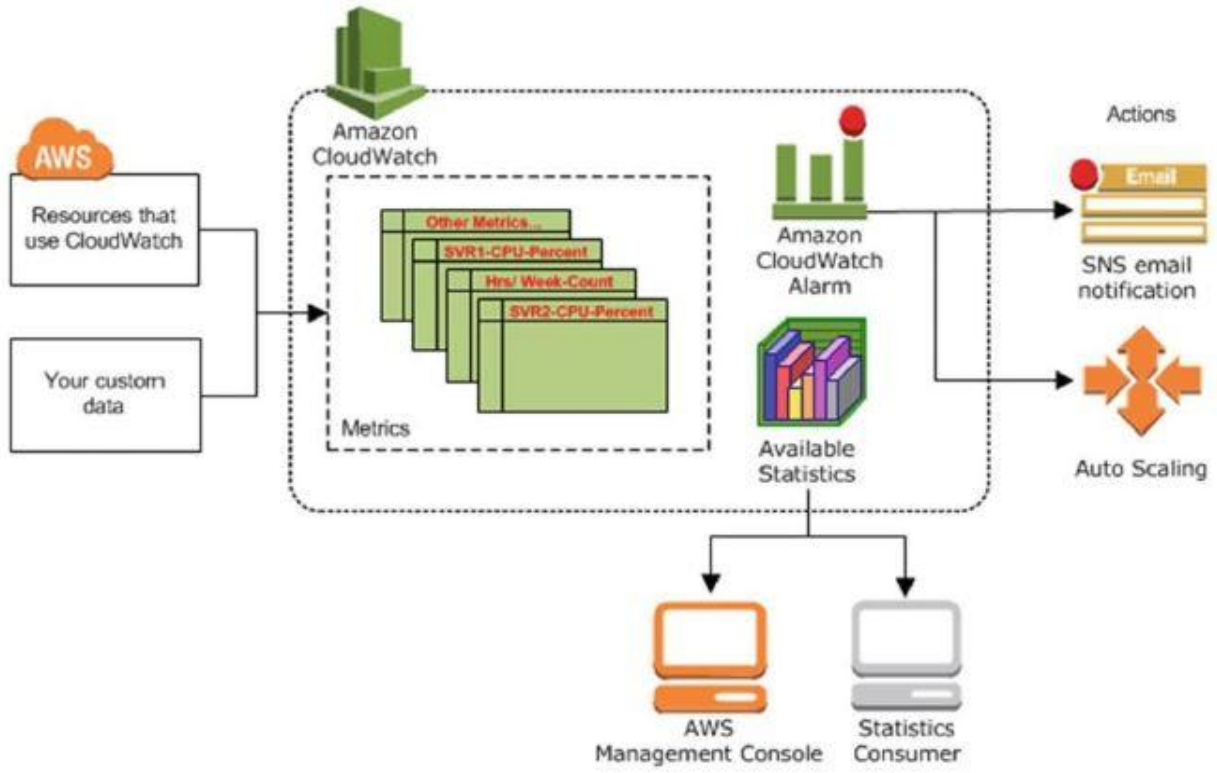


Figura 3. Arquitectura de CloudWatch. Recuperado de (Singh, 2021)

2.6 CloudWatch Metrics

CloudWatch Metrics proporciona valores sobre la exactitud de un modelo, la precisión, el error, etc. También puede suministrar métricas relacionadas con los recursos, tales como la utilización de la GPU, la utilización de la memoria, etc.

La figura 3 muestra que además de acceder a los servicios a través de Amazon Management Console, se puede integrar alarmas a través de Amazon SNS y un correo electrónico. Además es factible establecer reglas personalizadas basadas en ciertos criterios (iniciar, detener y finalizar un proceso) o usar funciones como el escalado automático (Singh, 2021).

2.7 Log Insight

Cloudwatch Insights es una herramienta que permite escribir y ejecutar consultas en grupos de registro seleccionados utilizando un lenguaje similar a SQL y extraer información relevante.

Una sola consulta contiene uno o más comandos que se separan mediante el símbolo "|". Entre los comandos de consulta están: campos, filtros, análisis, estadísticas, visualización, etc. Además se admiten varias funciones y operaciones, incluidas funciones de fecha y hora, funciones de cadena, funciones numéricas y expresiones regulares.

Si un evento de registro contiene un objeto JSON como parte del mensaje de registro, Cloudwatch Insights extrae automáticamente los campos del objeto JSON para luego hacer referencia en la consulta (Eroshenko, I. 2021).

2.8 Bases de Datos NoSQL - DB NoSQL -

Hoy en día existen problemas los cuales no pueden solucionarse de forma viable a través de las bases de datos tradicionales como son las SQL. Estas dificultades se deben en su mayoría, a la manipulación de grandes cantidades de información. Debido a lo mencionado anteriormente se hizo necesario la búsqueda de distintas formas para poder tratar con el manejo de la información dando finalmente con un modelo de base de datos más flexible llamado NoSQL.

Las bases de datos NoSQL son sistemas con el fin de almacenar información, la cual no se corresponde con el modelo entidad-relación. Algunas de las principales características de estas BDs son que: poseen una gran escalabilidad, utilizan modelos de consistencia relajados centrándose en lograr un mayor performance y disponibilidad de datos y no tienen un lenguaje de consulta declarativo por lo que se requiere de programación adicional para la manipulación de estos. Cuando se trabaja con enormes cantidades de información, los sistemas NoSQL trabajan mejor y más rápidos que los sistemas de base de datos tradicionales. De igual manera estos sistemas tradicionales son la solución para una gran variedad de aplicaciones (Martín et al., 2013).

Recientemente se han diseñado nuevos sistemas que proveen buena escalabilidad horizontal para las operaciones simples de escritura y lectura de las bases de datos distribuidas en muchos servidores. En comparación con estas, las bases de datos tradicionales tienen poca o ninguna capacidad de escalar horizontalmente.

Existen seis características específicas que permiten agrupar a las bases de datos NoSQL. Estas son (Cattell, 2011):

- Habilidad para escalar horizontalmente en muchos servidores.
- Habilidad para replicar y distribuir datos sobre varios servidores.
- Una interfaz o protocolo de consultas más simples, en contraste con SQL.
- Modelo de concurrencia más débil que el ACID.
- Uso eficiente de índices distribuidos y RAM para almacenamiento de datos.
- Habilidad para agregar nuevos atributos de forma dinámica a los registros de datos.

Aunque existen varías tecnologías cuando se habla de base de datos NoSQL, se pueden distinguir cuatro:

- Almacenamiento clave-valor.
- Almacenamiento de grafos.
- Almacenamiento de documentos.
- Almacenamiento de familia de columnas.

2.8.1 NoSQL vs. SQL

En la figura 4 se pueden observar las principales diferencias que existen entre las bases de datos SQL y NoSQL. (MUS, 2019)

Figura 4. Comparación de características de base de datos NoSQL y SQL

2.9 Operaciones CRUD: Create–Read–Update–Delete

El término CRUD hace referencia a las operaciones básicas que se pueden realizar sobre un conjunto de datos. Sus siglas son por:

Create (Crear); nuevos registros, insertar información.

Read (Leer); consultar la información, ya sea un registro o una colección de registros.

Update (Actualizar), que significa tomar un registro que ya existe en la base de datos y modificar alguna de las columnas.

Delete (Eliminar) registros, es decir tomar un registro y quitarlo del almacén.

Muchos autores consideran a las operaciones CRUD como las principales para la gestión de datos en aplicaciones de Base de Datos, pues ofrecen un punto de partida para diseñadores y desarrolladores a la hora de elegir un Sistema Gestor de Bases de Datos y son las que distribuyen la carga entre los diferentes puntos del sistema de información, tanto localmente como remotamente (Becerra Urbina, 2019; Castillo et al., 2017).

Si bien el objetivo de trabajar con estas aplicaciones es almacenar, organizar y clasificar los datos, también sirven para corregir errores en la manipulación de la información.

Como se mencionó en el capítulo anterior estas operaciones son base para muchos software de aplicación empresarial, tales como sistemas de ventas, de recepción de pedidos, software para hoteles, colegios, de seguridad, entre otros. Por ejemplo, en un sistema de ventas se tiene al cliente como un recurso y éste puede ser manipulado a través de su CRUD.

De pretender incorporar un nuevo cliente en el sistema mencionado, se estaría realizando un insert (create) en una base de datos y se estaría agregando un nuevo registro en ella. Esto es, se inserta el nombre de cada uno de los clientes, correo electrónico, documento de identidad y demás datos correspondientes a estos. Un usuario administrativo del sistema debe tener la posibilidad de visualizar un listado de clientes y esta acción se llevaría a cabo mediante un select (read). Por otro lado se ejecuta un update cuando se quiere modificar la información correspondiente de algún cliente, como por ejemplo actualizar el domicilio de uno de estos. De igual modo, si se quiere eliminar un registro de un cliente de la base de datos, se realiza un delete con lo cual se borra la información pertinente a éste.

2.10 Amazon DynamoDB

En el sitio oficial de Amazon, se expresa que Amazon DynamoDB es un servicio de DB NoSQL totalmente administrado que ofrece un rendimiento rápido y predecible, así como una perfecta escalabilidad. Este servicio permite transferir las cargas administrativas que supone tener que utilizar y escalar bases de datos distribuidas, para que no haya que preocuparse del aprovisionamiento, de la configuración del hardware, ni tampoco de las tareas de replicación (AWS, 2022).

Con DynamoDB se pueden crear tablas de DB capaces de almacenar y recuperar cualquier cantidad de datos, así como atender cualquier nivel de tráfico de solicitudes. También se puede usar la consola de administración de AWS (AWS Management Console) para supervisar la utilización de recursos y las métricas de rendimiento. Tiene la posibilidad de crear copias de seguridad bajo demanda y habilitar la recuperación en un momento dado.

La recuperación a un momento dado ayuda a proteger las tablas de operaciones accidentales de escritura o eliminación. Con la recuperación a un momento dado, se puede restaurar una tabla a cualquier momento de los últimos 35 días.

En el sitio web, también está disponible la información sobre cómo funciona, su configuración y otras funcionalidades más.

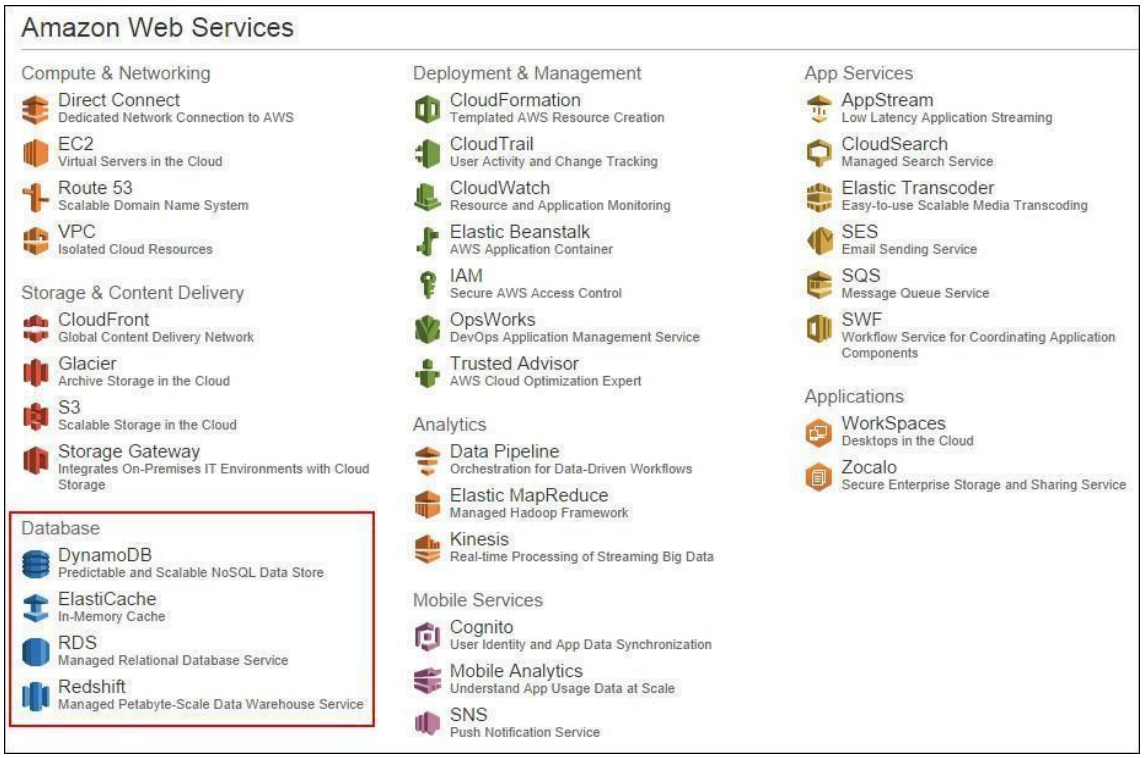


Figura 5. Consola de administración de bases de datos de AWS. Recuperada de (Pulipaka, 2016).

El acceso inicial a Amazon DynamoDB se puede obtener a través de la sección de la consola de la base de datos de AWS, ver figura 5. La sección de inicio de la pantalla de Amazon DynamoDB estará visible al hacer clic en el flujo de navegación en la consola.

2.11 Lenguajes de programación

Si bien la elección de los lenguajes depende de diferentes propósitos y según la demanda laboral, para muchos Python es el lenguaje principal, seguido de Java, JavaScript, C++ y otros. Lo anterior se refleja en la figura 6.

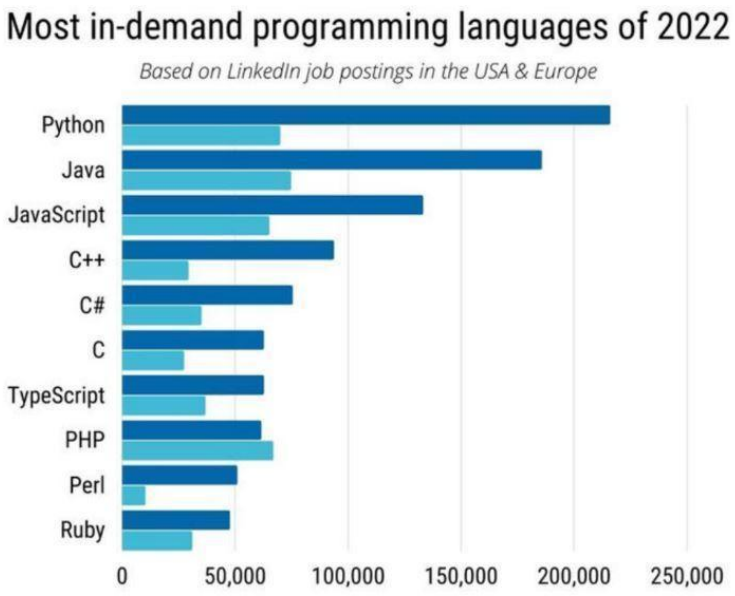


Figura 6. Los mejores lenguajes de programación, según demanda, 2022. Recuperada de (TechGuy, 2022).

Los top en los lenguajes ya viene siendo similar desde hace unos años, lo que se puede apreciar en la figura 7.

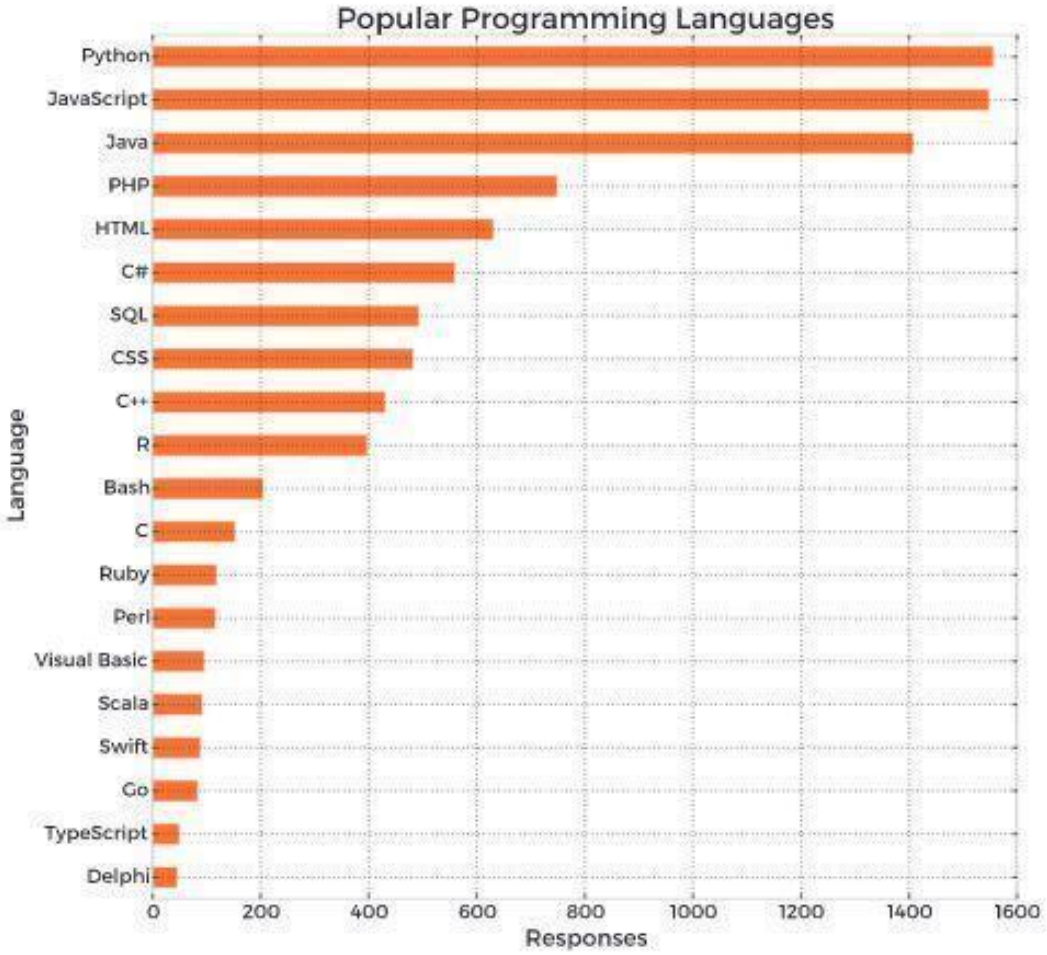


Figura 7. Los lenguajes de programación más populares en el 2016. Recuperada de (Cimpanu, 2016).

La figura 7 data del año 2016 y muestra el estudio realizado por Packt Publishing, una de las mayores editoriales del mundo de libros relacionados con la programación. En esa oportunidad, la encuesta reveló el favoritismo hacia Python y JavaScript.

2.11.1 Java, Javascript y Python

Java es un buen lenguaje como punto de partida, porque se adquiere buenos hábitos de programación que luego son útiles. Su sintaxis es orientada a objetos, lo que permite a los desarrolladores de aplicaciones escribir el programa de una sola vez y ejecutarlo en cualquier dispositivo. Es un lenguaje de arquitectura neutral, robusto, seguro y dinámico; entre otras características (Abiola, 2017; Froufe, 1996).

JavaScript se creó inicialmente para la web, pero es una excelente opción para desarrollar aplicaciones nativas multiplataforma. La mayoría de las aplicaciones web se construyen completamente en JavaScript, por lo que se ha convertido en un lenguaje de programación muy popular. Debido a su simplicidad y velocidad, muchas empresas están comenzando a usar JavaScript en el backend a través del entorno de ejecución NodeJS. NodeJS tiene una arquitectura orientada a eventos la cual permite gestionar una gran cantidad de estos de forma asíncrona, debido a que se ejecutan de forma independiente y sin interferir unos con otros. En vez de generar un hilo

por cada cliente, se genera un evento por cada petición y esta se gestiona de manera independiente y sin bloqueos. Esta es la principal característica que le permite tener un gran rendimiento, especialmente en proyectos de gran envergadura. Otra característica importante es que está basado en el motor v8 de google, el cual es un intérprete que se encarga de compilar el código javascript en código de máquina y no hace falta que sea interpretado por el navegador, lo cual brinda mayor velocidad y rendimiento (w3schools, s.f.).

Según Stack Overflow, sitio de preguntas y respuestas para programadores y profesionales de la informática, Python se fue convirtiendo en el lenguaje de elección más popular. Es un lenguaje de alto nivel y de propósito general. Que sea de propósito general quiere decir que puede ser utilizado para el desarrollo de todo tipo de aplicaciones. Es un lenguaje interpretado, dinámico y multiplataforma que tiene mucho foco en la legibilidad de su código, por esta razón es recomendado como uno de los lenguajes más fáciles para aprender a programar. En los últimos años Python ha ganado mucha popularidad ya que facilita el desarrollo con inteligencia artificial, big data, machine learning y data science, entre muchos otros campos en auge (Yoan, s.f).

2.12 Test de Carga

Actualmente las APIs, interfaz de programación de aplicaciones, son las bases de toda empresa de software. Estas reciben una carga de tráfico proveniente de distintos lugares, por lo que esta carga es variable y puede ser difícil de predecir. Debido a esta situación, se necesita la forma de disminuir los riesgos y asegurarse de que una API sea capaz de soportar una cierta cantidad de tráfico sin afectar su rendimiento y sus tiempos de respuesta.

Los tests de cargas o pruebas de carga son ideales para determinar si una API cumple con el objetivo de poder servir una cierta cantidad de solicitudes sin perder performance. Estos tests nos permiten simular estas cargas de tráfico hacia una API en desarrollo y poder detectar posibles cuellos de botella antes de alcanzar un ambiente de producción.

Para estos tests existen diversas herramientas de código abierto que se pueden utilizar. Un desafío común de estas herramientas tiene que ver con la creación de flujos de trabajo, por ejemplo, una solicitud puede requerir una autenticación y luego una autorización (Schulz et al., 2020).

2.13 Artillery

Artillery, es un kit de herramientas moderno que permite testear de manera muy fácil, la performance de aplicaciones. Permite realizar dos tipos de tests de performance:

- Tests que ponen carga sobre un sistema, por ejemplo test de carga, de estrés, de humo y de remojo.
- Tests que verifican que el sistema esté trabajando como es esperado, por ejemplo pruebas funcionales continuas, también conocidas por otros nombres, como: monitoreo sintético, monitoreo semántico, pruebas con guión de producción y verificación continua.

Cuando Artillery ejecuta un test, realiza un seguimiento y registro de métricas de rendimiento, tales como tiempos de respuesta, throughput y errores. Estas métricas se reportan en tiempo real y también se brinda un resumen al finalizar la ejecución del test.

Artillery se basa en el concepto de usuarios virtuales. Un usuario virtual es la simulación de un usuario real o cliente que se encuentra usando un servicio o API. Estos usuarios virtuales son totalmente independientes unos con otros, es decir, por ejemplo cuando se prueba un servicio basado en HTTP cada usuario va a abrir y mantener sus propias conexiones TCP así también como sus cookies y cualquier otra información (Artillery, s.f.).

2.14 Métricas

En el sitio web de Amazon CloudWatch, la sección Types of metrics describe los tipos de métricas disponibles en su consola. Las métricas de AWS a utilizar son las siguientes:

- **Ejecuciones Concurrentes**: Indica el número de instancias, contenedores levantados, de la función que están procesando eventos. Si este número supera el límite establecido en la región o el límite de simultaneidad reservado configurado en la función, Lambda limita las solicitudes de invocación adicionales.
- **Limitaciones**: Indica el número de solicitudes de invocación que se han limitado. Cuando todas las instancias de funciones están procesando solicitudes y no hay concurrencia disponible para escalar, Lambda rechaza solicitudes adicionales con un error TooManyRequestsException. Las solicitudes limitadas y otros errores de invocación no cuentan como invocaciones o errores.
- **Invocaciones**: Es el número de veces que se invoca el código de la función, incluidas las invocaciones correctas y las que dan como resultado un error de función. Las invocaciones no se registran si la solicitud de invocación se limita o genera un error de invocación. Esto es igual al número de solicitudes facturadas.
- **Duration**: Indica la cantidad de tiempo que consume una función a la hora de procesar un evento. La duración facturada para una invocación es el valor de Duration redondeado al milisegundo más próximo.
- **Consumo de memoria**: Indica la cantidad de memoria utilizada o consumida por la función.
- **Tiempo de inicialización** (Cold Start): Indica la cantidad de tiempo que demoró el servicio Lambda en inicializar el ambiente de ejecución de la función. Si el ambiente es reutilizado en una invocación siguiente no se debe levantar nuevamente y por ende el tiempo de inicialización no existirá para esa invocación.
- **Costos**: Este indicador se puede obtener en base a la duración promedio de las invocaciones realizadas en cada lenguaje durante los tests, suponiendo un mismo número de invocaciones por cada lenguaje.
- **Unidades de capacidad de lectura consumida:** Cantidad de unidades de capacidad de lectura usadas durante el periodo de tiempo especificado.

- **Unidades de capacidad de escritura consumida**: Cantidad de unidades de capacidad de escritura consumidas durante el periodo de tiempo especificado, para saber cuánto rendimiento aprovisionado es utilizado
- **Latencia de solicitud exitosa**: Puede proporcionar dos tipos distintos de información:
    - El tiempo transcurrido para las solicitudes correctas.
    - El número de solicitudes realizadas correctamente.

CAPÍTULO 3

Procedimiento Aplicado

3.1 Metodología

Para llevar a cabo los estudios de laboratorio pertinentes sobre diversas variables en la ejecución y puesta en marcha de funciones serverless (FaaS), se realizaron las siguientes actividades:

- Análisis y estudio de la plataforma o servicio AWS Lambda.
- Estudio y funcionamiento detallado de las bases de datos disponibles en AWS.
- Determinación de los lenguajes a comparar haciendo uso de funciones.
- Determinación de los escenarios a utilizar que permitan contrastar los valores resultantes.
- Estudio comparativo de la eficiencia y otros parámetros como el uso de la memoria y los costos.

Según Keith Stanovich, citado en un artículo de la Universidad Iberoamericana Tijuana, la investigación aplicada busca "predecir el comportamiento específico". Además, *"el propósito de la investigación aplicada, en síntesis, es resolver determinados problemas o planteamientos, centrándose en el estudio y la consolidación del conocimiento para aplicarlo en la vida real"* (Ibaro, 2020).

El tipo de investigación es aplicada o práctica y se empleará una metodología cuantitativa en el marco de una investigación experimental, que permita comparar los valores en los distintos escenarios.

3.2 Problemática

La problemática abordada es la evaluación del comportamiento de los lenguajes de programación en una plataforma serverless, tomando como caso de estudio a Amazon Web Services.

En este capítulo se describe cada uno de los pasos realizados, con el fin de que se comprenda la base sobre la cual se realizó todo el proceso de investigación y desarrollo.

Para el desarrollo de la investigación se procedió como se muestra en la figura 8, la cual consiste en un esquema que sintetiza la labor realizada.

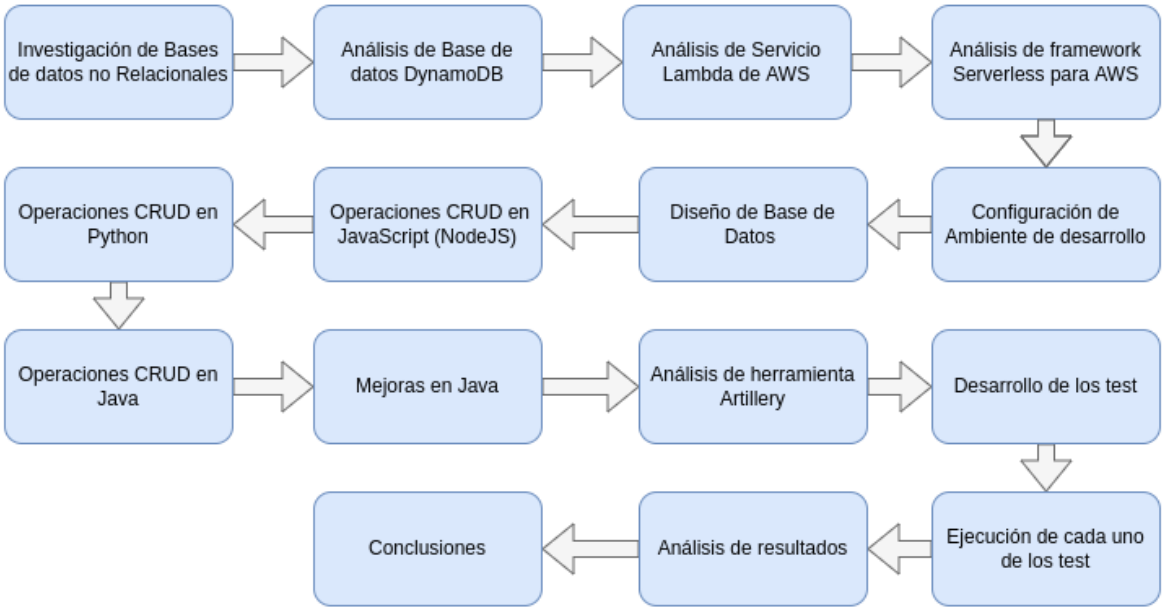


Figura 8. Esquema del procedimiento de la investigación

Para la investigación se tomó la decisión de desarrollar una API de formularios, utilizando los servicios de AWS que permiten un desarrollo serverless. Se consideró una API que permita el manejo de distintos formularios dinámicos y que, a su vez, conserve la información pertinente a estos junto con sus respuestas. Lo anterior se pensó adecuado para reproducir un problema del mundo real en el que se necesite de recursos de software escalables cuyo objetivos sean los de soportar grandes volúmenes de usuarios sin afectar el rendimiento del sistema.

3.3 Representación del formulario

Se dispuso realizar una API de formularios, como se mencionó en los objetivos, para la comparativa de los distintos lenguajes de programación en un entorno serverless.

La función de esta API es la de crear formularios dinámicos y almacenar toda la información requerida por éstos, incluyendo las respuestas de los usuarios. Para realizar la evaluación de los lenguajes bajo este contexto, solamente necesitaremos un formulario.

Para la evaluación se implementará un extracto del formulario utilizado para el Censo Nacional de Población, Hogares y Viviendas 2022 (Censo 2022). Dicho formulario puede ser descargado de la página oficial del Instituto Nacional de Estadística y Censos (Instituto Nacional de Estadística y Censos, 2022).
Los fragmentos utilizados se pueden ver en las figuras 9 a 12.

Figura 9. Primera sección correspondiente al Censo 2022, rótulo vivienda

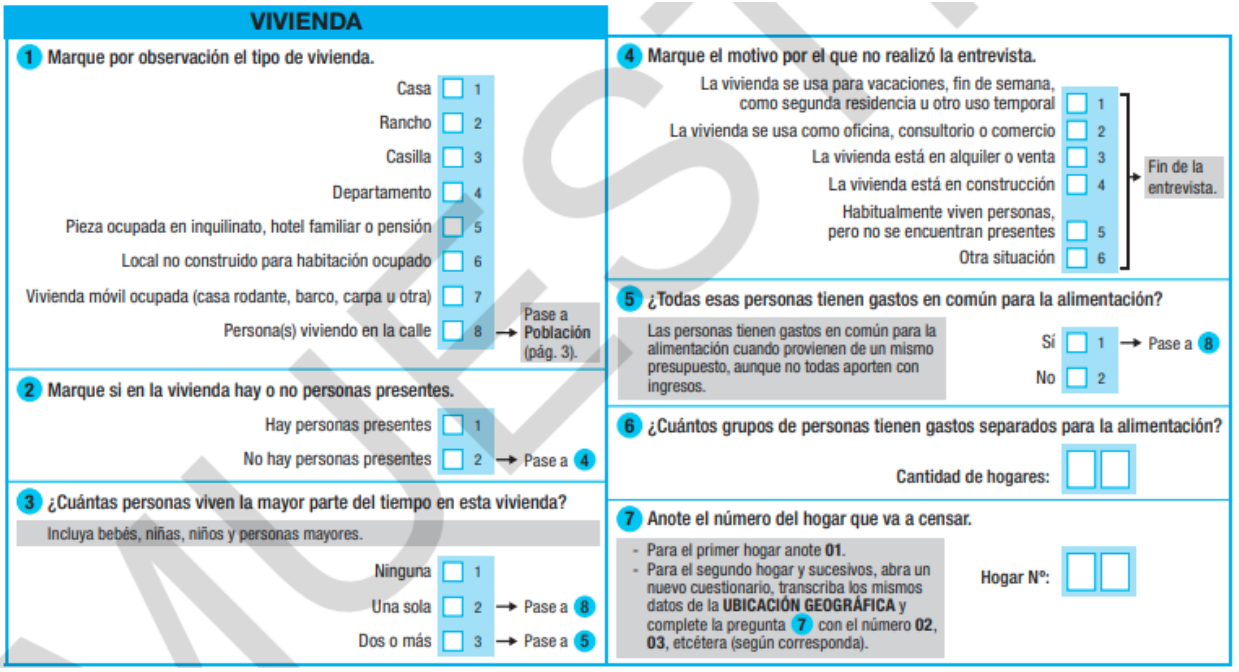
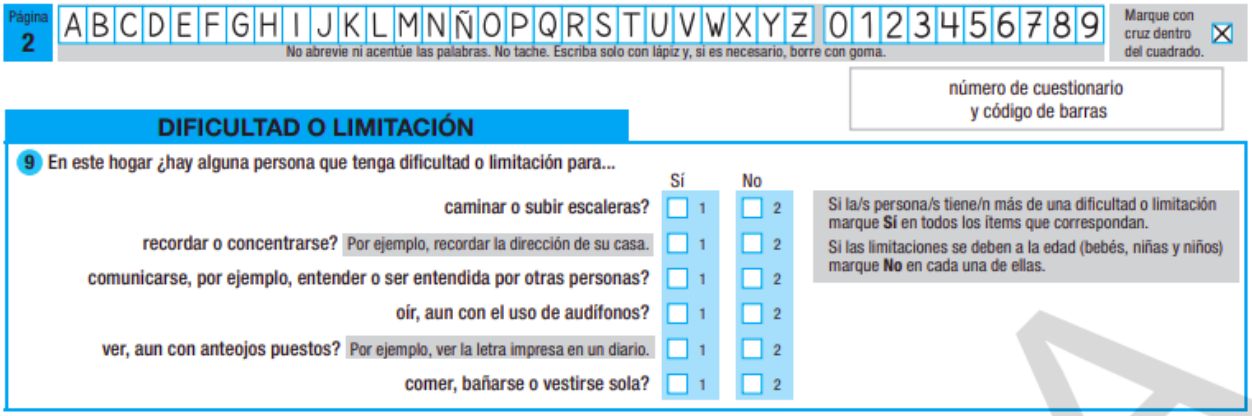


Figura 10. Segunda sección correspondiente al Censo 2022, rótulo vivienda



Página 2

A B C D E F G H I J K L M N Ñ O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9

No abrevie ni acentúe las palabras. No tache. Escriba solo con lápiz y, si es necesario, borre con goma.

Marque con cruz dentro del cuadrado. ☒

número de cuestionario y código de barras

DIFICULTAD O LIMITACIÓN

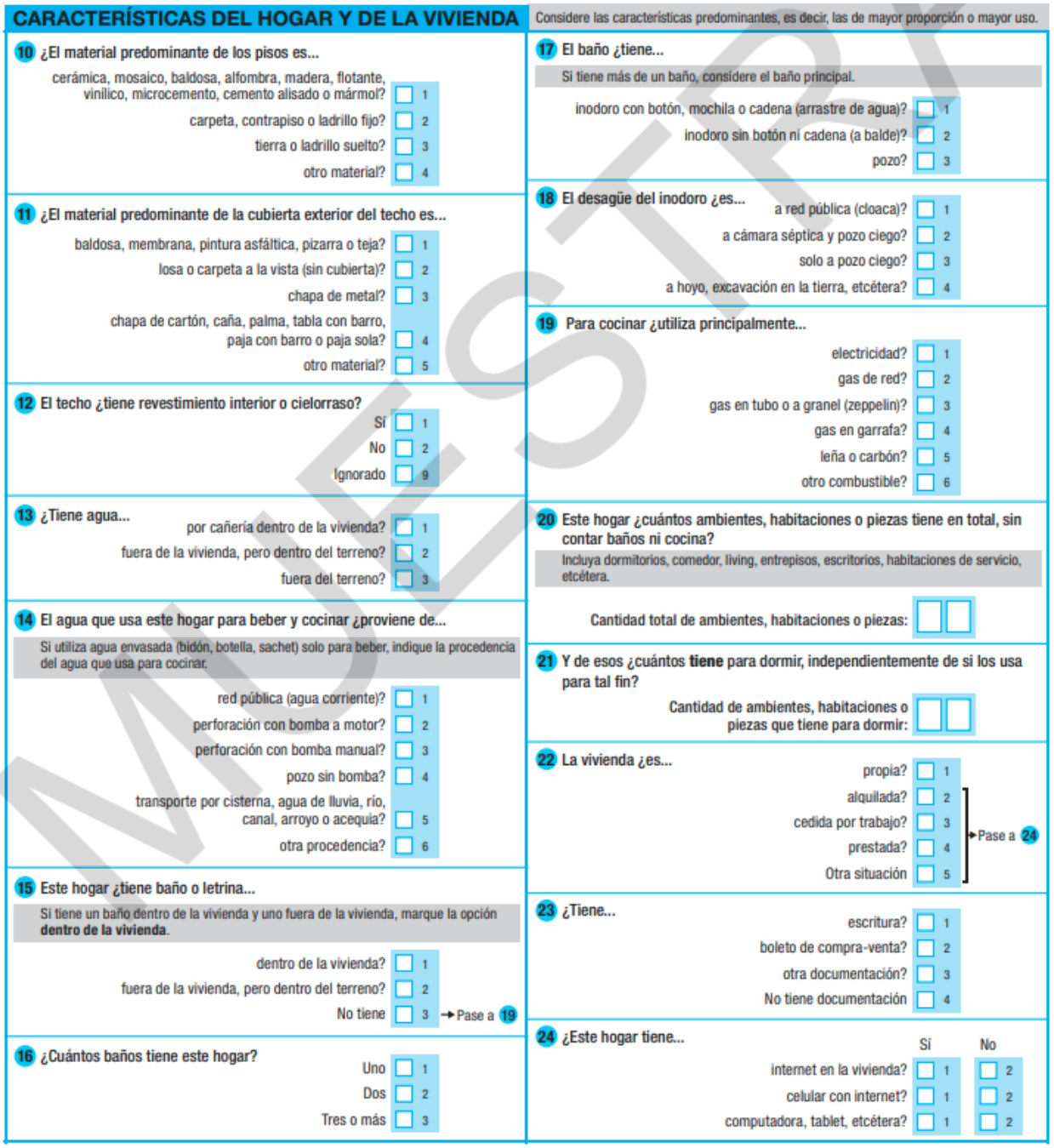
9 En este hogar ¿hay alguna persona que tenga dificultad o limitación para...

	Sí	No
caminar o subir escaleras?	1	2
recordar o concentrarse? Por ejemplo, recordar la dirección de su casa.	1	2
comunicarse, por ejemplo, entender o ser entendida por otras personas?	1	2
oír, aun con el uso de audífonos?	1	2
ver, aun con anteojos puestos? Por ejemplo, ver la letra impresa en un diario.	1	2
comer, bañarse o vestirse sola?	1	2

Si la/s persona/s tiene/n más de una dificultad o limitación marque **Sí** en todos los ítems que correspondan.

Si las limitaciones se deben a la edad (bebés, niñas y niños) marque **No** en cada una de ellas.

Figura 11. Fragmento "Dificultad o Limitación" de la persona censada, Censo 2022



CARACTERÍSTICAS DEL HOGAR Y DE LA VIVIENDA

Considere las características predominantes, es decir, las de mayor proporción o mayor uso.

10 ¿El material predominante de los pisos es...

- cerámica, mosaico, baldosa, alfombra, madera, flotante, vinílico, microcemento, cemento alisado o mármol? 1
- carpeta, contrapiso o ladrillo fijo? 2
- tierra o ladrillo suelto? 3
- otro material? 4

11 ¿El material predominante de la cubierta exterior del techo es...

- baldosa, membrana, pintura asfáltica, pizarra o teja? 1
- losa o carpeta a la vista (sin cubierta)? 2
- chapa de metal? 3
- chapa de cartón, caña, palma, tabla con barro, paja con barro o paja sola? 4
- otro material? 5

12 El techo ¿tiene revestimiento interior o cielorraso?

- Sí 1
- No 2
- Ignorado 9

13 ¿Tiene agua...

- por cañería dentro de la vivienda? 1
- fuera de la vivienda, pero dentro del terreno? 2
- fuera del terreno? 3

14 El agua que usa este hogar para beber y cocinar ¿proviene de...

Si utiliza agua envasada (bidón, botella, sachet) solo para beber, indique la procedencia del agua que usa para cocinar.

- red pública (agua corriente)? 1
- perforación con bomba a motor? 2
- perforación con bomba manual? 3
- pozo sin bomba? 4
- transporte por cisterna, agua de lluvia, río, canal, arroyo o acequia? 5
- otra procedencia? 6

15 Este hogar ¿tiene baño o letrina...

Si tiene un baño dentro de la vivienda y uno fuera de la vivienda, marque la opción **dentro de la vivienda**.

- dentro de la vivienda? 1
- fuera de la vivienda, pero dentro del terreno? 2
- No tiene 3 → Pase a 19

16 ¿Cuántos baños tiene este hogar?

- Uno 1
- Dos 2
- Tres o más 3

17 El baño ¿tiene...

Si tiene más de un baño, considere el baño principal.

- inodoro con botón, mochila o cadena (arrastre de agua)? 1
- inodoro sin botón ni cadena (a balde)? 2
- pozo? 3

18 El desagüe del inodoro ¿es...

- a red pública (cloaca)? 1
- a cámara séptica y pozo ciego? 2
- solo a pozo ciego? 3
- a hoyo, excavación en la tierra, etcétera? 4

19 Para cocinar ¿utiliza principalmente...

- electricidad? 1
- gas de red? 2
- gas en tubo o a granel (zeppelin)? 3
- gas en garrafa? 4
- leña o carbón? 5
- otro combustible? 6

20 Este hogar ¿cuántos ambientes, habitaciones o piezas tiene en total, sin contar baños ni cocina?

Incluya dormitorios, comedor, living, entrepisos, escritorios, habitaciones de servicio, etcétera.

Cantidad total de ambientes, habitaciones o piezas: ☐☐

21 Y de esos ¿cuántos **tiene** para dormir, independientemente de si los usa para tal fin?

Cantidad de ambientes, habitaciones o piezas que tiene para dormir: ☐☐

22 La vivienda ¿es...

- propia? 1
- alquilada? 2
- cedida por trabajo? 3
- prestada? 4
- Otra situación 5

→ Pase a 24

23 ¿Tiene...

- escritura? 1
- boleto de compra-venta? 2
- otra documentación? 3
- No tiene documentación 4

24 ¿Este hogar tiene...

	Sí	No
internet en la vivienda?	1	2
celular con internet?	1	2
computadora, tablet, etcétera?	1	2

Figura 12. Fragmento “Característica del hogar y de la vivienda”, Censo 2022

De esa síntesis del formulario nacional del censo, se omitieron algunas preguntas que no contribuyen al trabajo de investigación propuesto. Algunas de estas son:

- Información adicional sobre la dirección de vivienda.
- Motivo por el cual no se realizó la entrevista. Sección Vivienda pregunta 4.
- Número de hogar que se va a censar. Sección Vivienda pregunta 7.

3.4 Base de datos

Con la finalidad de almacenar los datos necesarios para la representación de formularios y sus respuestas, se optó por utilizar el servicio DynamoDB de AWS ya que permite realizar la investigación bajo el enfoque serverless debido a que este servicio es totalmente administrado y evita las cargas administrativas que supone tener que utilizar y escalar bases de datos. También este servicio dispone de la potencia necesaria para crear tablas que permiten almacenar y recuperar miles y hasta millones de datos y solicitudes.

En esta investigación se propone imitar de la mejor forma al tráfico de un aplicativo real y de uso constante, por lo que este servicio será fundamental para poder replicar el contexto deseado en el cual se evaluará a los lenguajes de programación elegidos.

3.5 Diseño de la Base de datos

En la figura 13 se encuentra el diseño final resultante que se implementó con el servicio DynamoDB y se utilizó para la evaluación en este trabajo.

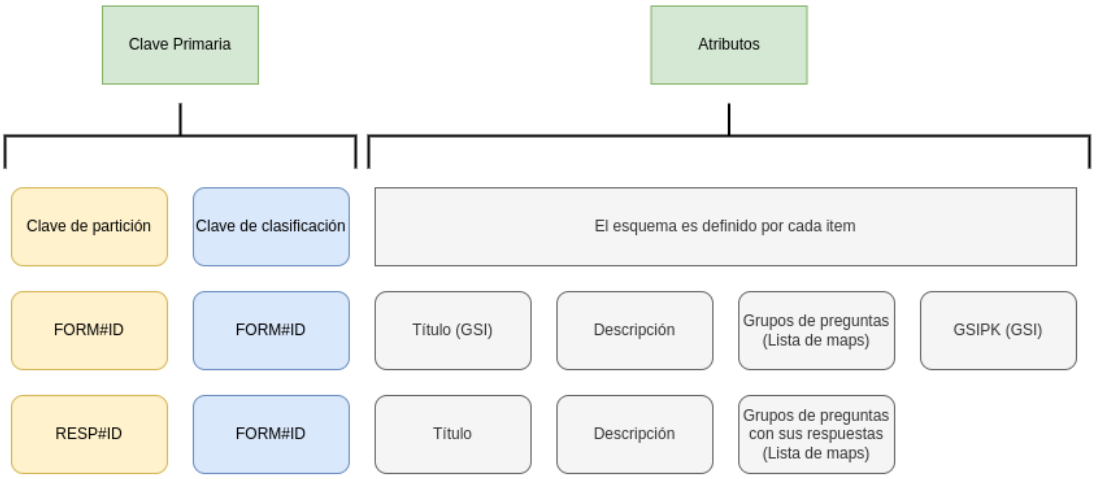


Figura 13. Diseño de la base de datos implementada en DynamoDB para la evaluación

La estructura de la base de datos está basada en el patrón de diseño llamado **single table**. Este patrón consiste en utilizar una sola tabla de la base de datos para toda la aplicación, almacenando distintos tipos de entidades en la misma tabla. La ventaja de este diseño es que permite un mayor rendimiento al reducir el número de solicitudes necesarias para recuperar la información. Este tipo de tabla también simplifica los cambios y la evolución de los diseños al desacoplar campos y atributos clave de entidad en la estructura de la tabla física.

El diseño de la base de datos que se implementó consiste de una clave primaría compuesta por una clave de partición (PK, partition key) que va a permitir identificar los tipos de entidades en la aplicación. Estas entidades son formularios y respuestas. El otro componente de la clave primaría es la clave de clasificación (SK, sort key) que permite identificar, para las entidades de respuesta, a qué formulario corresponde cada respuesta. Uno de los atributos a destacar es aquel que se puede observar con el nombre **GSIPK** en la figura 13. Este atributo es lo que se denomina como índice global secundario (GSI, global secondary index) y permite en este contexto realizar consultas para obtener aquellas entidades (formularios) pero solamente con aquellos atributos que se consideran necesarios; en este caso el título de cada formulario.

En caso de querer obtener todos los formularios sin este índice, obtendremos información innecesaria que puede afectar el rendimiento de las solicitudes.

Entre los demás atributos se puede encontrar el título de un formulario el cual también es un índice global secundario que permite obtener los formularios a través de sus títulos junto con sus descripciones y unas listas de colecciones donde se guardan las preguntas con su información. En el caso de ser una entidad respuesta, esta lista de preguntas tendrá también el valor de la respuesta a la pregunta. Puede verse que en la respuesta hay datos redundantes del formulario correspondiente, esto se debe a la redundancia de datos que facilita el acceso a la información sin tener que hacer solicitudes adicionales a la base de datos.

3.6 Ambiente de desarrollo

Para el desarrollo de las APIs de formularios en los distintos lenguajes sobre la plataforma de AWS, se hizo uso de varias herramientas que permiten y facilitan este desarrollo. Estas son:

- Github
- Serverless Framework
- AWS CLI
- NodeJS y NPM
- Python y PIP
- Maven
- Java SDK

Algunas de las mencionadas anteriormente son dependencias necesarias para el desarrollo de las APIs dependiendo de cada lenguaje de programación. Por ejemplo para JavaScript con NodeJS se necesita instalar NodeJS y NPM, para Python se requiere de Python y PIP y por último para desarrollar con Java se necesita Maven y el SDK de Java.

El desarrollo de este trabajo fue realizado en una notebook personal bajo la distribución Ubuntu 20.04 del sistema operativo Linux.

3.6.1 Github

Antes de comenzar con la configuración y la codificación necesaria para implementar todos los recursos que se utilizaron en el trabajo, se tomó la decisión de utilizar una plataforma que brinde la posibilidad de controlar las versiones del trabajo y de alojarlas en la nube. Con este propósito se procedió a hacer uso de la plataforma **GitHub** (https://docs.github.com/es) que, como su nombre lo indica, hace uso del sistema de control de versiones **Git**.

Se creó un repositorio de código en Github para lo mencionado anteriormente. El repositorio se encuentra albergado en https://github.com/mrodriguezbegueri/tesis y es de público acceso para ser descargado y replicado.

Una vez que se creó el repositorio donde almacenar todo lo necesario para este trabajo, se implementó la estructura de carpetas, la cual se puede visualizar en la figura 14.

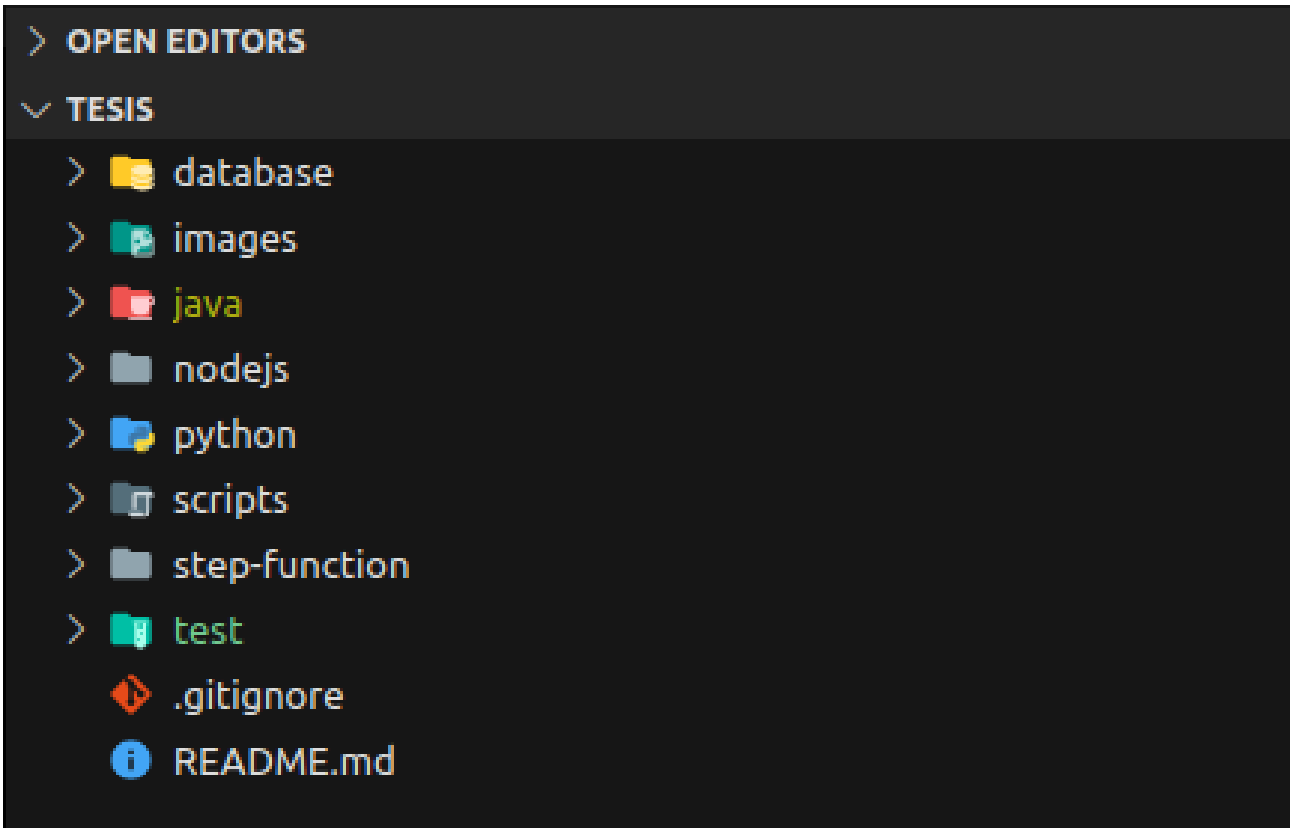


Figura 14. Estructura de carpetas a utilizar para la organización de los recursos necesarios

Esta estructura consta de las siguientes carpetas:

- **database**: Contiene todas las configuraciones necesarias para la implementar la base de datos DynamoDB en AWS.
- **images**: Esta carpeta se creó con el fin de almacenar las imágenes que se fueron tomando a medida que el trabajo avanzaba.
- **java/nodejs/python**: Cada una de estas carpetas dispone del código y las configuraciones requeridas para implementar las APIs de formularios en los distintos lenguajes de programación sobre la plataforma AWS.
- **step-function**: En esta carpeta se realizó una implementación de una step function de AWS, pero que luego no se necesitó para la evaluación de los lenguajes.
- **test**: Carpeta que almacena la configuración de los test realizados y los recursos necesarios para ejecutarlos por cada lenguaje de programación.

3.6.2 Serverless Framework

Una de las principales herramientas que se utilizaron en este trabajo fue Serverless. Este framework consiste de una interfaz de línea de comandos de código abierto que brinda una administración completa del ciclo de vida de las aplicaciones serverless. Mediante las funcionalidades de serverless, se procedió a implementar los archivos de configuración; tanto de la base de datos como de las funciones de las cuales cada una de las APIs de formularios, harán uso (Serverless Framework Documentation, s.f.).

Por cada API y la base de datos dispondremos de un archivo de configuración en el formato YAML. Este archivo será interpretado por el framework serverless, el cual implementará los recursos expuestos en estos archivos en una cuenta AWS configurada. YAML es un formato de serialización de datos. Entre sus ventajas está la legibilidad, la capacidad de escritura y admite varios tipos de datos como matrices, diccionarios, listas, escalares, etc. Además, tiene un buen soporte para los lenguajes más populares como JavaScript, Python, Rubí, Java, entre otros (Bigyan, 2021).

En la figura 15 se puede observar la configuración de la base de datos DynamoDB. La configuración en serverless del diseño de la base de datos DynamoDB a implementarse en AWS está dispuesta en la figura 16. Las figuras 17, 18 y 19 presentan la configuración de cada una de las APIs de formularios.

```yaml
service: forms-api-database

frameworkVersion: '2'

provider:
  name: aws
  runtime: nodejs12.x
  lambdaHashingVersion: 20201221
  region: us-east-1
  environment: ${file(config/serverless/environment/env.yml)}

resources:
  - ${file(config/serverless/database/database_config.yml)}
```

Figura 15. Configuración en serverless, para implementar la base de datos DynamoDB

```yaml
Resources:
  FormsTable:
    Type: "AWS::DynamoDB::Table"
    Properties:
      TableName: ${self:provider.environment.FORMS_TABLE_NAME}
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: "PK"
          AttributeType: "S"
        - AttributeName: "SK"
          AttributeType: "S"
        - AttributeName: "GSI1PK"
          AttributeType: "S"
        - AttributeName: "title"
          AttributeType: "S"
      KeySchema:
        - AttributeName: "PK"
          KeyType: "HASH"
        - AttributeName: "SK"
          KeyType: "RANGE"
      GlobalSecondaryIndexes:
        - IndexName: ListFormsTitles
          KeySchema:
            - AttributeName: "GSI1PK"
              KeyType: "HASH"
          Projection:
            NonKeyAttributes:
              - title
            ProjectionType: "INCLUDE"
        - IndexName: SearchFormByTitle
          KeySchema:
            - AttributeName: "title"
              KeyType: "HASH"
            - AttributeName: "PK"
              KeyType: "RANGE"
          Projection:
            ProjectionType: "ALL"
```

Figura 16. Configuración en serverless del diseño de la base de datos DynamoDB a implementar en AWS

```yaml
service: forms-api-nodejs

frameworkVersion: '2'

provider:
  name: aws
  runtime: nodejs14.x
  lambdaHashingVersion: 20201221
  region: us-east-1
  iamRoleStatements:
    - Effect: "Allow"
      Action:
        - dynamodb:Query
        - dynamodb:DescribeTable
        - dynamodb:Scan
        - dynamodb:GetItem
        - dynamodb:UpdateItem
        - dynamodb:DeleteItem
        - dynamodb:BatchWriteItem
        - dynamodb:PutItem
        - lambda:InvokeFunction
        - lambda:InvokeAsync
      Resource:
        - arn:aws:dynamodb:${self:provider.region}:*:table/${self:provider.environment.FORMS_TABLE_NAME}
        - arn:aws:dynamodb:${self:provider.region}:*:table/${self:provider.environment.FORMS_TABLE_NAME}/index/ListFormsTitles
        - arn:aws:dynamodb:${self:provider.region}:*:table/${self:provider.environment.FORMS_TABLE_NAME}/index/SearchFormByTitle
  environment: ${file(config/serverless/environment/env.yml)}

package:
  individually: true
  patterns:
    - '!./**'
    - ./node_modules/**

functions:
  - ${file(config/serverless/functions/api/forms_functions.yml)}
  - ${file(config/serverless/functions/api/results_functions.yml)}
```

Figura 17. Configuración de la API en JavaScript

```yaml
service: forms-api-python

provider:
  name: aws
  runtime: python3.8
  lambdaHashingVersion: 20201221
  region: us-east-1
  iamRoleStatements:
    - Effect: "Allow"
      Action:
        - dynamodb:Query
        - dynamodb:DescribeTable
        - dynamodb:Scan
        - dynamodb:GetItem
        - dynamodb:UpdateItem
        - dynamodb:DeleteItem
        - dynamodb:BatchWriteItem
        - dynamodb:PutItem
        - lambda:InvokeFunction
        - lambda:InvokeAsync
      Resource:
        - arn:aws:dynamodb:${self:provider.region}:*:table/${self:provider.environment.FORMS_TABLE_NAME}
        - arn:aws:dynamodb:${self:provider.region}:*:table/${self:provider.environment.FORMS_TABLE_NAME}/index/ListFormsTitles
        - arn:aws:dynamodb:${self:provider.region}:*:table/${self:provider.environment.FORMS_TABLE_NAME}/index/SearchFormByTitle
  environment: ${file(config/serverless/environment/env.yml)}

plugins:
  - serverless-python-requirements

custom:
  pythonRequirements:
    dockerizePip: non-linux
    zip: true
    slim: true
    strip: false
    useDownloadCache: false
    useStaticCache: false

package:
  individually: true
  patterns:
    - '!./**'

functions:
  - ${file(config/serverless/functions/api/forms_functions.yml)}
  - ${file(config/serverless/functions/api/results_functions.yml)}
```

Figura 18. Configuración de la API en Python

```yaml
service: forms-api-java

provider:
  name: aws
  runtime: java11
  lambdaHashingVersion: 20201221
  region: us-east-1
  iamRoleStatements:
    - Effect: "Allow"
      Action:
        - dynamodb:Query
        - dynamodb:DescribeTable
        - dynamodb:Scan
        - dynamodb:GetItem
        - dynamodb:UpdateItem
        - dynamodb:DeleteItem
        - dynamodb:BatchWriteItem
        - dynamodb:PutItem
        - lambda:InvokeFunction
        - lambda:InvokeAsync
      Resource:
        - arn:aws:dynamodb:${self:provider.region}:*:table/${self:provider.environment.FORMS_TABLE_NAME}
        - arn:aws:dynamodb:${self:provider.region}:*:table/${self:provider.environment.FORMS_TABLE_NAME}/index/ListFormsTitles
        - arn:aws:dynamodb:${self:provider.region}:*:table/${self:provider.environment.FORMS_TABLE_NAME}/index/SearchFormByTitle
  environment: ${file(config/serverless/environment/env.yml)}

package:
  individually: true
  artifact: target/${self:service}-${self:provider.stage}.jar

functions:
  - ${file(config/serverless/functions/api/forms_functions.yml)}
  - ${file(config/serverless/functions/api/results_functions.yml)}
```

Figura 19. Configuración de la API en Java

3.6.3 Configuración cuenta AWS

Como se mencionó a lo largo de este trabajo, se utilizó la plataforma de servicios web AWS. Para hacer uso de estos servicios, se necesita configurar una cuenta en https://aws.amazon.com/es/ que nos otorgue acceso hacia la consola o credenciales que nos permitan interactuar con estos servicios. Hay varias formas de crear una cuenta y se consideraron dos de estas:

- **Cuenta gratuita**: AWS dispone de un programa llamado AWS Educate, que permite a estudiantes de cualquier edad, acceder a una cuenta totalmente gratuita y con distintos cursos a seguir para poder capacitarse en sus servicios de la nube.
- **Cuenta personal**: Esta cuenta se puede crear de forma muy sencilla ingresando a la web de AWS y completando la información requerida por la plataforma. Es importante destacar que se necesitará sí o sí de una tarjeta de crédito para el registro.

En este trabajo se usó una cuenta personal, dado que podía contar con la posibilidad de trabajar con una de estas cuentas. Esto permitió el uso de los servicios necesarios para la investigación.

3.6.4 Configuración AWS CLI

Una vez creada la cuenta de AWS se procedió a la descarga y configuración de la interfaz de línea de comandos de AWS (CLI, Command Line Interface). CLI es una herramienta unificada que nos permite administrar los servicios de AWS. Con esta herramienta podemos crear, eliminar, actualizar e interactuar con casi todos los recursos de la plataforma AWS, simplemente mediante el uso de comandos en una consola.

Como se mencionó, el trabajo se realizó con el sistema operativo linux con la distribución de Ubuntu 20.04 y para la correcta instalación del CLI de AWS (https://aws.amazon.com/es/cli/) se

necesitó tener Python y PIP instalados. Una vez que contamos con estos requisitos se ejecutó el comando "**pip install awscli**".

Una vez que contamos con una cuenta y el CLI de AWS, se crea un perfil de AWS de forma local que tendrá las credenciales necesarias para administrar los servicios de nuestra cuenta. En la figura 20 se puede observar un perfil con el nombre **awsTesis** y sus correspondientes credenciales.

```
[awsTesis]
aws_access_key_id=AKIA2WRCXCR5JOME446T
aws_secret_access_key=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
aws_region=us-east-1
```

Figura 20. Perfil de AWS que dará acceso a los servicios de una cuenta específica de AWS

3.7 Operaciones CRUD

La elección de estos lenguajes se basó en que, como se mencionó anteriormente en el capítulo 2, actualmente son los más populares entre los programadores para el desarrollo web. Esta característica es muy importante ya que en los últimos años el desarrollo web fue el principal impulsor de plataformas de computación en la nube que facilitan y mejoran los procesos de desarrollo.

A continuación se detallan las implementaciones de las funciones CRUD en cada uno de los lenguajes seleccionados. Cada una de estas implementaciones corresponde a una API distinta que se desplegó dentro de la plataforma AWS utilizando el framework Serverless y los servicios de API Gateway, Lambda y DynamoDB.

3.7.1 Operaciones CRUD en JavaScript

Para el desarrollo de las operaciones CRUD en JavaScript, se optó por utilizar NodeJS que es un entorno de ejecución para javascript multiplataforma y de código que se ejecuta del lado del servidor. La elección fue realizada ya que es una de las herramientas más populares actualmente para el desarrollo web y consta de una comunidad muy grande, lo que permite encontrar soluciones rápidas a los problemas comunes que puedan aparecer. Además cuenta con soporte en el servicio Lambda de AWS que se utilizó en este trabajo.

Para empezar a desarrollar en NodeJS simplemente se descargó el instalador de la página oficial https://nodejs.org/es/ que ya consta del sistema de gestión de dependencias NPM. Una vez instalado, se procedió con el desarrollo de las operaciones.

En primer lugar se definieron los archivos de configuración yml del framework serverless de las operaciones CRUD para las entidades de formularios y respuestas. Estos archivos se pueden observar en las figuras 21 y 22.

```yaml
createForm:
  handler: src/api/handlers/forms/createForm/createFormHandler.createForm
  package:
    patterns:
      - src/api/handlers/forms/createForm/**
  events:
    - http:
        path: /forms
        method: post
getFormByTitle:
  handler: src/api/handlers/forms/getFormByTitle/getFormByTitleHandler.getFormByTitle
  package:
    patterns:
      - src/api/handlers/forms/getFormByTitle/**
  events:
    - http:
        path: /forms/getFormByTitle
        method: get
updateForm:
  handler: src/api/handlers/forms/updateForm/updateFormHandler.updateForm
  package:
    patterns:
      - src/api/handlers/forms/updateForm/**
  events:
    - http:
        path: /forms/{id}
        method: put
deleteForm:
  handler: src/api/handlers/forms/deleteForm/deleteFormHandler.deleteForm
  package:
    patterns:
      - src/api/handlers/forms/deleteForm/**
  events:
    - http:
        path: /forms/{id}
        method: delete
```

Figura 21. Archivo de configuración para desplegar las operaciones en NodeJS de los formularios en AWS Lambda

```
createResult:
  handler: src/api/handlers/results/createResult/createResultHandler.createResult
  package:
    patterns:
      - src/api/handlers/results/createResult/**
  events:
    - http:
        path: /results
        method: post
getResult:
  handler: src/api/handlers/results/getResult/getResultHandler.getResult
  package:
    patterns:
      - src/api/handlers/results/getResult/**
  events:
    - http:
        path: /results/{id}
        method: get
createRandomResult:
  handler: src/api/handlers/results/createRandomResult/createRandomResultHandler.createRandomResult
  package:
    patterns:
      - src/api/handlers/results/createRandomResult/**
  events:
    - http:
        path: /results/randomResults
        method: post
updateResult:
  handler: src/api/handlers/results/updateResult/updateResultHandler.updateResult
  package:
    patterns:
      - src/api/handlers/results/updateResult/**
  events:
    - http:
        path: /results/{id}
        method: put
deleteResult:
  handler: src/api/handlers/results/deleteResult/deleteResultHandler.deleteResult
  package:
    patterns:
      - src/api/handlers/results/deleteResult/**
  events:
    - http:
        path: /results/{id}
        method: delete
```

Figura 22. Archivo de configuración para desplegar las operaciones en NodeJS de las respuestas en AWS Lambda

Con los archivos de configuración listos, se avanzó con la creación de la estructura de carpetas indicada en la configuración y con el desarrollo del código de las funciones especificadas. El código de estas funciones puede observarse en el repositorio de github indicado anteriormente.

Finalmente con la configuración, el código listo y con la ayuda del framework serverless se desplegaron las operaciones al servicio Lambda de la cuenta de AWS previamente configurada. Este despliegue se llevó a cabo ejecutando el comando "**serverless deploy --aws-profile awsTesis --region us-east-1**". En la figura 23 se puede observar el resultado del despliegue de las funciones a AWS Lambda con serverless. En esta figura se puede observar un resumen con los nombres de las funciones creadas y las direcciones URL a las cuales se les pueden realizar peticiones HTTPS. Por último en la figura 24 se puede observar que las funciones ya se encuentran desplegadas en AWS Lambda.

```
service: forms-api-nodejs
stage: tesis
region: us-east-1
stack: forms-api-nodejs-tesis
resources: 69
api keys:
  None
endpoints:
  POST - https://ulxgjhorie.execute-api.us-east-1.amazonaws.com/tesis/forms
  GET - https://ulxgjhorie.execute-api.us-east-1.amazonaws.com/tesis/forms/getFormByTitle
  PUT - https://ulxgjhorie.execute-api.us-east-1.amazonaws.com/tesis/forms/{id}
  DELETE - https://ulxgjhorie.execute-api.us-east-1.amazonaws.com/tesis/forms/{id}
  POST - https://ulxgjhorie.execute-api.us-east-1.amazonaws.com/tesis/results
  GET - https://ulxgjhorie.execute-api.us-east-1.amazonaws.com/tesis/results/{id}
  POST - https://ulxgjhorie.execute-api.us-east-1.amazonaws.com/tesis/results/randomResults
  PUT - https://ulxgjhorie.execute-api.us-east-1.amazonaws.com/tesis/results/{id}
  DELETE - https://ulxgjhorie.execute-api.us-east-1.amazonaws.com/tesis/results/{id}
  GET - https://ulxgjhorie.execute-api.us-east-1.amazonaws.com/tesis/bot/run
  GET - https://ulxgjhorie.execute-api.us-east-1.amazonaws.com/tesis/bot/callBot
functions:
  createForm: forms-api-nodejs-tesis-createForm
  getFormByTitle: forms-api-nodejs-tesis-getFormByTitle
  updateForm: forms-api-nodejs-tesis-updateForm
  deleteForm: forms-api-nodejs-tesis-deleteForm
  createResult: forms-api-nodejs-tesis-createResult
  getResult: forms-api-nodejs-tesis-getResult
  createRandomResult: forms-api-nodejs-tesis-createRandomResult
  updateResult: forms-api-nodejs-tesis-updateResult
  deleteResult: forms-api-nodejs-tesis-deleteResult
  saveRandomResultsBot: forms-api-nodejs-tesis-saveRandomResultsBot
  callBot: forms-api-nodejs-tesis-callBot
layers:
  None

**********************************************************************************************************************************
Serverless: Announcing Metrics, CI/CD, Secrets and more built into Serverless Framework. Run "serverless login" to activate for free..
**********************************************************************************************************************************
```

Figura 23. Resultado de la ejecución del comando del framework serverless que despliega una API en NodeJS con funciones de AWS Lambda

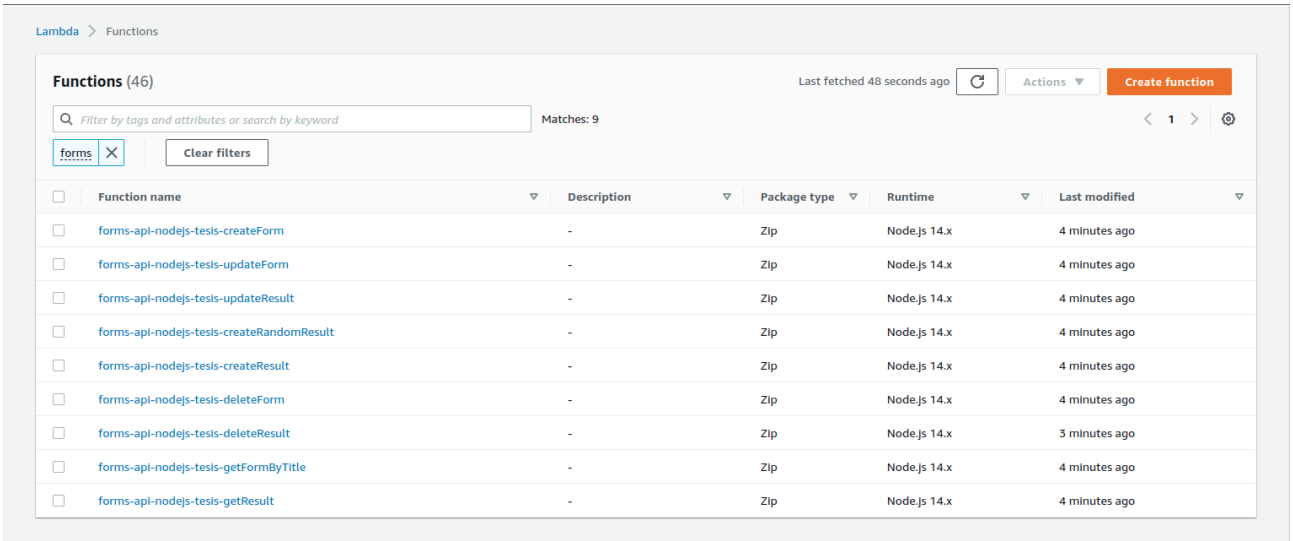


Figura 24. Funciones en NodeJS desplegadas en el servicio AWS Lambda

Con las operaciones desplegadas en AWS se pueden realizar peticiones HTTPS a estas para comprobar su funcionamiento. En la figura 25 se encuentra una solicitud a la operación de crear formulario junto a la respuesta de esa solicitud.

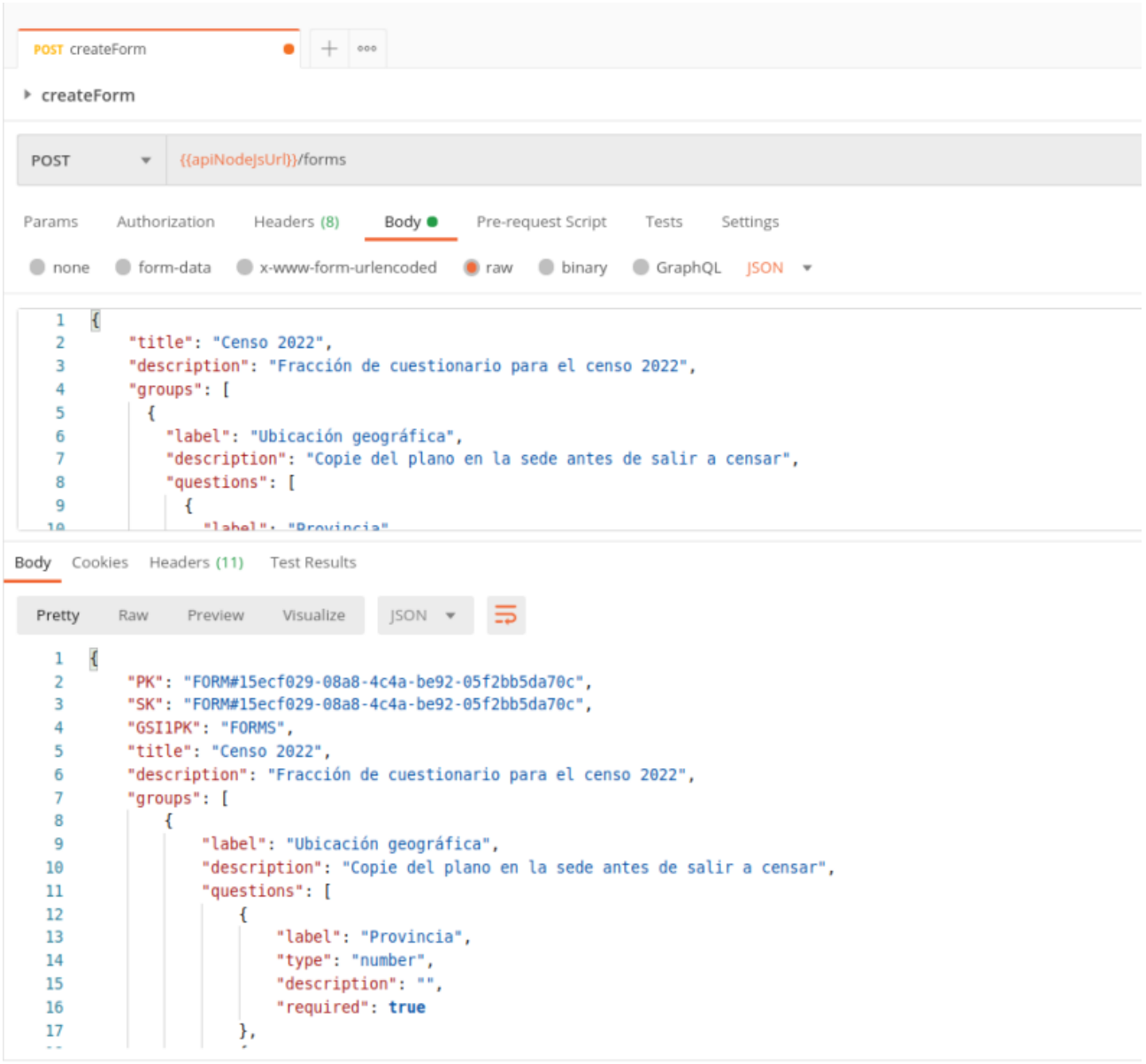


Figura 25. Petición de crear formulario mediante la herramienta postman

En la parte superior de la figura 25 se puede ver el cuerpo de la petición enviada y en la parte inferior, el de la respuesta a la petición realizada.

3.7.1.1 Función de autocompletado aleatoria

Con el objetivo de obtener distintas respuestas del formulario, sin importar su valor, se optó para este caso específico crear una función aleatoria. Lo optado se debe a que la mayoría de las preguntas consisten en opciones. Esta función va a permitir, mediante el envío de un formulario cualquiera, poder obtener una respuesta completa a este. Su funcionamiento es simple y para cada pregunta que disponga de opciones como respuestas, está función seleccionará una de las opciones disponibles de forma totalmente aleatoria. En caso de que una pregunta no consista de opciones la respuesta a esta podrá ser solamente del tipo numérica por lo que la función elegirá un número dentro del rango entre 1 y 100.

Los códigos para la generación aleatoria de las opciones y de las respuestas se pueden visualizar en las figuras 26 y 27.

```
const createRandomResult = async (event) => {
    try {
        const form = JSON.parse(event.body)

        let randomResult = buildRandomResult(form)
        console.log('answer: ', JSON.stringify(randomResult))

        return response(200, randomResult)

    } catch (err) {
        console.log('err', err)
    }
}

const buildRandomResult = (form) => {

    const resultId = uuidv4()

    let result = {
        PK: RESULTS_ID + '#' + resultId,
        SK: form.PK,
        title: form.title,
        description: form.description,
        groups: form.groups,
    }

    form.groups.forEach((group) => {
        group.questions.forEach((question) => {
            const randomAnswer = getRandomAnswer(question)
            question['value'] = randomAnswer
        })
    })
    return result
}
```

Figura 26. Código en JavaScript que genera una opción aleatoria como respuesta

```
const getRandomInt = (min, max) => {
    min = Math.ceil(min)
    max = Math.floor(max)
    return Math.floor(Math.random() * (max - min + 1)) + min
}
```

Figura 27. Código en JavaScript que genera un número aleatorio entre un mínimo y un máximo

3.7.2 Operaciones CRUD en Python

El desarrollo con el lenguaje de programación Python necesitó de la instalación de dos dependencias, Python en sí y PIP que al igual que NPM para NodeJS es un sistema de gestión de paquetes que nos permite administrar e instalar paquetes de software escritos en Python. Actualmente las últimas versiones de Python ya vienen con PIP por lo que no hubo que instalarlo de forma independiente, sino que ya venía incluido. Se pueden obtener las últimas versiones de Python accediendo a su página oficial https://www.python.org/downloads/. Para este trabajo se utilizó la versión 3.8 de Python.

Una vez dadas las condiciones anteriores, se inició el desarrollo con este lenguaje. Se crearon los archivos de configuración YAML para que serverless pueda desplegar las operaciones de formularios y respuestas. Esta configuración realizada se puede observar en las figuras 28 y 29.

```yaml
createForm:
  module: src/api/handlers/forms/createForm
  handler: createFormHandler.createForm
  package:
    patterns:
      - src/api/handlers/forms/**
  events:
    - http:
        path: /forms
        method: post
getFormByTitle:
  module: src/api/handlers/forms/getFormByTitle
  handler: getFormByTitleHandler.getFormByTitle
  package:
    patterns:
      - src/api/handlers/forms/**
  events:
    - http:
        path: /forms/getFormByTitle
        method: get
updateForm:
  module: src/api/handlers/forms/updateForm
  handler: updateFormHandler.updateForm
  package:
    patterns:
      - src/api/handlers/forms/**
  events:
    - http:
        path: /forms/{id}
        method: put
deleteForm:
  module: src/api/handlers/forms/deleteForm
  handler: deleteFormHandler.deleteForm
  package:
    patterns:
      - src/api/handlers/forms/**
  events:
    - http:
        path: /forms/{id}
        method: delete
```

Figura 28. Archivo de configuración para desplegar las operaciones en Python de los formularios en AWS Lambda

```yaml
createResult:
  module: src/api/handlers/results/createResult
  handler: createResultHandler.createResult
  package:
    patterns:
      - src/api/handlers/results/createResult/**
  events:
    - http:
        path: /results
        method: post
getResult:
  module: src/api/handlers/results/getResult
  handler: getResultHandler.getResult
  package:
    patterns:
      - src/api/handlers/results/getResult/**
  events:
    - http:
        path: /results/{id}
        method: put
updateResult:
  module: src/api/handlers/results/updateResult
  handler: updateResultHandler.updateResult
  package:
    patterns:
      - src/api/handlers/results/updateResult/**
  events:
    - http:
        path: /results/{id}
        method: put
deleteResult:
  module: src/api/handlers/results/deleteResult
  handler: deleteResultHandler.deleteResult
  package:
    patterns:
      - src/api/handlers/results/deleteResult/**
  events:
    - http:
        path: /results/{id}
        method: delete
```

Figura 29. Archivo de configuración para desplegar las operaciones en Python de las respuestas en AWS Lambda

Con la configuración de las funciones listas, se creó la misma estructura mencionada anteriormente en la sección de operaciones CRUD en Javascript (NodeJS) y se realizó el código de estas funciones en Python. Este código está disponible en el repositorio GitHub ya mencionado.

Por último con la ayuda del framework serverless se desplegaron las operaciones al servicio Lambda de AWS. Este despliegue se llevó a cabo ejecutando el comando "**serverless deploy --aws-profile awsTesis --region us-east-1**". En la figura 30 se puede observar el resultado del despliegue de las funciones a AWS Lambda con serverless. Se indica un resumen con los nombres de las funciones creadas y las direcciones URL a las cuales se les pueden realizar peticiones HTTPS. Por último, en la figura 31 ya las funciones se encuentran desplegadas en AWS Lambda.

```
stack: forms-api-python-tesis
resources: 50
api keys:
  None
endpoints:
  POST - https://p1akbeueni.execute-api.us-east-1.amazonaws.com/tesis/forms
  GET - https://p1akbeueni.execute-api.us-east-1.amazonaws.com/tesis/forms/getFormByTitle
  PUT - https://p1akbeueni.execute-api.us-east-1.amazonaws.com/tesis/forms/{id}
  DELETE - https://p1akbeueni.execute-api.us-east-1.amazonaws.com/tesis/forms/{id}
  POST - https://p1akbeueni.execute-api.us-east-1.amazonaws.com/tesis/results
  GET - https://p1akbeueni.execute-api.us-east-1.amazonaws.com/tesis/results/{id}
  PUT - https://p1akbeueni.execute-api.us-east-1.amazonaws.com/tesis/results/{id}
  DELETE - https://p1akbeueni.execute-api.us-east-1.amazonaws.com/tesis/results/{id}
functions:
  createForm: forms-api-python-tesis-createForm
  getFormByTitle: forms-api-python-tesis-getFormByTitle
  updateForm: forms-api-python-tesis-updateForm
  deleteForm: forms-api-python-tesis-deleteForm
  createResult: forms-api-python-tesis-createResult
  getResult: forms-api-python-tesis-getResult
  updateResult: forms-api-python-tesis-updateResult
  deleteResult: forms-api-python-tesis-deleteResult
layers:
  None
```

Figura 30. Resultado de la ejecución del comando del framework serverless que despliega una API en Python con funciones de AWS Lambda

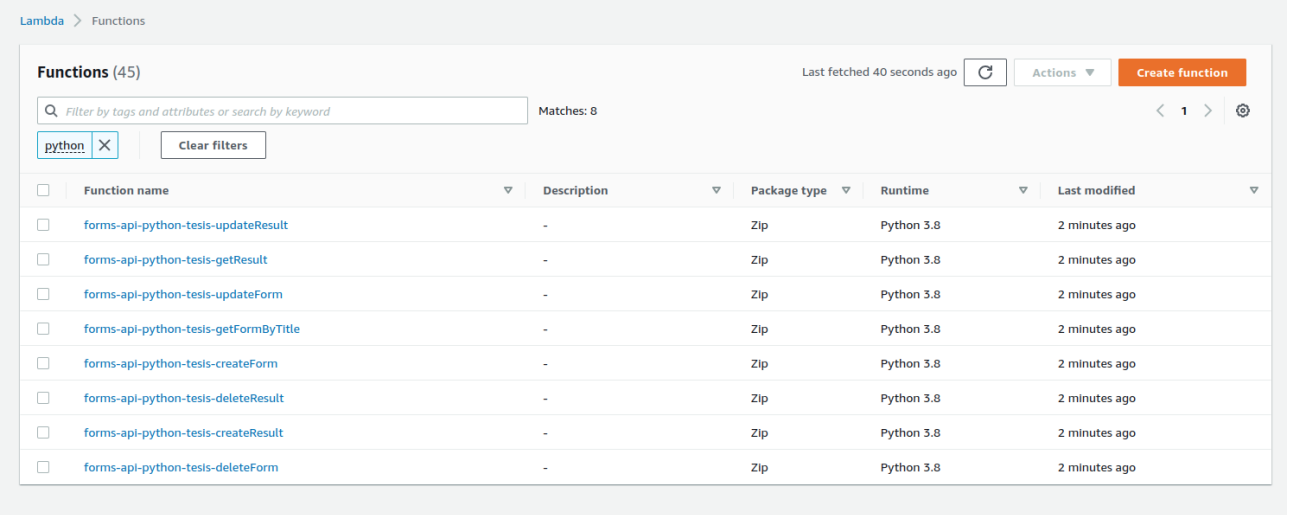


Figura 31. Funciones en Python desplegadas en el servicio AWS Lambda

Con las operaciones desplegadas en AWS se pueden realizar peticiones HTTPS a estas para comprobar su funcionamiento. Encontramos en la figura 32, una solicitud a la operación de obtener formulario junto a la respuesta de esta solicitud.

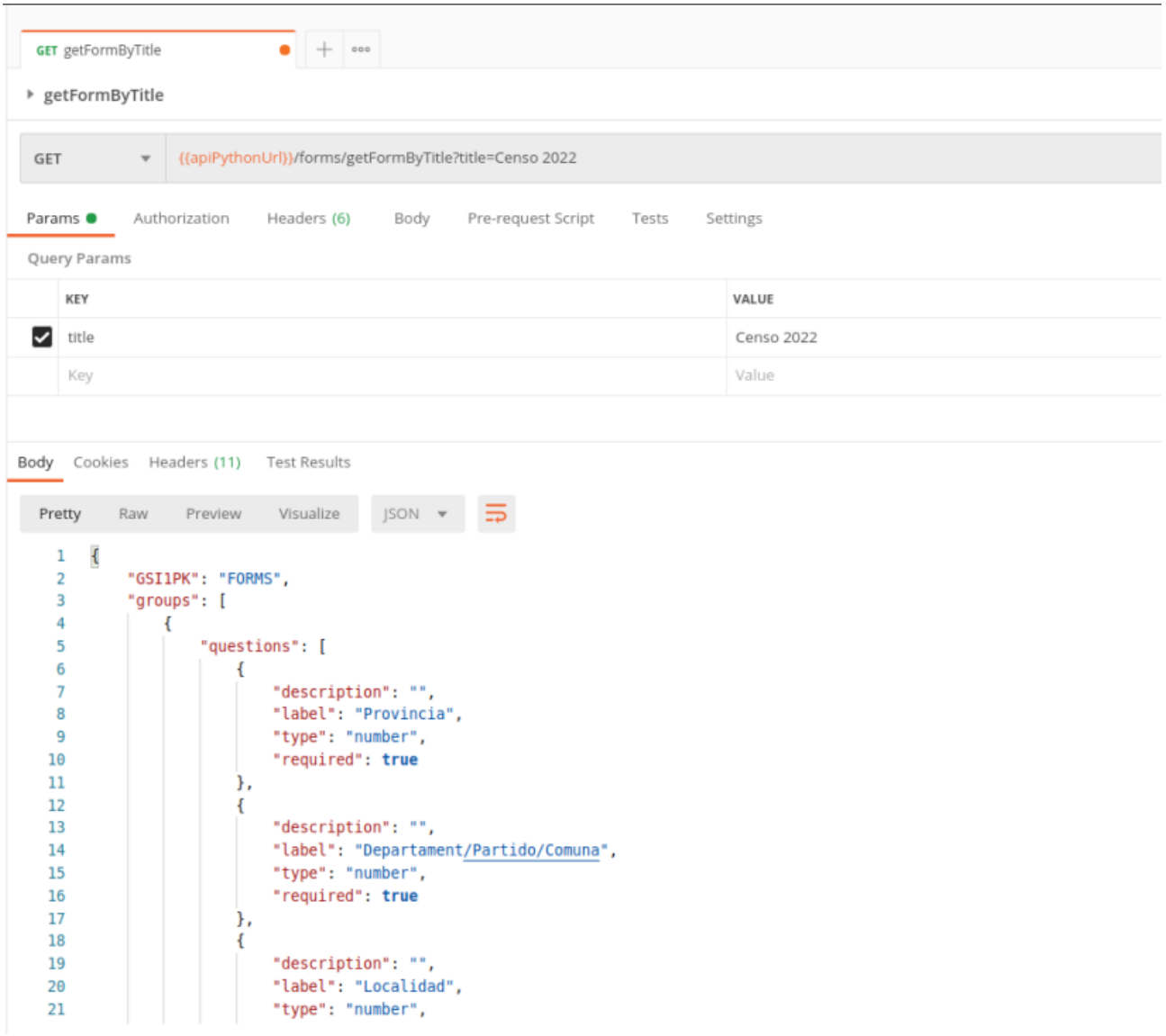


Figura 32. Petición de obtener formulario mediante la herramienta postman

En la parte inferior se puede ver el cuerpo de la respuesta a la petición realizada con el formulario correspondiente.

3.7.3 Operaciones CRUD en Java

Para el desarrollo de las operaciones con Java, se tuvo que proceder con la instalación de OpenJDK versión 11 y con la instalación de la herramienta de software para la gestión y construcción de proyectos Java llamada Maven.

Una vez instaladas estas dependencias, se inició el desarrollo con Java. Se crearon los archivos de configuración YAML para que serverless pueda desplegar las operaciones de formularios y respuestas. Esta configuración realizada se puede observar en las figuras 33 y 34.

```yaml
createForm:
  handler: com.serverless.forms.createform.CreateFormHandler
  timeout: 15
  events:
    - http:
        path: /forms
        method: post
getFormByTitle:
  handler: com.serverless.forms.getformbytitle.GetFormByTitleHandler
  timeout: 15
  events:
    - http:
        path: /forms/getFormByTitle
        method: get
updateForm:
  handler: com.serverless.forms.updateform.UpdateFormHandler
  timeout: 15
  events:
    - http:
        path: /forms/{id}
        method: put
deleteForm:
  handler: com.serverless.forms.deleteform.DeleteFormHandler
  timeout: 15
  events:
    - http:
        path: /forms/{id}
        method: delete
```

Figura 33. Archivo de configuración para desplegar las operaciones en Java, de los formularios en AWS Lambda

```yaml
createResult:
  handler: com.serverless.results.createresult.CreateResultHandler
  timeout: 15
  events:
    - http:
        path: /results
        method: post
getResult:
  handler: com.serverless.results.getresult.GetResultHandler
  timeout: 15
  events:
    - http:
        path: /results/{id}
        method: get
updateResult:
  handler: com.serverless.results.updateresult.UpdateResultHandler
  timeout: 15
  events:
    - http:
        path: /results/{id}
        method: put
deleteResult:
  handler: com.serverless.results.deleteresult.DeleteResultHandler
  timeout: 15
  events:
    - http:
        path: /results/{id}
        method: delete
```

Figura 34. Archivo de configuración para desplegar las operaciones en Java, de las respuestas en AWS Lambda

Con la configuración de las funciones listas, se creó la misma estructura mencionada anteriormente en la sección de operaciones CRUD en Javascript (NodeJS) y se realizó el código de estas funciones en Java. Este código está disponible en el repositorio GitHub ya mencionado.

Por último con la ayuda del framework serverless se desplegaron las operaciones al servicio Lambda de AWS. Este despliegue se llevó a cabo ejecutando el comando "**serverless deploy --aws-profile awsTesis --region us-east-1**". En la figura 35 se puede observar el resultado del despliegue de las funciones a AWS Lambda con serverless. Se indica un resumen con los nombres de las funciones creadas y las direcciones URL a las cuales se les pueden realizar peticiones HTTPS. Por último en la figura 36 las funciones ya se encuentran desplegadas en AWS Lambda.

```
stack: forms-api-java-tesis
resources: 50
api keys:
  None
endpoints:
  POST - https://51vfyk8pgb.execute-api.us-east-1.amazonaws.com/tesis/forms
  GET - https://51vfyk8pgb.execute-api.us-east-1.amazonaws.com/tesis/forms/getFormByTitle
  PUT - https://51vfyk8pgb.execute-api.us-east-1.amazonaws.com/tesis/forms/{id}
  DELETE - https://51vfyk8pgb.execute-api.us-east-1.amazonaws.com/tesis/forms/{id}
  POST - https://51vfyk8pgb.execute-api.us-east-1.amazonaws.com/tesis/results
  GET - https://51vfyk8pgb.execute-api.us-east-1.amazonaws.com/tesis/results/{id}
  PUT - https://51vfyk8pgb.execute-api.us-east-1.amazonaws.com/tesis/results/{id}
  DELETE - https://51vfyk8pgb.execute-api.us-east-1.amazonaws.com/tesis/results/{id}
functions:
  createForm: forms-api-java-tesis-createForm
  getFormByTitle: forms-api-java-tesis-getFormByTitle
  updateForm: forms-api-java-tesis-updateForm
  deleteForm: forms-api-java-tesis-deleteForm
  createResult: forms-api-java-tesis-createResult
  getResult: forms-api-java-tesis-getResult
  updateResult: forms-api-java-tesis-updateResult
  deleteResult: forms-api-java-tesis-deleteResult
layers:
  None
```

Figura 35. Resultado de la ejecución del comando del framework serverless que despliega una API en Java, con funciones de AWS Lambda

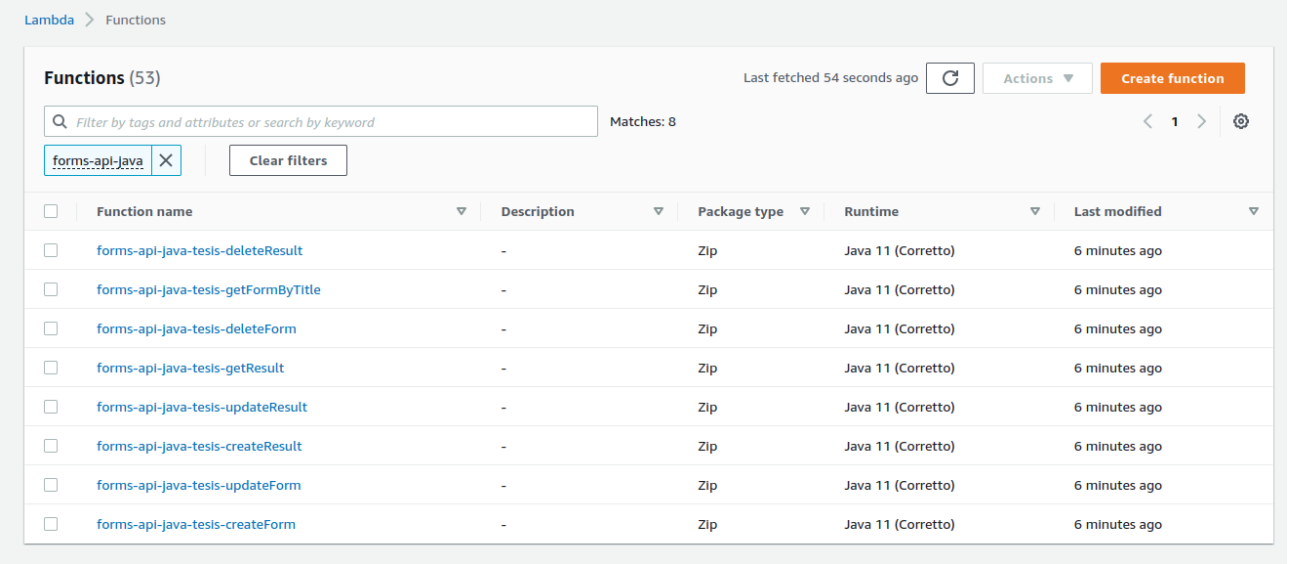


Figura 36. Funciones en Java desplegadas en el servicio AWS Lambda

Con las operaciones desplegadas en AWS se pueden realizar peticiones HTTPS a estas para comprobar su funcionamiento. En la figura 37 se encuentra una solicitud a la operación de crear respuesta de la API en Java.

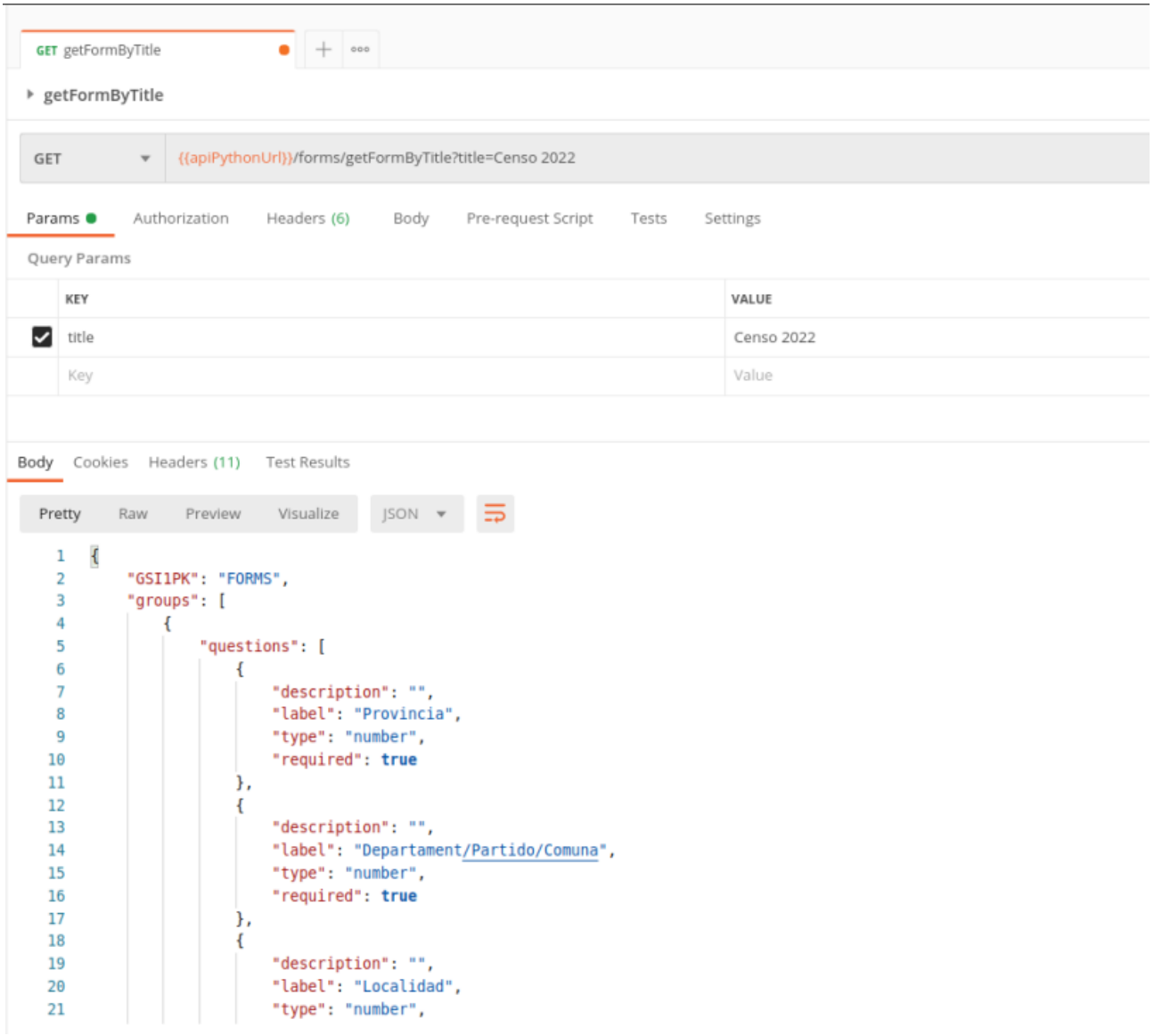


Figura 37. Petición de crear respuesta mediante la herramienta postman

En la parte superior de la figura 37 se puede ver el cuerpo de la petición enviada y en la parte inferior, el de la respuesta a la petición realizada.

3.7.3.1 Problemas de implementación en Java

En las primeras ejecuciones de los lambda que se desarrollaron con el lenguaje de programación JAVA se pudo observar que en la primera ejecución de una función, los tiempos obtenidos resultaron muy grandes en comparación con una segunda ejecución. Ese tiempo no debería variar de forma significativa. Esto, según la documentación oficial de AWS se puede deber a dos principales factores:

Java virtual machine (JVM) lazy class loading: Para reducir el uso de la memoria, la JVM retrasa la inicialización de una biblioteca de clases Java hasta que se llama por primera vez a la biblioteca en una aplicación. Este retraso puede causar una gran cantidad de operaciones de entrada y salida (E/S), lo que da como resultado latencias de mayor duración para las primeras invocaciones en un entorno de ejecución de Lambda.

The Java Reflection API: La API de Java Reflection permite que el código Java descubra información sobre otras clases, interfaces, campos y métodos, y luego opere en sus valores subyacentes. Debido a que la reflexión involucra tipos que se resuelven dinámicamente, no se pueden realizar ciertas optimizaciones de la JVM. En consecuencia, las operaciones reflectantes tienen un rendimiento más lento que sus contrapartes no reflectantes.

Para optimizar el rendimiento de las funciones lambda desarrolladas en JAVA, amazon describe algunas buenas prácticas que fueron tomadas para su implementación (https://aws.amazon.com/es/premiumsupport/knowledge-center/lambda-improve-java-function-performance/).

3.8 Test

Para poder tener una vista clara del comportamiento de los distintos lenguajes de programación en un entorno serverless y en un contexto de aplicativo real, se realizaron tres test diferentes. Estas tres pruebas tienen como objetivo poner a discusión los tiempos de ejecución, los tiempos de inicialización y las implicancias del tamaño de memoria aprovisionado en las funciones lambdas. Los test desarrollados son los siguientes:

- Test de inicialización.
- Test de escalabilidad.
- Test de memoria.

Para realizar la comparativa con los distintos lenguajes, se utilizó la herramienta de código abierta Artillery la cual facilita la creación de estos test a través de archivos de configuración simples.

La definición de los test de Artillery se realizaron en formato YAML y para ejecutarlos se utilizó la interfaz de línea de comandos de la herramienta. Estas pruebas se componen de una sección con uno o más escenarios y de otra sección de configuración. Dentro de esta sección de configuración se encuentran las fases de carga, estas se encargan de decirle a Artillery cuántos usuarios virtuales debe crear durante un periodo de tiempo determinado. Estas fases permiten modelar la carga que tendrá un servicio durante la ejecución de una prueba. Generalmente un test tendrá varías fases de carga, comenzando por una de calentamiento, seguida de una fase de aceleración suave, seguida de una o más fases de cargas más pesadas.

En la figura 38 se puede observar una definición estándar de una prueba de Artillery.

```yaml
config:
  target: https://artillery.io
  phases:
    - duration: 60
      arrivalRate: 10
    - duration: 600
      arrivalRate: 100
scenarios:
  - flow:
      - get:
          url: "/"
      - get:
          url: "/docs"
```

Figura 38. Ejemplo de definición de un test con la herramienta Artillery

Para este trabajo se creó un archivo de configuración para ser utilizado en cada una de las ejecuciones de los test y con cada lenguaje de programación. Estos archivos tienen la URL específica de cada API y las distintas fases de carga del test con sus tiempos y número de solicitudes correspondientes (también llamados como usuarios virtuales). Estos archivos se detallarán más adelante en la sección de cada uno de ellos.

3.8.1 Escenario de prueba

Se definió un escenario en Artillery el cual se utilizó en los tres test mencionados anteriormente. Este marca el flujo que se realizó por cada usuario virtual creado por Artillery a la hora de ejecutar cada test y fue planteado para simular a un usuario que está "utilizando" una web la cual realiza peticiones a una API Rest. Este escenario consiste de los siguientes pasos:

1. **Obtener formulario**: El usuario selecciona un formulario por su nombre, en este caso selecciona "Censo digital 2022". Se le solicita a la API un formulario específico enviando el id o el nombre de este.
2. **Crear respuesta aleatoria**: Se ejecuta la operación que devuelve una respuesta aleatoria a un formulario enviado.
3. **Guardar respuesta del formulario**: El usuario completa el formulario y lo guarda. Se envía una solicitud a la API para que guarde la respuesta.
4. **Actualizar respuesta**: El usuario actualiza la respuesta guardada anteriormente y se envía una solicitud a la API para que actualice la respuesta.
5. **Borrar respuesta**: El usuario borra la respuesta del formulario.

La figura 39 muestra la definición del escenario que se utilizará en los test planteados.

```yaml
scenarios:
  - name: "Load test scenario"
    flow:
      - get:
          url: "/forms/getFormByTitle"
          qs:
            title: "Censo digital 2022"
          capture:
            - json: "$"
              as: "form"
      - post:
          url: "https://c320u3kjjf.execute-api.us-east-1.amazonaws.com/tesis/results/randomResults"
          json: "{{ form }}"
          capture:
            - json: "$"
              as: "randomResult"
      - post:
          url: "/results"
          json: "{{ randomResult }}"
          capture:
            - json: "$"
              as: "savedResult"
      - function: "processIds"
      - put:
          url: "/results/{{ resultPK }}"
          json: "{{ savedResult }}"
      - delete:
          url: "/results/{{ resultPK }}?formId={{ resultSK }}"
```

Figura 39. Escenario que realizará cada usuario virtual a la hora de ejecutar el test

3.8.2 Test de inicialización

Para poder observar las diferencias de los tiempos que toma AWS para levantar los contextos de ejecución de las lambdas de cada uno de los lenguajes, se realizó este test cuyo propósito fue el de marcar estas diferencias mediante una ejecución en frío de las lambdas cada una hora, ya que según Cordingly et al. (2020), al esperar 1 hora entre cada ejecución, podemos asegurar que AWS da de baja la infraestructura FaaS retornando cada función a un estado frío. Además, también se realizó una solicitud al minuto de la primera para forzar una ejecución en caliente y poder comparar los tiempos de ejecución en frío y caliente. Esto nos permite ver si hay discrepancias entre estas dos.

En la figura 40 se encuentra el archivo de configuración de este test. En este se define un target que indica la URL de la API Rest a donde se realizarán las solicitudes, un timeout de 10 segundos, un archivo donde se puede poner código personalizado para lograr los comportamientos deseados (en el caso de todos los test se utiliza para cambiar el formato de unos identificadores que vienen de la API) y finalmente se indica una etapa de fases donde podemos definir la carga con la que someteremos a las APIs implementadas. Como se dijo anteriormente tenemos una primera solicitud, seguida de una pausa de 60 segundos y finalmente una última llamada para forzar una ejecución en caliente de las lambdas.

```yaml
config:
  target: "{{$processEnvironment.API_URL}}"
  http:
    timeout: 10
  processor: "./utilities/utilities.js"
  phases:
    - duration: 1
      arrivalCount: 1
    - pause: 60
      name: "Pausa"
    - duration: 1
      arrivalCount: 1
```

Figura 40. Archivo de configuración del test de inicialización

3.8.3 Test de escalabilidad

A diferencia del test de inicialización, con este test se busca someter a las distintas APIs a diferentes cargas de solicitudes concurrentes realizadas en un tiempo de 1 segundo. El motivo es el de observar cómo escala el servicio Lambda de AWS para cada lenguaje de programación distinto y como afecta esto a los tiempos de ejecución de las lambdas. Es de gran importancia observar aquellas ejecuciones que sufrieron de tiempos de inicialización, ya que esto permite explicar algunos incrementos en los tiempos de ejecución obtenidos.

Este test difiere del test de inicialización en la definición de sus fases de carga. La figura 41 muestra las diferencias entre estos dos primeros test. Para este caso, se definen seis fases que marcan el número de solicitudes a realizarse en un segundo. Este número de solicitudes es la carga que pondremos sobre las APIs desarrolladas. Cada una de estas fases de carga estará seguida, excepto por la última, por una fase de pausa de 1 minuto que dará lugar a la siguiente carga establecida. El número de solicitudes comienza con una sola, la cual permite calentar la infraestructura y luego se va incrementando con las siguientes cantidades: 10, 25, 50, 100 y 200.

```yaml
config:
 target: "{{$processEnvironment.API_URL}}"
 http:
   timeout: 20
 processor: "./utilities/utilities.js"
 phases:
   - duration: 1
     arrivalCount: 1
   - pause: 60
     name: "Pausa"
   - duration: 1
     arrivalCount: 10
   - pause: 60
     name: "Pausa"
   - duration: 1
     arrivalCount: 25
   - pause: 60
     name: "Pausa"
   - duration: 1
     arrivalCount: 50
   - pause: 60
     name: "Pausa"
   - duration: 1
     arrivalCount: 100
   - pause: 60
     name: "Pausa"
   - duration: 1
     arrivalCount: 200
```

Figura 41. Archivo de configuración del test de escalabilidad

3.8.4 Test de memoria

Una de las características a configurar a la hora de desplegar una función Lambda tiene que ver con el tamaño de memoria de esta. Para poder comprobar si el tamaño de memoria atañe de alguna forma al rendimiento de las funciones se implementó este test. El archivo de configuración coincide con el de inicialización (figura 40), es decir, se realiza una primera ejecución en frío y luego de un minuto se realiza otra para forzar una ejecución en caliente. Mediante la primera solicitud se obtuvo información acerca de cómo afecta el tamaño de memoria a la inicialización de las funciones en los diferentes lenguajes. Con la segunda llamada se adquirió datos sobre los tiempos de ejecución y como estos se ven alterados por el tamaño de memoria de las lambdas. Este test fue ejecutado modificando el tamaño de memoria, es decir, se comenzó configurando las lambdas con un tamaño de 10 GB y se fue decrementando la memoria con los siguientes valores: 8, 6, 4, 2, 1 GB. Finalmente se ejecutó con lambdas con 512 MB de memoria.

3.8.5 Ejecución de los test

La ejecución de un test de Artillery se realiza mediante la interfaz de comandos de esta, simplemente con el comando "**artillery run".** Para poner en marcha los tests se realizaron dos comandos en el archivo **package.json** lo que permite que se ejecuten mediante NPM indicando los archivos de configuración detallados anteriormente. En la figura 42 se tiene la definición de los scripts mencionados anteriormente.

```
{
  "name": "tesis-tests",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "init_and_memory_test": "artillery run --config ./config/init_and_memory_conf.yml -k test_flow.yml --output ./reports/init_and_memory.json",
    "scalability_test": "artillery run --config ./config/scalability_conf.yml -k test_flow.yml --output ./reports/scalability.json"
  },
  "author": "Matias Rodriguez",
  "license": "ISC"
}
```

Figura 42. Scripts de NPM para la ejecución de los test

3.8.6 Datos de los test

Una vez que se ejecutaron los test desarrollados anteriormente, se utilizó los servicios de AWS de CloudWatch con el lenguaje de consultas Log Insight para poder extraer los datos generados a través de los test. Para esto se realizaron las consultas que se pueden visualizar en las figuras 43, 44 y 45 mostrando los resultados de las ejecuciones de los test. Además se valió de los gráficos de Cloudwatch para un primer análisis visual que luego fueron reemplazados por gráficos propios generados a través de los datos de los logs. En la figura 46 se observa, a muestra de ejemplo, un gráfico generado con cloudwatch .

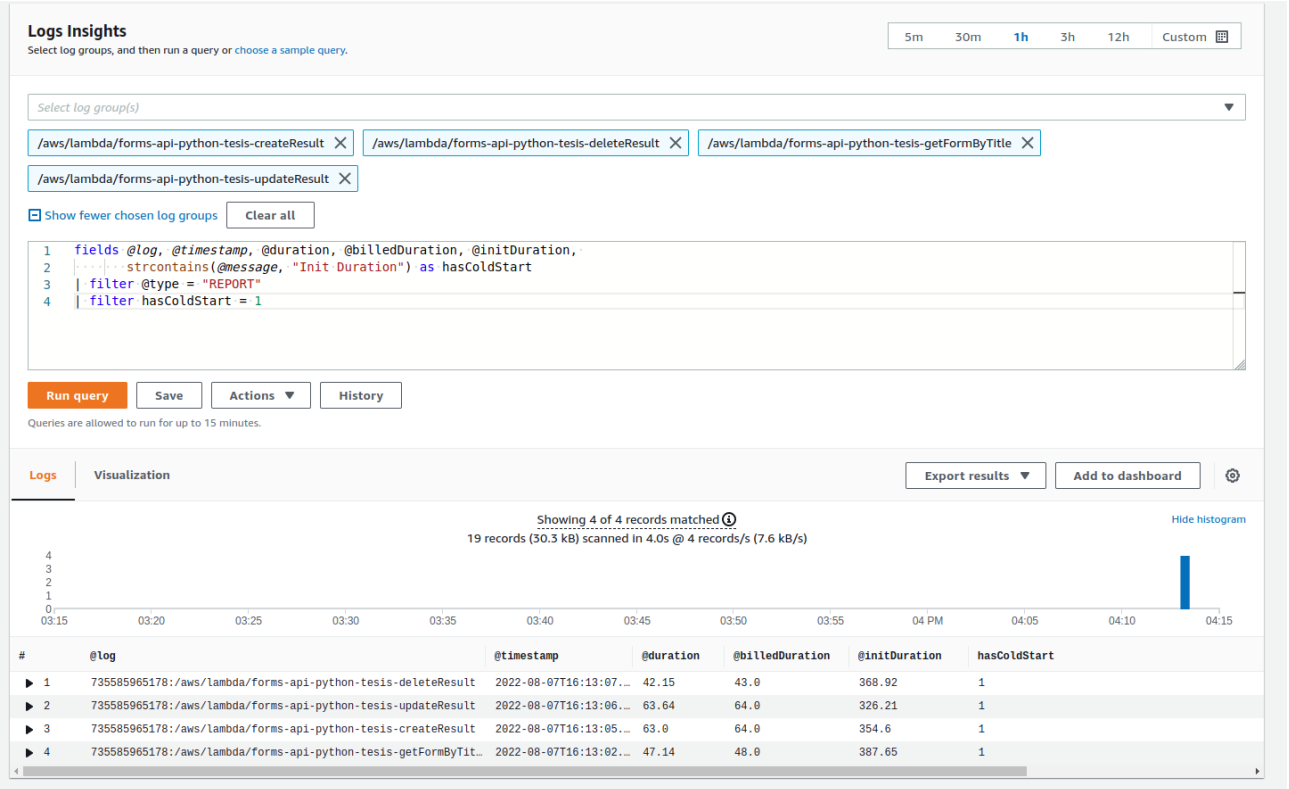


Figura 43. Consulta con Log Insight, para Python, del test de inicialización

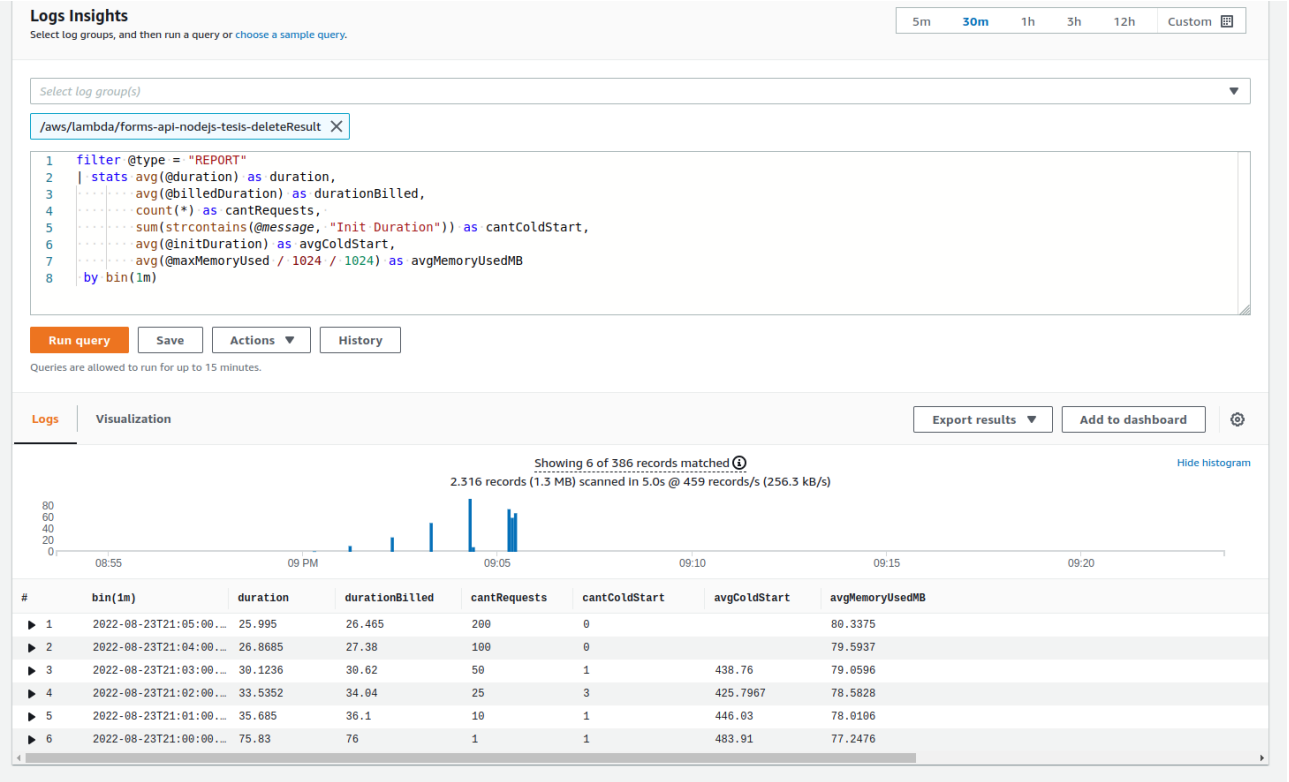


Figura 44. Consulta con Log Insight para delete, con NodeJS, del test de escalabilidad

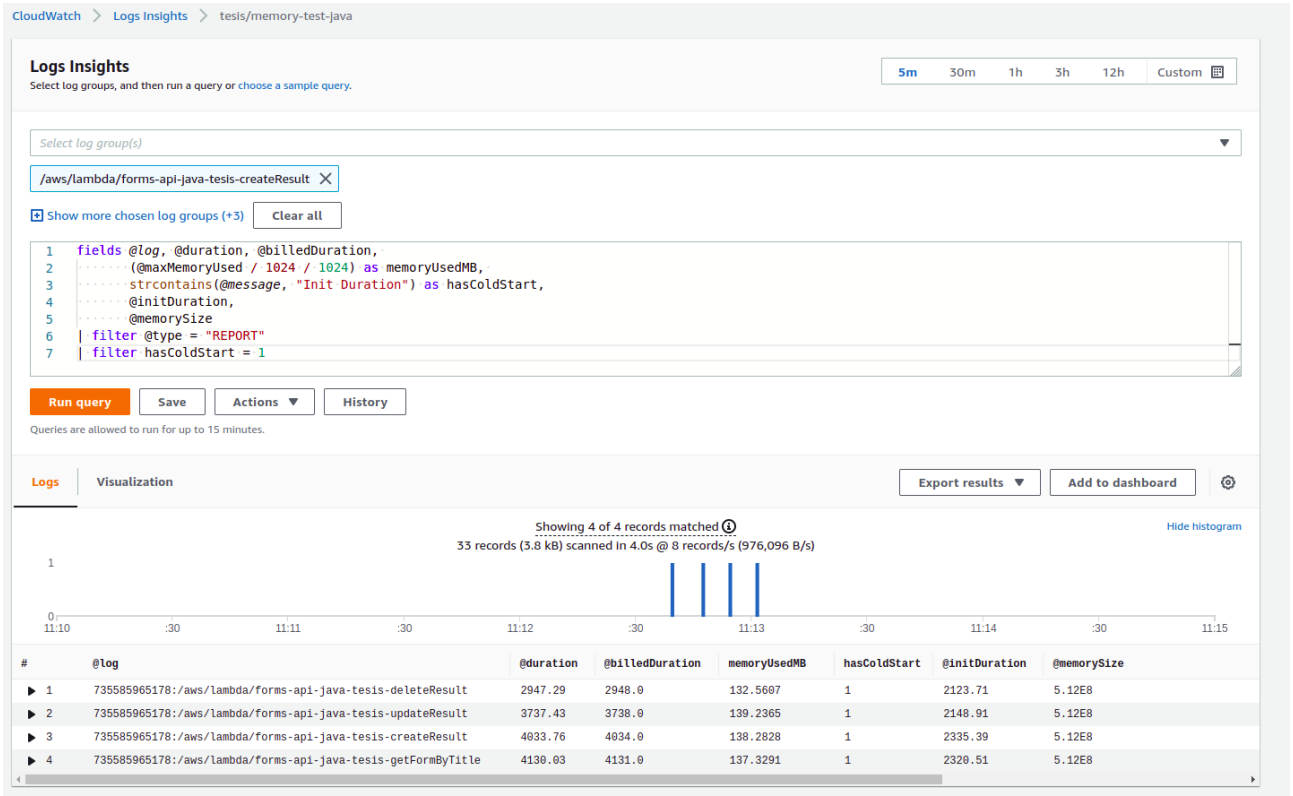


Figura 45. Consulta con Log Insight, para Java, del test de memoria

En las figuras anteriores se puede observar la interfaz de AWS para el servicio de CloudWatch Log Insight. Esta interfaz en primer lugar nos permite seleccionar los grupos de registros sobre los cuales queremos realizar una consulta. Se utiliza un lenguaje de consultas cuya sintaxis admite diferentes funciones, operaciones y expresiones regulares. Una vez seleccionados estos grupos, se procede a escribir la consulta la cual al ser ejecutada permite obtener los resultados de la forma deseada. Por último en la parte inferior de todas estas figuras (43, 44, 45) se pueden visualizar los resultados obtenidos, los cuales pueden ser exportados o convertidos en gráficos lo que facilita la visualización de estos.

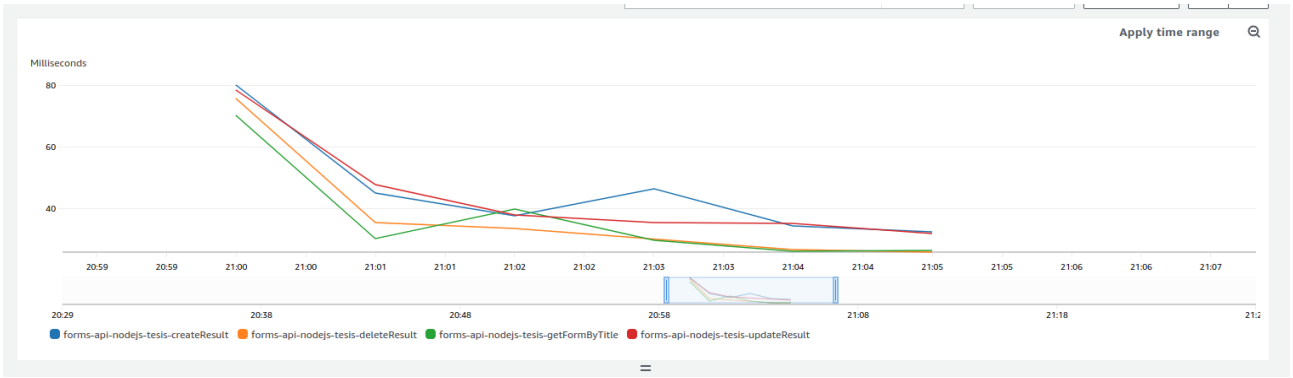


Figura 46. Gráfico de Cloudwatch, para NodeJS, del test de escalabilidad

La figura 46 muestra una captura de la interfaz de AWS para el servicio CloudWatch metrics donde se dibuja un gráfico generado con los datos de los test ejecutados. Este servicio permite seleccionar

las distintas métricas que existen para cada uno de los servicios que ofrece la plataforma. Para este caso se seleccionaron los tiempos de ejecución de las funciones lambdas generadas en los lenguajes de programación mencionados anteriormente.

CAPÍTULO 4

Análisis de Resultados, Conclusiones y Trabajos Futuros

4.1 Métricas

Para realizar el análisis comparativo entre los lenguajes de programación seleccionados, con el caso de estudio elegido, se realizaron varias experiencias a través de tres test diferentes. Cada uno de estos test está orientado a un objetivo específico y fueron desarrollados y ejecutados mediante la herramienta open source Artillery. De esta manera se evaluaron diversos aspectos, tales como: tiempos de inicialización para las primeras solicitudes a las funciones lambdas, tiempos de ejecución en frío, tiempos de ejecución en caliente, número de solicitudes y tamaño de memoria asignado a las funciones.

Los tests desarrollados son los siguientes:

- Test de inicialización.
- Test de escalabilidad.
- Test de memoria.

Los test anteriormente mencionados permitieron el análisis del performance de los lenguajes bajo distintas situaciones.

Por cada uno de estos tests se dará una visión general de los lenguajes y luego se tomará una de las operaciones CRUD para un análisis más detallado. Para llevar a cabo el test de inicialización y el de escalabilidad se optó por ejecutarlos con 10 GB de memoria. Este es el tamaño máximo de memoria ofrecido por AWS en el servicio Lambda, lo cual posibilita utilizar la mayor potencia de CPU permitida por este.

4.2 Test de inicialización

El test de inicialización tiene por objetivo probar las operaciones CRUD de tal forma que siempre se produzca la inicialización del contexto de ejecución de cada función lambda. Cuando esto ocurre AWS nos brinda en sus métricas una variable de tiempo de inicialización que indica cuánto le tomó a la plataforma preparar la infraestructura y dependencias necesarias para permitir la ejecución del código dentro de la lambda que fue escrito por el consumidor del servicio.

Se realizaron cinco ejecuciones en frío para cada lenguaje y operación CRUD, esto se hizo mediante la ejecución del flujo del test mencionado en el capítulo anterior. Es importante remarcar que estas ejecuciones se realizaron cada una hora para asegurar que AWS dió de baja la infraestructura FaaS retornando cada función a un estado frío. Cada vez que se hizo una ejecución en frío, también se realizó una segunda ejecución de forma inmediata para tomar el tiempo de ejecución en caliente, lo que permitió realizar una comparativa entre los tiempos de ejecución en frío y caliente.

4.2.1 Tiempos de inicialización

En la tabla 1, se presentan los promedios obtenidos de los tiempos de inicialización correspondientes a las operaciones CRUD por cada lenguaje de programación.

Tabla 1. *Promedio de tiempos de inicialización por cada operación CRUD y lenguaje de programación*

Lenguaje	Create	Read	Update	Delete
JavaScript (NodeJS)	418,232	431,04	413,586	442,15
Python	384,902	400,43	361,624	411,2
Java	1623,528	1630,52	1526,12	1577,302

En la figura 47 se puede observar que para todas las operaciones y entre los tres lenguajes de programación, Java es el de peor rendimiento cuando se habla de inicialización de los contextos de ejecución para cada función lambda.

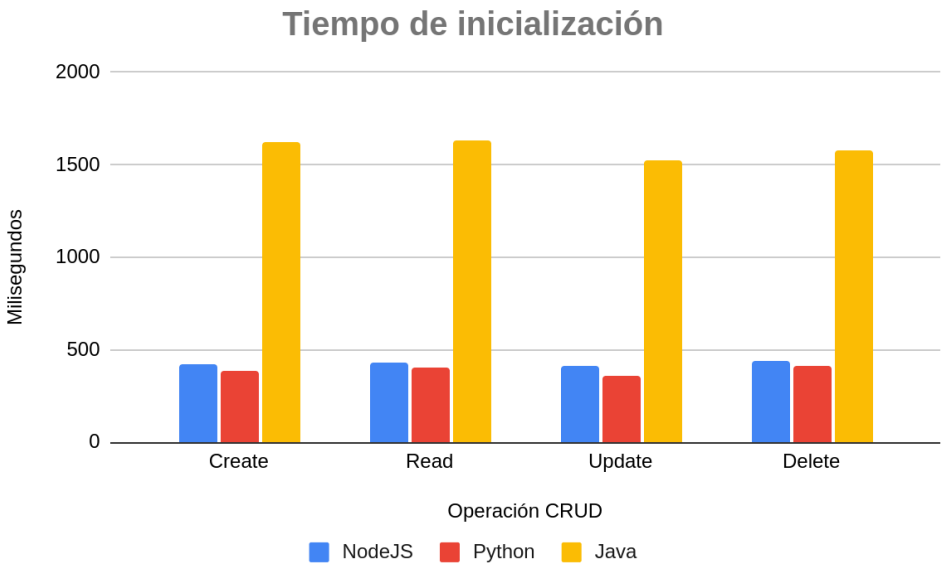


Figura 47. Tiempo de inicialización, en cada lenguaje para cada operación CRUD

Se puede ver una clara diferencia en todas las operaciones CRUD donde la operación read de java es la que mayor tiempo de inicialización registra con 1630.52 ms. En cuanto a NodeJS y Python la

operación más deficiente es delete con tiempos de 442.15 ms. y 411.2 ms. respectivamente. Estas diferencias pueden apreciarse en la figura 48 aunque no son significativas.

Siguiendo el mismo razonamiento la operación update es la de mejor rendimiento a la hora de inicializar el contexto de ejecución para una lambda, cualquiera sea el lenguaje.

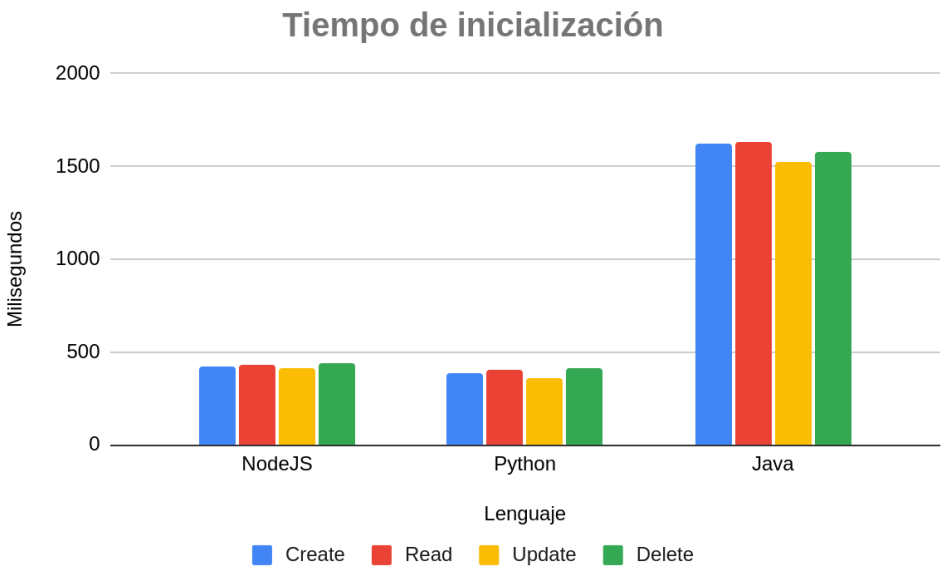


Figura 48. Tiempo de inicialización por cada lenguaje y operación CRUD

Como se mencionó anteriormente, también se obtuvieron resultados en cuanto a los tiempos de ejecución en frío y de una segunda ejecución en caliente. Las tablas 2. 3 y 4, muestran las diferencias entre los tiempos de ejecución en frío y en caliente para cada una de las operaciones CRUD y por cada lenguaje. Las celdas sombreadas corresponden a la operación CRUD que mayor mejora logra. Se puede apreciar que en Java es donde las diferencias son mayores. Por ejemplo con delete, la ejecución pasa de tardar 581.494 ms. en frío a 84.654 ms. en caliente. Es decir, Java es el lenguaje con mayor discrepancia entre esos tiempos de ejecución, obteniendo una mejora en promedio del 84%. Dicho de otro modo, Java es el que reduce los tiempos de forma más abrupta. También NodeJS y Python mejoran al ejecutarse en caliente, encontrando las mayores ganancias en Read y Delete, respectivamente, de aproximadamente un 48% en promedio.

Tabla 2. *Tiempo de ejecución en NodeJS, por cada operación CRUD*

	NodeJS			
Estado del tiempo de ejecución	Create	Read	Update	Delete
Frío	76,698	74,542	84,04	77,768
Caliente	51,238	39,248	45,536	41,066
Diferencia	25,46	35,294	38,504	36,702
Porcentaje de mejora	33,19512895	47,3478039	45,816278	47,1942187

Tabla 3. *Tiempo de ejecución con Python, por cada operación CRUD*

	Python			
Estado del tiempo de ejecución	Create	Read	Update	Delete
Frío	60,734	49,752	65,738	43,97
Caliente	32,582	27,388	38,472	22,566
Diferencia	28,152	22,364	27,266	21,404
Porcentaje de mejora	46,35294892	44,9509567	41,4767714	48,6786445

Tabla 4. *Tiempo de ejecución con Java, por cada operación CRUD*

	Java			
Estado del tiempo de ejecución	Create	Read	Update	Delete
Frío	701,192	664,628	644,328	581,494
Caliente	116,638	98,102	116,89	84,654
Diferencia	584,554	566,526	527,438	496,84
Porcentaje de mejora	83,36575432	85,2395626	81,8586186	85,4419822

4.3 Test de escalabilidad

El segundo test que se planteó fue el de escalabilidad con el propósito de dar una visión clara de cómo se comportan los lenguajes al recibir una cierta cantidad de solicitudes en forma concurrente bajo la infraestructura serverless implementada. Acá lo importante a medir en este test es el tiempo de ejecución. Se tomó en cuenta la cantidad de ejecuciones en frío que hubo dentro del total de ejecuciones, ya que permite explicar algunas diferencias entre los tiempos obtenidos. Además da la posibilidad de ver qué lenguaje escala mejor en este modelo de ejecución serverless.

Este test al igual que los demás sigue el mismo flujo diseñado anteriormente con diferencias en la etapa de configuración. En esta etapa se implementaron seis fases con las cantidades de solicitudes a realizar y cada una de ellas se realizó en un tiempo de un segundo. El número de solicitudes comienza con una sola, la cual permite calentar la infraestructura y luego se va incrementando con

las siguientes cantidades: 10, 25, 50, 100 y 200. Cada una de estas fases está seguida de otra fase de "espera" donde se realiza una pausa de un minuto antes de iniciar la siguiente.

4.3.1 Tiempos de ejecución

Anteriormente en el test de inicialización se obtuvo que Java poseía la peor performance a la hora de preparar la infraestructura encargada de ejecutar el código de una función lambda, pero también se pudo observar que era el lenguaje con el mayor rango de mejora cuando la ejecución se hacía en caliente. Los resultados que se muestran en la figura 49 corresponden a la operación CRUD read para cada lenguaje y permiten confirmar los resultados obtenidos en el test anterior.

Java comienza con un tiempo muy malo en la primera ejecución debido a que sufre los retrasos de inicialización, pero luego al subir la cantidad de ejecuciones concurrentes observamos que para 10 ejecuciones los tiempos mejoran drásticamente a 20 ms. teniendo un mejor performance que NodeJS y Python cuyos tiempos fueron de 30 y 31 ms respectivamente. Este rendimiento superior de Java también se debe a que no ocurrió ninguna inicialización en las 10 ejecuciones, mientras que NodeJS y Python sufrieron de 1 inicialización cada uno lo que afectó a sus tiempos de ejecución. También se puede observar que en las 25 ejecuciones Java empeora debido a que sufre 6 inicializaciones sufriendo un tiempo de 158 ms mientras que NodeJS y Python obtuvieron tiempos de 40 ms y 17 ms respectivamente.

Finalmente a partir de las 50 ejecuciones los tiempos en los 3 lenguajes se estabilizan siendo Java el que nuevamente obtiene los mejores tiempos en las 200 ejecuciones, con un tiempo de 14 ms contra 27 ms y 15 ms de NodeJS y Python respectivamente (ver tabla 5).

Tabla 5. *Tiempo de ejecución para Read, en milisegundos*

Número de ejecuciones concurrentes	Read		
	NodeJS	Python	Java
1	70.32	58.11	652.72
10	30.359	31.658	20.001
25	39.9252	16.6476	158.3124
50	29.7432	17.994	24.3094
100	26.2374	16.5708	19.7675
200	26.6532	15.444	13.9122

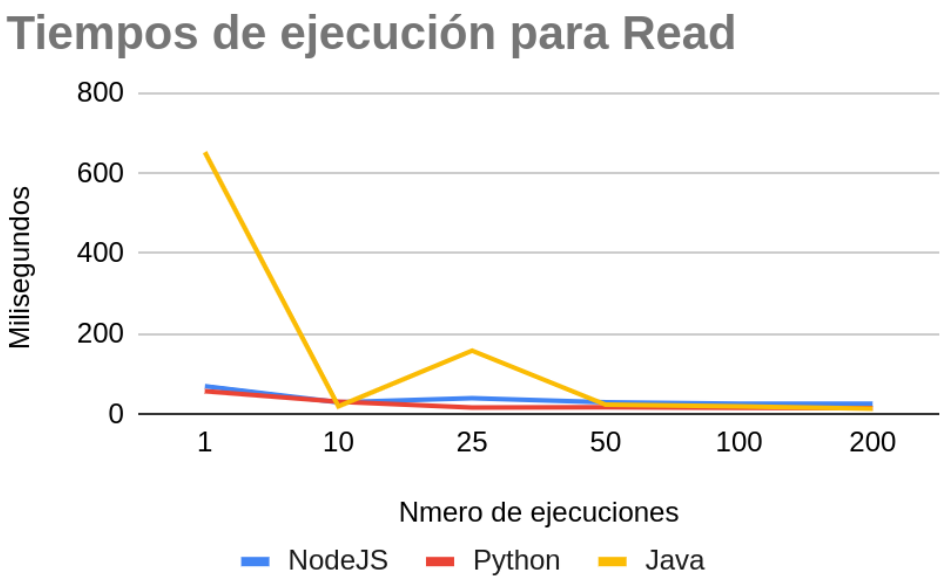


Figura 49. Tiempos de ejecución para la operación Read en cada lenguaje

4.4 Test de memoria

Con este test se pone a descubierto cómo afecta el monto de memoria asignado a una función lambda a sus tiempos de ejecución y a sus tiempos de inicialización. La plataforma AWS permite configurar la memoria de una función lambda hasta un máximo de 10 GB, pero no permite configurar la potencia del CPU sino que esta potencia es asignada según la memoria configurada. Como se mencionó anteriormente, mientras más memoria se coloque más potencia de CPU será asignada a la función lambda. Este test al igual que el de inicialización tiene una etapa de configuración que consiste de 3 fases. En la primera se hace una solicitud con el fin de calentar la infraestructura y de obtener datos relativos a los tiempos de inicialización. Luego en una segunda fase se hace una espera de un minuto para finalmente en la tercera hacer la solicitud que provocará una ejecución en caliente de la lambda ya que esta se encuentra en un estado caliente luego de la primera fase. En las fases 1 y 3 se ejecuta el mismo flujo configurado para todos los test detallados anteriormente. Este test fué ejecutado una vez por tamaño de memoria, es decir, se comenzó configurando las lambdas con un tamaño de 10 GB y se fue decrementando la memoria con los siguientes valores: 8, 6, 4, 2, 1 GB y finalmente se ejecutó con lambdas con 512 MB de tamaño de memoria.

4.4.1 Tiempos de ejecución, según memoria

Los resultados obtenidos en este test de memoria pueden observarse en la figura 50. En cuanto a tiempo de ejecución se puede ver que mientras menos memoria tiene asignada una función lambda mayores son los tiempos de ejecución de cada lenguaje. Este comportamiento es el esperado ya que AWS aprovisiona menos CPU a medida que la memoria asignada disminuye, por ende el

contenedor encargado de ejecutar la función dispondrá de recursos cada vez más reducidos. Es interesante observar que se encontró un punto en común de los lenguajes para la operación CRUD create en los 4096 MB donde los tiempos de ejecución de cada lenguaje son muy similares siendo NodeJS el de mejor performance con 42 ms, mientras que Python y Java obtuvieron un tiempo de 50 ms y 48 ms respectivamente. Un aspecto muy importante a tener en cuenta es la performance de Java cuando la memoria es menor a 4096 MB. En este caso, se puede apreciar cómo los tiempos de ejecución van aumentando y una vez que está por debajo de los 1024 MB el tiempo de ejecución aumenta considerablemente.

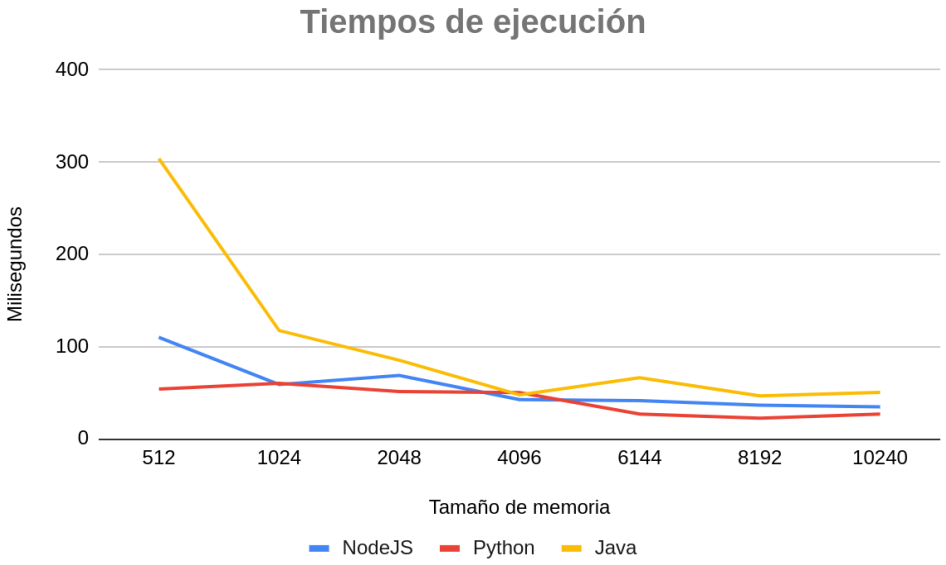


Figura 50. Tiempos de ejecución para create, para cada lenguaje y de acuerdo al tamaño de memoria

Con este test también se vuelve interesante hacer una observación sobre los tiempos de inicialización que aunque estos no sean importantes a la hora de hablar de facturación, si son de suma importancia para el tiempo de respuesta de las funciones lambda. Ya se sabe que cuando una función lambda es ejecutada en un estado frío, el tiempo de respuesta final de la función estará dado por la suma de su tiempo de inicialización más el tiempo de ejecución propiamente dicho. Dependiendo del aplicativo en el que se encuentre la función estos tiempos pueden ser críticos o no. En una función CRUD realmente no son muy críticos pero igualmente si las funciones tardan un par de segundos pueden afectar la experiencia de un usuario final al usar un sistema o aplicativo. El aprovisionamiento adecuado de memoria de una función lambda puede ser vital para lograr un equilibrio tanto del tiempo de ejecución como se vio anteriormente y también de los tiempos de inicializaciones de las funciones. Además mientras los tiempos de ejecución sean menores, menor será el monto facturado por AWS, de igual forma mientras mayor sea la memoría asignada mayor será el costo a pagar. Por estas razones encontrar la configuración adecuada de memoria puede brindar un beneficio en cuanto a costo y tiempo de ejecución del servicio lambda.

En la figura 51 se puede ver de forma muy clara que para NodeJS y Python el aprovisionamiento de memoria no afecta para nada los tiempos de inicialización de la lambda pero es importante tener en cuenta que si afecta los tiempos de ejecución como se vio anteriormente. Finalmente tenemos a Java como el gran perdedor cuando entra en consideración la inicialización de una lambda. A medida que la memoria disminuye de 10 GB a 6 GB se puede ver un rendimiento irregular en Java ya que por ejemplo se observa como los tiempos disminuyen de 1652 ms en 8 GB a 1346 ms en 6 GB pero a partir de este punto los tiempo de inicialización aumentan llegando a un tiempo máximo de 4000

ms para el tamaño de memoria de 512 MB lo cual es un performance muy malo para una operación CRUD.

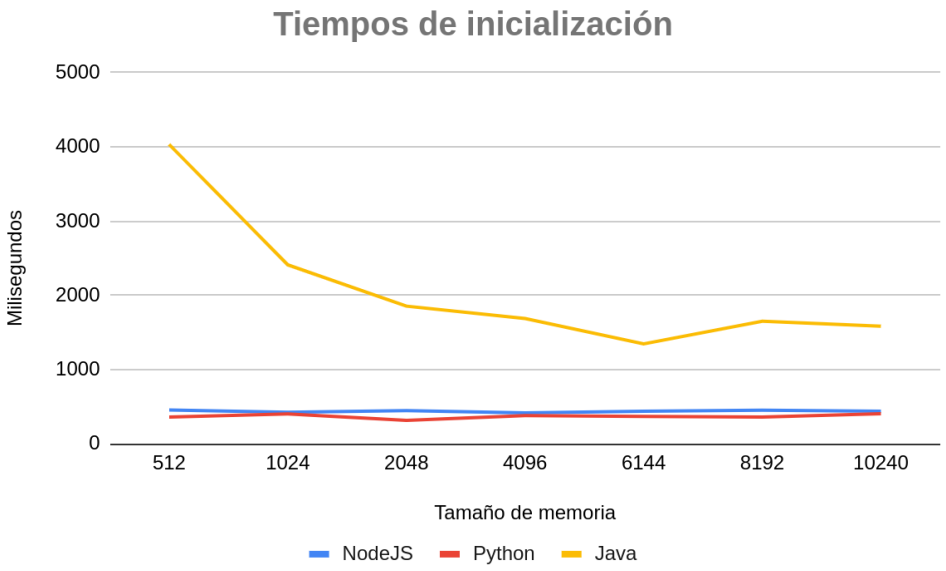


Figura 51. Tiempos de inicialización para create, para cada lenguaje y de acuerdo al tamaño de memoria

4.5 Conclusiones y trabajos futuros

En este trabajo se desarrolló una API Rest con el objetivo de realizar operaciones CRUD para la carga de un formulario y sus respuestas. Esta API fue desarrollada en tres lenguajes diferentes: Javascript con NodeJS, Python y Java. Con estas operaciones se investigó, en general, la variación de la performance, la latencia de los tiempos de inicialización, la escalabilidad y las implicaciones del tamaño de memoria configurado en las funciones. Se investigó cómo afecta la selección de los lenguajes de programación a la hora de realizar operaciones CRUD de una API Rest con una serie de test automáticos.

Entre los resultados logrados, se tiene que:

- El lenguaje de programación con mejores tiempos a la hora de levantar el contexto de ejecución de una función lambda es Python, obteniendo los mejores tiempos para cualquiera de las operaciones CRUD. De igual forma, las diferencias entre Python y NodeJS son realmente insignificantes. La elección entre cualquiera de estos dos lenguajes, teniendo en cuenta la inicialización de las lambdas, no será muy conflictiva. El lenguaje más deficiente fue Java, teniendo tiempos de inicialización muy grandes con respecto a los demás. Java a diferencia de NodeJS y Python debe levantar una máquina virtual y cargar el código de la aplicación en memoria, lo cual produce que demore su primera ejecución. La elección de Java para una API Rest serverless con Lambda puede ocasionar tiempos de respuesta lentos en caso de usarse en un sistema de poco o mediano tráfico. Para futuros trabajos se puede hacer hincapié en implementar optimizaciones para acotar los tiempos de inicialización de las funciones lambda, desarrolladas con Java.

- De acuerdo al test de inicialización también se pudo concluir en que Java, a pesar de ser el lenguaje que peor performance tiene en cuanto a tiempos de inicialización, es el que tiene un mayor rango de mejora de una ejecución en estado frío a una en estado caliente. La elección de este lenguaje puede estar justificada en un sistema de tráfico moderado-alto donde sea muy poco probable tener funciones lambdas en estado frío. Los rangos de mejora de NodeJS y Python son grandes porcentualmente pero poco significativos en un caso real. Debido a que los tiempos de ejecución bajan un porcentaje grande de estado frío a caliente, se debe tener en cuenta que mientras menos tarden las lambda, menos tiempo será facturado por AWS. Por ende el lenguaje con menor tiempo de ejecución será el más barato. Un caso interesante para trabajos futuros podría ser investigar acerca de las distintas alternativas para mantener las lambdas en estado caliente, siendo una de ellas ofrecida por AWS pero sumamente costosa.

- Para el caso de estudio, se obtuvo que Java fue el lenguaje que mejor escaló. Esto se debe como se mencionó anteriormente al inicio de su máquina virtual y la carga del código en memoria. Esto solo ocurre en la primera ejecución o solicitud por lo que las siguientes son mucho más rápidas y si la lambda se mantiene en estado caliente se llegará a una mejor performance, dado que su compilador observa el uso del código durante un período de tiempo para encontrar mejores optimizaciones.

- El lenguaje que peor se comportó a la hora de escalar fue NodeJS quedando en tercer lugar. Las diferencias para este caso de estudio no fueron realmente grandes como para descartar a éste como opción para el desarrollo.

- Con el test de memoria se pudo advertir que el lenguaje que más se ve afectado por el tamaño de memoria asignada al lambda es Java. A medida que se reduce la memoria aprovisionada, los tiempos de ejecución comienzan a subir pero no es totalmente cierto que mientras más memoria se asigne, mejores tiempos se obtienen. Esto se debe a que llega un punto en el que asignarle más memoria a una función lambda, no daŕa mejoras sino que los tiempos se mantendrán en un rango determinado. Como se mencionó, la mejora se debe a que AWS asigna más potencia de CPU mientras más memoria se le otorgue a la función. Con Java es de suma importancia prestar atención a esta característica ya que el tamaño de memoria adecuado puede servir para no tener costos elevados por memoria extra y alcanzar los mejores tiempos de ejecución que hará que las funciones demoren menos y que la facturación sea más reducida.

- El aprovisionamiento de memoria también es de gran importancia a la hora de mejorar los tiempos de inicialización de las funciones lambda. En este punto no es sustancial para la facturación ya que los tiempos de inicialización no son considerados por AWS. Esto es fundamental para el tiempo final de respuesta de una función, ya que la demora en la inicialización lo afecta directamente. Las funciones que demoran mucho en inicializarse pueden causar una mala experiencia a los usuarios finales que utilicen aplicaciones las cuales se vean afectadas en gran medida por esta causa.

- Mediante el test de memoria se pudo comprobar que los tiempos de inicialización de las lambdas implementadas en NodeJS y Python no se ven afectados por el aprovisionamiento de memoria.

- Para el desarrollo de una API Rest con AWS Lambda se puede concluir que la elección de Java como lenguaje de programación puede llevar a costos más elevados que NodeJS y Python. Lo dicho se debe a que es necesario un mayor aprovisionamiento de memoria para

tener tiempos más reducidos tanto de inicialización como de ejecución, además de invertir tiempo de desarrollo en búsqueda de optimizaciones tediosas. Aun así con todas estas diferencias, los tiempos de ejecución con respecto a NodeJS y Python son mínimas. En cuanto a la selección entre los anteriores dos lenguajes, la decisión de trabajar con alguno de ellos bajo un enfoque serverless pasará más que nada por el caso de uso específico, ya que ambos estuvieron bastante parejos a través de todos los tests. Para una API Rest sería sensato la elección de NodeJS sobre Python debido a que NodeJS está más orientado al desarrollo Web y a aplicaciones en tiempo real contando con una arquitectura basada en eventos. De igual forma se podría lograr un híbrido mediante la utilización de Python para la parte de análisis y visualización de datos, o procesamiento de imágenes, que a veces son comunes en el desarrollo de estas APIs. Por lo tanto podría decirse que no existe “la” combinación más conveniente para desarrollar una API Rest con operaciones CRUD. Quitando el análisis de datos y procesamiento de imágenes, lo más adecuado sería la utilización de NodeJS.

CAPÍTULO 5

Bibliografía

Artillery (s.f.) Artillery Docs [Online]. https://www.artillery.io/docs/guides/getting-started/core-concepts

Abiola, E. (2017, Aug). Should I learn Java, Javascript or Python? [Online]. https://medium.com/@Emmanuel.A/software-engineer-should-i-learn-java-javascript-or-python-part-1-6399fdbaa319

Al-Ameen, M., & Spillner, J. (2018). Systematic and open exploration of FaaS and Serverless Computing research. In *ESSCA@ UCC* (pp. 30-35).

Amazon CloudWatch, (s.f.). Types of metrics. [Online]. https://docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html#monitoring-metrics-types

Amazon CloudWatch, (s.f.). Dimensiones y métricas de DynamoDB [Online]. https://docs.aws.amazon.com/es_es/amazondynamodb/latest/developerguide/metrics-dimensions.html

Amazon Web Service, (s.f.). Características de AWS Lambda. [Online]. https://aws.amazon.com/es/lambda/features/

AWS, (s.f.). How do I reduce initialization and invocation duration latency for my Java Lambda function? https://aws.amazon.com/es/premiumsupport/knowledge-center/lambda-improve-java-function-performance/

AWS, (2022). Qué es Amazon DynamoDB. [Online]. https://docs.aws.amazon.com/es_es/amazondynamodb/latest/developerguide/Introduction.html

Becerra Urbina, J. C. (2019). Generador de código de funcionalidades tipo crud en la mantenibilidad de software aplicado a sistemas de información empresariales.

Bigyan Ghimire, (enero, 2021). Introducción a YAML para principiantes. Geekflare. https://geekflare.com/es/yaml-introduction/

Castillo, J. N., Garcés, J. R., Navas, M. P., Jácome Segovia, D. F., & Armas Naranjo, J. E. (2017). Base de Datos NoSQL: MongoDB vs. Cassandra en operaciones CRUD (Create, Read, Update, Delete). *Revista Publicando*, *4*(11(1), 79-107. Recuperado a partir de https://revistapublicando.org/revista/index.php/crv/article/view/398

Castro, P., Ishakian, V., Muthusamy, V., & Slominski, A. (2019). The rise of serverless computing. *Communications of the ACM*, *62*(12), 44-54.

Cattell, R. (2011). Scalable SQL and NoSQL data stores. Acm Sigmod Record, 39(4), 12-27.

Cimpanu, C. (2016, Jul). Today's Top 3 Programming Languages: Python, JavaScript, Java. [Online]. https://webscripts.softpedia.com/blog/today-s-top-3-programming-languages-javascript-python-java-506596.shtml

CNCF Serverless Working Group, 2018. CNCF WG-Serverless Whitepaper v1.0. Accessed 8.4.2019. https://github.com/cncf/wg-serverless/raw/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0. pdf.

Cordingly, R., Yu, H., Hoang, V., Perez, D., Foster, D., Sadeghi, Z., ... & Lloyd, W. J. (2020, August). Implications of programming language selection for serverless data processing pipelines. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)* (pp. 704-711). IEEE.Eroshenko, I. (2021). *Scaling Web Services for the Lightweight Collector* (Doctoral dissertation, Worcester Polytechnic Institute).

Fox GC, Ishakian V, Muthusamy V, Slominski A Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research. (2017) arXiv e-prints:1708–08028. http://arxiv.org/abs/1708.08028

Froufe, A. (1996). Tutorial de Java. [Online]. http://www.itlp.edu.mx/web/java/Tutorial%20de%20Java/Intro/carac

Hassan B, Saman A. Barakat2 & Qusay I. Sarhan2. Survey on serverless Computing (2021). Journal of Cloud Computing: Advances, Systems and Applications (2021) 10:39. https://doi.org/10.1186/s13677-021-00253-7

Ibero Tijuana - Universidad Iberoamericana Tijuana (2020) ¿Qué es la investigación aplicada y cuáles son sus principales características? https://blogposgrados.tijuana.ibero.mx/investigacion-aplicada/

Instituto Nacional de Estadística y Censos, (2022) [Online]. https://www.indec.gob.ar/ftp/cuadros/poblacion/Censo2022_cuestionario_viviendas_particulares.pdf

Jangda, A., Pinckney, D., Brun, Y., & Guha, A. (2019). Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, *3*(OOPSLA), 1-26.

Jonas E, Schleier-Smith J, Sreekanti V, Tsai C-C, Khandelwal A, Pu Q, Shankar V, Carreira J, Krauth K, Yadwadkar N, Gonzalez JE, Popa RA, Stoica I, Patterson. . Cloud Programming Simplified: A Berkeley View on Serverless Computing.(2019) http://arxiv.org/abs/1902.03383.

Martín, A. E., Chávez, S. B., Rodríguez, N. R., Valenzuela, A., & Murazzo, M. A. (2013, June). Bases de datos NoSQL en cloud computing. In *XV Workshop de Investigadores en Ciencias de la Computación*.

Meissner, D., Erb, B., Kargl, F., & Tichy, M. (2018, June). Retro-λ: An Event-sourced Platform for Serverless Applications with Retroactive Computing Support. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems* (pp. 76-87).

Moczurad, P., & Malawski, M. (2018, December). Visual-textual framework for serverless computation: a Luna Language approach. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* (pp. 169-174). IEEE.

MUS, M. (2019). Comparison between SQL and NoSQL databases and their relationship with big data analytics.

Node.js (s.f.). Entorno de ejecución para JavaScript. https://nodejs.org/es/

Paakkunainen, O. (2019). Serverless computing and FaaS platform as a web application backend.

Passwater, A., (2018). "2018 Serverless Community Survey: huge growth in serverless usage". Disponible en https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage

Pons, D. B., Ollobarren, A. R., Pinto, D. A., & López, P. G. (2018). Studying the feasibility of serverless actors. In *ESSCA@ UCC* (pp. 25-29).

Pulipaka, Ganapati (2016, Oct). An introduction to Amazon DynamoDB NoSQL database. [Online]. https://medium.com/@gp_pulipaka/an-introduction-to-amazon-dynamodb-nosql-database-96fb1982bb4a

Python. Active Python Releases. https://www.python.org/downloads/

Schulz, H., van Hoorn, A., & Wert, A. (2020). Reducing the maintenance effort for parameterization of representative load tests using annotations. *Software Testing, Verification and Reliability*, *30*(1), e1712. [Online]. https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1712

Serverless Framework Documentation. (s.f.), Serverless. https://www.serverless.com/framework/docs

TechGuy, The best programming languages to learn in 2022. (2022, Enero). [Online]. https://techguysfdc.medium.com/the-best-programming-languages-to-learn-in-2022-332340550fd7

w3schools (s.f.). Node.js Tutorial https://www.w3schools.com/nodejs/

Yoan. (s.f.). El Lenguaje de Programación Python, Pros y Contras. https://lovtechnology.com/lenguaje-programacion-python-pros-contras/

Yussupov, V., Breitenbücher, U., Leymann, F., & Wurster, M. (2019, December). A Systematic Mapping Study on Engineering Function-as-a-Service Platforms and Tools. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing* (pp. 229-240).