Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Programa de Maestría en Computación

Modelado y simulación de funciones en la nube en plataformas *Function-as-a-Service*

Tesis para optar por el grado de *Magíster Scientiae* en Computación, con énfasis en Ciencias de la Computación

Estudiante

Carlos Martín Flores González

Profesor Asesor

Ignacio Trejos Zelaya

Mayo, 2019

```
 Git: (HEAD -> master)

Branch: master

Tag:

Release:

Commit: 4b8f16b

Date: 2019-04-13 22:11:36 -0600

Author: Martin Flores

Email: martin.flores@bodybuilding.com

Committer: Martin Flores

Committer email: martin.flores@bodybuilding.com
```

Dedicatoria

Agradecimientos

Resumen

Abstract

Índice

Índice de figuras

Capítulo 1

Introducción

Los servicios de funciones en la nube (*Function-as-a-Service, FaaS*) representan una nueva tendencia de la computación en la nube en donde se permite a los desarrolladores instalar código, en forma de función, en una plataforma de servicios en la nube y en donde la infraestructura de la plataforma es responsable de la ejecución, el aprovisionamiento de recursos, monitoreo y el escalamiento automático del entorno de ejecución. El uso de recursos generalmente se mide con una precisión de milisegundos y la facturación es por usualmente 100 ms de tiempo de CPU utilizado.

En este contexto, el "código en forma de función" es un código que es pequeño, sin estado, que trabaja bajo demanda y que tiene una sola responsabilidad funcional. Debido a que el desarrollador no necesita preocuparse de los aspectos operacionales de la instalación o el mantenimiento del código, la industria empezó a describir este código como uno que no necesitaba de un servidor para su ejecución, o al menos de una instalación de servidor como las utilizadas en esquemas tradicionales de desarrollo, y acuñó el término *serverless* (sin servidor) para referirse a ello.

Serverless se utiliza entonces para describir un modelo de programación y una arquitectura en donde fragmentos de código son ejecutados en la nube sin ningún control sobre los recursos de cómputo en donde el código se ejecuta. Esto de ninguna manera es una indicación de que no hay servidores, sino simplemente que el desarrollador delega la mayoría de aspectos operacionales al proveedor de servicios en la nube. A la versión de *serverless* que utiliza explícitamente funciones como unidad de instalación se le conoce como *Function-as-a-Service*[1].

Aunque el modelo FaaS brinda nuevas oportunidades, también introduce nuevos retos. Uno de ellos tiene que ver con el rendimiento de la función, puesto que en este modelo solamente se conoce una parte de la historia, la del código, pero se omiten los detalles de la infraestructura que lo ejecuta. La información de esta infraestructura, su configuración y capacidades es relevante para arquitectos y diseñadores de software para lograr estimar el comportamiento de una función en plataformas FaaS.

El problema de la estimación del rendimiento de aplicaciones en la nube, como lo son las que se ejecutan en plataformas FaaS y arquitecturas basadas en microservicios, es uno de los problemas que está recibiendo mayor atención especialmente dentro de la comunidad de investigación en ingeniería de rendimiento de software. Se argumenta que a pesar de la importancia de contar con niveles altos de rendimiento, todavía hay una falta de enfoques de ingeniería de rendimiento que consideren de forma explícita las particularidades de los microservicios[2].

Si bien, para FaaS, existen plataformas *open source* por medio de las cuales se pueden obtener los detalles de la infraestructura y de esta manera lograr un mejor entendimiento acerca del rendimiento esperado, estas plataformas cuen-

tan con arquitecturas grandes y complejas, lo cual hace que generar estimación se convierta en una tarea sumamente retadora.

En este trabajo se plantea explorar la aplicación de modelado de rendimiento de software basado en componentes para funciones que se ejecutan en ambientes FaaS. Para esto se propone utilizar una función de referencia y, a partir de esta, generar cargas de trabajo para recolectar datos de la bitácora(*logs*) de ejecución y extraer un modelo a partir de ellos. Una vez que se cuente con un modelo, se procederá con su análisis y simulación a fin de evaluar si el modelo generado logra explicar el comportamiento de la función bajo las cargas de trabajo utilizadas.

Esta propuesta está organizada de la siguiente manera: en el capítulo **??** se presenta un marco conceptual sobre ingeniería de rendimiento de software y trabajos de investigación relacionados con ingeniería de rendimiento de software en aplicaciones en la nube, microservicios y *serverless*. En el capítulo **??** se define el problema a resolver. En el capítulo **??** se proporciona una justificación del proyecto desde las perspectivas de innovación, impacto y profundidad. El objetivo general y los objetivos específicos se plantean en el capítulo **??**. El alcance del proyecto se define en el capitulo **??**. Los entregables que se generarán a partir de esta propuesta se listan en el capítulo **??**. La metodología de trabajo se indica en el capítulo **??**. La propuesta concluye en el capítulo **??**, donde se presenta el cronograma de actividades.

Capítulo 2

Implementación de una función en la nube: manejador de imágenes

Uno de los principales problemas de hacer ingeniería de rendimiento para software en la nube es que no existen aplicaciones de referencia que hayan ganado popularidad o cuyo desarrollo se encuentre activo. A pesar de esto y de su reciente adopción, la industria ha empezado a reconocer casos de uso en donde las aplicaciones *serverless* encajan mejor. Amazon Web Services(AWS)[3] reconoce cinco patrones de uso predominantes en su servicio AWS Lambda:

1. Procesamiento de datos dirigidos por eventos.
2. Aplicaciones Web.
3. Aplicaciones móviles e Internet las cosas (IoT).
4. Ecosistemas de aplicaciones *serverless*.
5. Flujos de trabajo dirigidos por eventos.

Uno de las aplicaciones más comunes en *serverless* es desencadenar acciones luego de que ocurre un evento (1), por ejemplo luego de la modificación de un registro en una base de datos o bien luego de que se publica un mensaje en una cola de mensajería. Esto puede provocar que se active una función Lambda[1] que toma como entrada el evento recién publicado para su posterior procesamiento. Este estilo de caso de uso encaja bien en ambientes híbridos: ambientes en donde tecnologías *serverless* se aprovechan para realizar funciones específicas dentro de una aplicación (o aplicaciones) más grande.

AWS ha publicado una serie de arquitecturas de referencia[4] para su plataforma FaaS, AWS Lambda. Dentro de estas arquitecturas se destaca el caso de uso de un manejador de imágenes (*Image Handler*)[5].

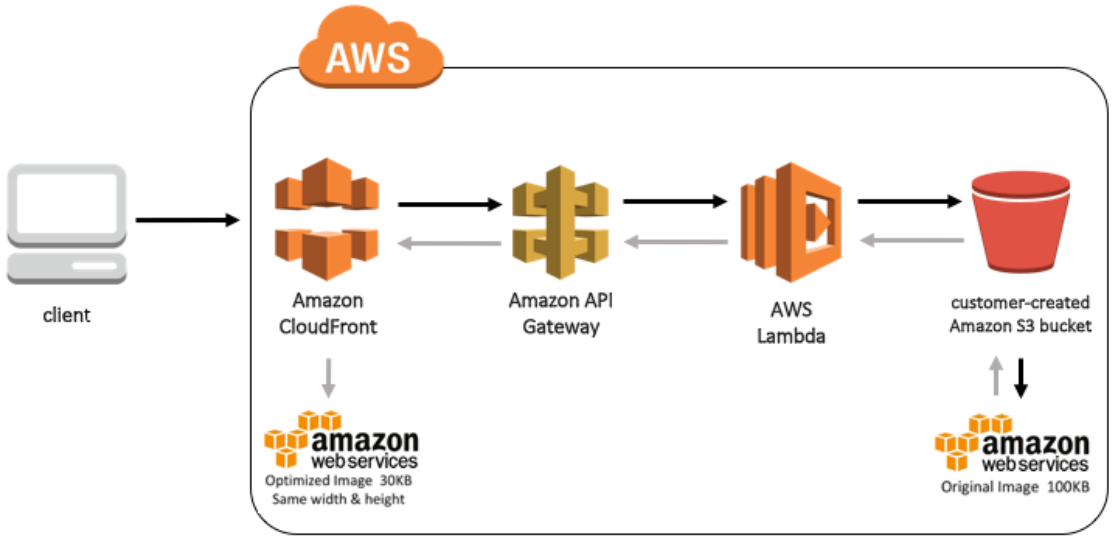

Figura 2.1: Arquitectura del manejador de imágenes. Tomado de [5]

[1] En la plataforma AWS Lambda

2.1. *Manejador de imágenes*

Sitios Web con imágenes grandes pueden experimentar tiempos de carga prolongados, es por esto que los desarrolladores proporcionan diferentes versiones de cada imagen para que se acomoden a distintos anchos de banda o diseños de página. Para brindar tiempos de respuesta cortos y disminuir el costo de la optimización, manipulación y procesamiento de las imágenes, AWS propone un manejador de imágenes *serverless*, al cual se le pueda delegar tal trabajo como una función Lambda sobre la plataforma FaaS.

A continuación se describe la arquitectura de la figura 2.1:

1. Amazon CloudFront provee una capa de *cache* para reducir el costo del procesamiento de la imagen

2. Amazon API Gateway brinda acceso por medio de HTTP a las funciones Lambda

3. AWS Lambda obtiene la imagen de un repositorio de Amazon Simple Storage Service (Amazon S3) y por medio de la implementación de la función se retorna una versión modificada de la imagen al API Gateway

4. El API Gateway retorna una nueva imagen a CloudFront para su posterior entrega a los usuarios finales

Cabe mencionar que, en este contexto, una versión modificada de una imagen será cualquier imagen que haya presentado algún tipo de alteración con respeto de una imagen original como, por ejemplo, cambios de tamaño, color, metadatos, etc.

2.1.1. Manejador de imágenes para SPE

Para este estudio se proponemos implementar una variación del manejador de imágenes de la sección 2.1, que se muestra en la figura 2.2.

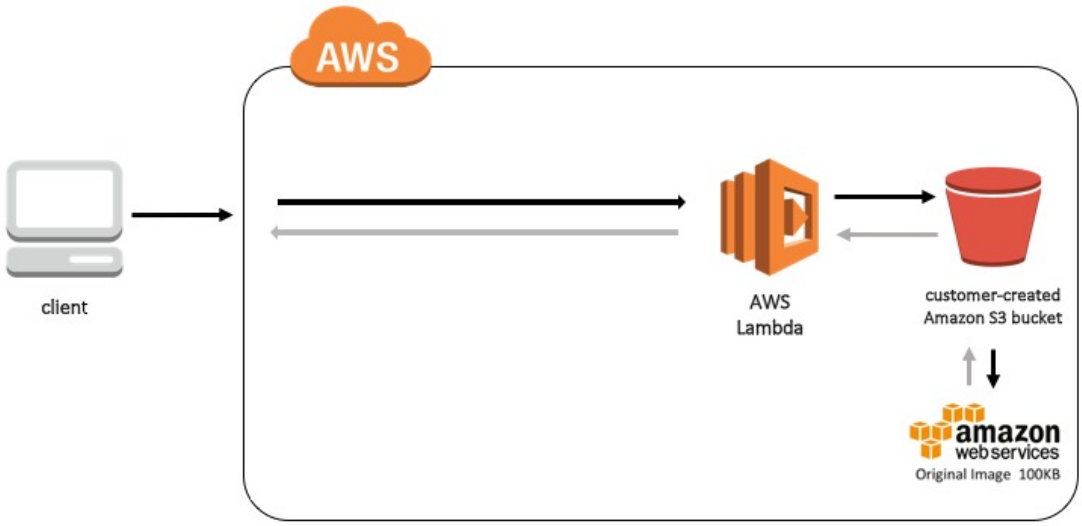

Figura 2.2: Arquitectura del manejador de imágenes propuesto para el estudio.

Se han dejado por fuera intencionalmente el AWS CloudFront y el AWS API Gateway. La razón de esto es porque se pretende ejercitar la función Lambda directamente. Se implementará una función Lambda que entregue a partir de una solicitud de redimensionamiento de una imagen almacenada, otra con dimensiones diferentes producida "al vuelo" como respuesta a la solicitud. Por ejemplo, si la imagen original mide 500 pixeles de ancho y alto, entregar una con dimensiones de 100 pixeles de ancho y alto.

Las actividades involucradas en el proceso de redimensionamientos de imágenes se muestran en la figura 2.3

1. Se envía una solicitud de redimensionamiento de imagen en formato `JSON` a la función Lambda con los datos acerca de la localización de la imagen y

su nuevo tamaño.

2. La solicitud de redimensionamiento llega a la función Lambda.

3. La función Lambda solicita al servicio de almacemiento AWS S3 la imagen.

4. AWS S3 entrega a la función Lambda la imagen solicitada.

5. La función Lambda inicia el redimensionamiento de la imagen de acuerdo a los parámetros solicitados.

6. La nueva imagen modificada se entrega al cliente(s).

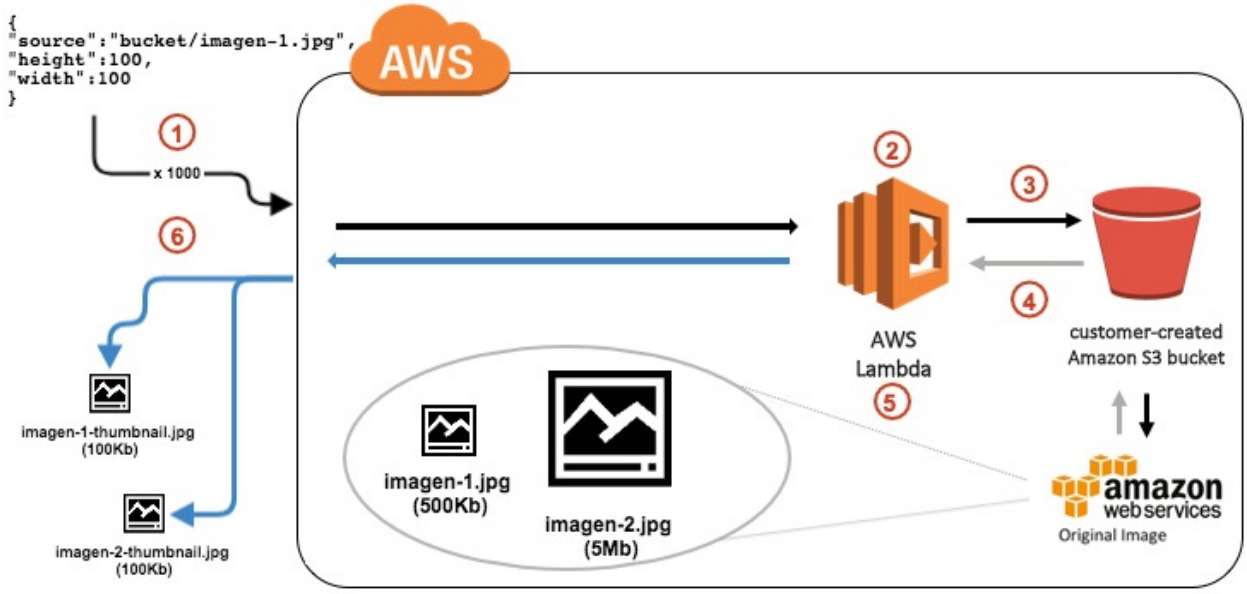

Figura 2.3: Carga de trabajo sugerida para el manejador de imágenes

A la función Lambda se le realizarán pruebas con imágenes de entrada de distinto tamaño y cargas de trabajo variables para evaluar su comportamiento bajo estos escenarios. Se desea observar el impacto de las pruebas en el tiempo de respuesta de la función. Los resultados obtenidos a partir de estas pruebas van a servir como un punto de referencia para experimentos futuros, como los

que se indican en la Sección **??**. La figura 2.3 muestra una sugerencia de dos posibles cargas de trabajo:

1. 100 solicitudes de cambio de tamaño de una imagen grande. En la figura 2.3, `imagen-2.jpg` de tamaño de 5Mb, representa una imagen grande.

2. 100 solicitudes de cambio de tamaño de una imagen pequeña. En la figura 2.3, `imagen-1.jpg` de tamaño menor o igual a 500Kb, representa una imagen pequeña.

En principio las cargas de trabajo generadas serían *cerradas*, lo que quiere decir que una solicitud se ejecuta solamente hasta que la anterior se termina. Esto ayudará en principio a tener mejor trazabilidad de lo que ocurre con la función.

¿Por qué este caso de uso se considera relevante? A continuación se listan las características que hacen este caso de uso representativo e interesante:

- Sencillo de entender e implementar: se cuenta únicamente con una función la cual lleva a cabo una tarea muy específica.

- Popular: sigue un patrón de procesamiento dirigido por eventos y, como se señala en [3], este es uno de los más populares que se ha empezado a adoptar para aplicaciones *serverless*. Otra de las razones de la popularidad de este caso de uso es que permite a los desarrolladores crear una unidad de instalación independiente y especializada para el manejo de imágenes, liberando así a sus servidores y aplicaciones del manejo de las peticiones y lógica asociadas a estas.

- Replicable en otros proveedores de servicios en la nube: varias de las arquitecturas de referencia para *serverless* propuestas por Amazon, están compuestas por herramientas y servicios muy propios de su plataforma, lo cual hace muy difícil su reproducibilidad utilizando otros proveedores. Aunque en principio este trabajo plantea ser elaborado en la plataforma FaaS de Amazon Web Services, AWS Lambda, otros proveedores de servicios (ver sección **??**) en la nube cuentan con sus propias plataformas de FaaS y de almacenamiento, lo cual permitiría replicar lo aquí propuesto en ellos.

- Replicable en los lenguajes de programación soportados por plataformas FaaS: actualmente JavaScript, Java (y lenguajes basados en la *Java Virtual Machine*), Python, C# y Go son los principales lenguajes de programación soportados por las plataformas FaaS. El caso de uso propuesto, no presenta ningún tipo de característica que lo ate a un lenguaje de programación en particular. En todos ellos se cuentan con bibliotecas para manejo de imágenes tanto de forma nativa como por medio de soluciones de terceros.

2.2. Implementación del *manejador de imágenes*

Existen soluciones disponibles que se pueden estudiar para implementar un manejador de imágenes. Amazon proporciona dos ejemplos que siguen la arquitectura de la figura 2.1:

1. **serverless-image-resizing**[2]: escrita en lenguaje JavaScript. Utiliza el mo-

[2] `https://github.com/amazon-archives/serverless-image-resizing`

dulo *sharp*[3] de NodeJS para aplicar operaciones de conversión en imágenes tales como redimensionamiento, rotación y corrección gamma.

2. **serverless-image-handler**[4]: escrita en lenguaje Python. Hace uso del paquete *Thumbor*[5] de código abierto para realizar operaciones de redimensionamiento, rotación, recorte y aplicación de filtros en imágenes.

A pesar que Amazon recomienda el uso de *serverless-image-handler* sobre *serverless-image-resizing*, ambas soluciones siguen un patrón sumamente similar en su codificación e instalación.

Otro ejemplo de una función en la nube encargada de ofrecer un servicio de redimensionamiento en imágenes, es la *Course_LambdaResizer*, una función lambda usada como referencia en el curso *"Serverless API on AWS for Java developers"* ofrecido en el sitio Web Udemy[6]. Está función está escrita en lenguaje Java y utiliza la biblioteca *imgscalr*[7] para redimensionar imágenes.

Para este estudio, se implementó una función escrita en lenguaje Java. Esto motivado principalmente por la compatibilidad de este lenguaje con las herramientas para monitoreo de aplicaciones y extracción de modelos de rendimiento, Kieker y PMX respectivamente.

2.2.1. Función Lambda: *Image-Handler*

La función Lambda creada para este estudio lleva por nombre *Image-Handler*. El código fuente y documentación relacionada con la misma se encuentra dis-

[3] https://github.com/lovell/sharp
[4] https://github.com/awslabs/serverless-image-handler
[5] http://thumbor.org
[6] https://www.udemy.com/serverless-api-aws-lambda-for-java-developers
[7] https://github.com/rkalla/imgscalr

ponible en GitHub.com, en el repositorio de código: `https://github.com/seminario-dos/image-handler`. El punto de entrada de la función Lambda es la clase `ImageHandler.java`. Esta función se encarga de realizar tres operaciones para procesar una solicitud de redimensionamiento de imagen:

1. Procesar la solicitud de redimiensionamiento (la entrada) que viende dada en formato JSON. Esta solicitud de redimensionamiento contiene entre otras cosas:
   - El nombre de la imagen original que reside en el servicio Amazon S3.
   - Los parámetros de altura y ancho a los que se desea redimensional la imagen original.
2. Obtener la imagen del servicio Amazon S3 y posteriormente aplicar la operación de redimiensionamiento sobre la misma de acuerdo a los parámetros de altura y ancho especificados en la solicitud de redimensionamiento.
3. Tomar la imagen redimensionada, codificarla en `Base64` y escribir el resultado en el flujo(*stream*) de salida de la función Lambda.

Un extracto de la clase `ImageHandler.java` se muestra en el listado 2.1. En la línea `22` se procesa el evento de entrada que viene dado en formato JSON. Como resultado de esto se entrega un objeto `ImageRequest` el cual contiene la información de la solicitud de la imagen que se desea redimensionar y que se encuentra alojada en el servicio Amazon S3.

En la línea `24` se llama al servicio `ImageService` con el fin de obtener la imagen original (de acuerdo a la información presente en el `ImageRequest` proporcionado) y se aplica la operación de redimensionamiento.

Por último, en la línea 26, `ImageHandlerResponseWriter.writeResponse()` toma la nueva imagen, con nuevas dimensiones de alto y ancho, la codifica en `Base64` y escribe el resultado en el *stream* de salida de la función.

```
1  public class ImageHandler implements RequestStreamHandler {
2
3      private static final AppConfig APP_CONFIG;
4      private final AppConfig appConfig;
5
6      static {
7          APP_CONFIG = AppConfig.getInstance();
8      }
9
10     public ImageHandler() {
11         this(APP_CONFIG);
12     }
13
14     public ImageHandler(AppConfig appConfig) {
15         this.appConfig = appConfig;
16     }
17
18     @Override
19     public void handleRequest(InputStream inputStream,
20                               OutputStream outputStream,
21                               Context context) throws IOException {
22         ImageRequest imageRequest =
23             this.inputEventParser().processInputEvent(inputStream);
24         InputStream imageResized =
25             this.imageService().getImageFrom(imageRequest);
26         this.imageHandlerResponseWriter()
27             .writeResponse(imageResized, outputStream, imageRequest);
28     }
29
30     private InputEventParser inputEventParser() {
31         return this.appConfig.getInputEventParser();
32     }
33
34     private ImageService imageService() {
35         return this.appConfig.getImageService();
36     }
37
38     private ImageHandlerResponseWriter imageHandlerResponseWriter() {
39         return this.appConfig.getImageHandlerResponseWriter();
40     }
41 }
```

Listing 2.1: Clase `ImageHandler.java`

Las funciones Lambda en AWS reciben como entrada un objeto JSON. Este objeto puede contener distintos campos dependiendo del servicio que haya

invocado previamente la ejecución de la función Lambda. Debido a que la función *Image-Handler* pretende ser invocada por medio de solicitudes `HTTP`, esta se configuró para que trabajara en conjunto con el servicio `API Gateway`. Dentro de este servicio se creó un recurso Web que entrega solicitudes de tipo `HTTP GET` a la función Lambda para su posterior procesamiento.

En términos generales, cada vez que una solicitud `HTTP GET` ingresa al `API Gateway` con el siguiente formato `https://{host}/image/{image}?width=&height=`, se tomarán el nombre de la imagen original que viene en el parámetro `image` y los parámetros de ancho y alto, `width` y `height` respectivamente, y se pasarán como parámetros de entrada a la función Lambda como parte de un objeto JSON. Este objeto JSON contiene otros campos que dan a conocer a la función Lambda información acerca de la solicitud `HTTP`.

Ejemplo: para la siguiente solicitud `HTTP`:

```
GET https://{host}/images/original-pic.jpg?width=50&height=66
```

`API Gateway` produce el objeto JSON listado en 2.2. A pesar que el objeto JSON incluye otros campos, para efectos del *Image-Handler* solamente tres de ellos serán utilizados:

1. `pathParameters`: contiene el nombre de la imagen original a ser redimensionada.
2. `isBase64Encoded`: señala si la solicitud necesita ser codificada en `Base64` o no.
3. `queryStringParameters`: bajo esta propiedad se listan los parámetros de ancho(`width`) y alto(`height`).

```
{
  "headers": {
    "Accept": "*/*",
    "User-Agent": "HTTPie/1.0.2",
    "Connection": "keep-alive",
    "X-Forwarded-Proto": "http",
    "Host": "localhost:3000",
    "Accept-Encoding": "gzip, deflate",
    "X-Forwarded-Port": "3000"
  },
  "pathParameters": {
    "image": "original-pic.jpg"
  },
  "path": "/images/original-pic.jpg",
  "isBase64Encoded": true,
  "requestContext": {
    "accountId": "123456789012",
    "path": "/images/{image+}",
    "resourceId": "123456",
    "stage": "prod",
    "requestId": "c6af9ac6-7b61-11e6-9a41-93e8deadbeef",
    "identity": {
      "cognitoIdentityPoolId": null,
      "accountId": null,
      "caller": null,
      "apiKey": null,
      "sourceIp": "127.0.0.1",
      "cognitoAuthenticationType": null,
      "cognitoAuthenticationProvider": null,
      "userArn": null,
      "userAgent": "Custom User Agent String",
      "user": null
    },
    "resourcePath": "/images/{image+}",
    "httpMethod": "GET",
    "extendedRequestId": null,
    "apiId": "1234567890"
  },
  "resource": "/images/{image+}",
  "httpMethod": "GET",
  "body": null,
  "queryStringParameters": {
    "width": "50",
    "height": "66"
  },
  "stageVariables": null
}
```

Listing 2.2: Clase `ImageHandler.java`

Principales interacciones dentro de *Image-Handler*

La figura 2.4 muestra las principales interacciones que lleva a cabo la función *Image-Handler*. Tanto las acciones como los actores involucrados, concuerdan con lo descrito en la Sección 2.2.1 aunque, a diferencia de lo descrito allí, aquí se presenta la clase `S3Dao` que es la que se encarga de buscar y traer la imagen original del servicio Amazon S3.

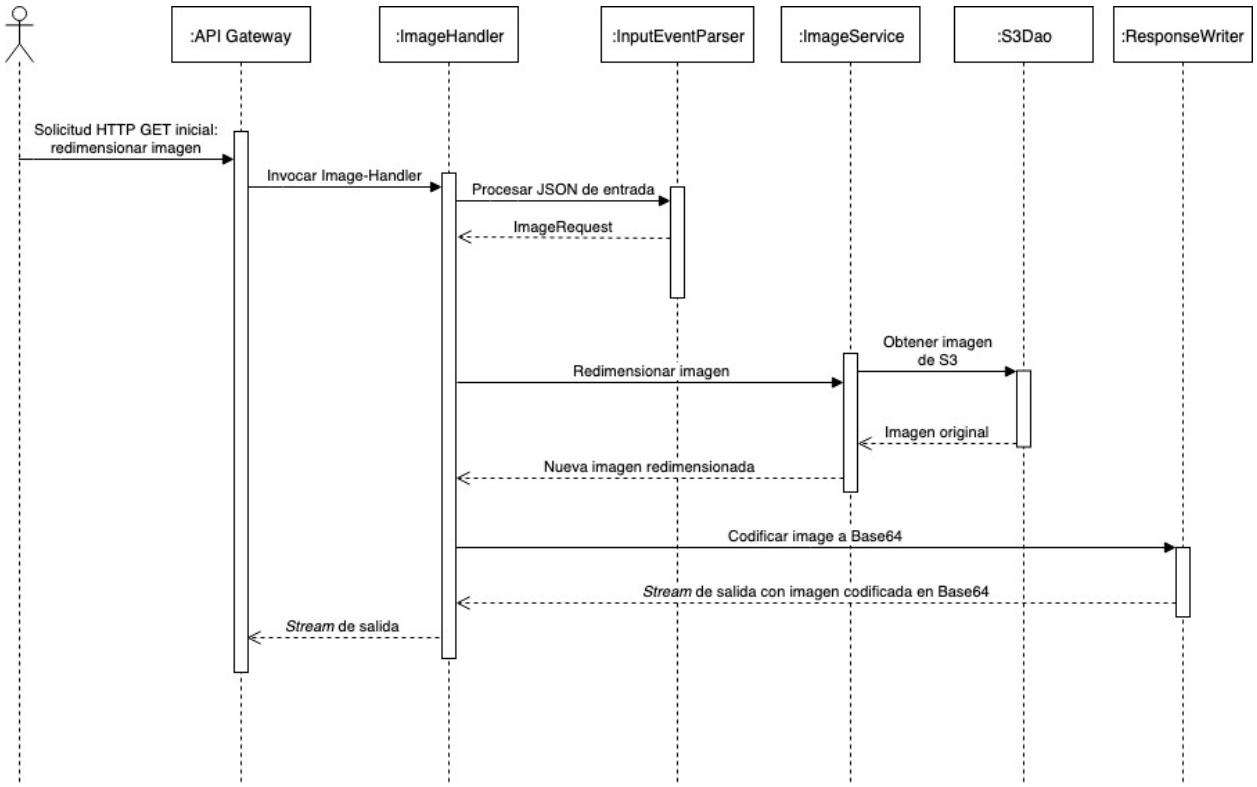

Figura 2.4: Secuencia de acciones llevadas a cabo por *Image-Handler*

2.2.2. Versiones alternas de *Image-Handler*

Aparte de la versión original de *Image-Handler*, se crearon dos versiones alternas con el fin de instrumentalizar el código fuente para la extracción y generación de un modelo en PCM a partir, y para la obtención de mediciones del

rendimiento.

Versión instrumentalizada para Kieker y PMX

Uno de los principales objetivos de este trabajo es el de obtener un modelo de rendmiento a partir del código en ejecución de una función en la nube. A pesar que, el modelo de rendimiento puede ser creado por los diseñadores e implementadores sin necesidad de una herramienta que ayude a su extracción, el uso de una herramienta especializada para esto contribuye a la generación de un modelo que pueda incluir mayores niveles de detalle en cuanto a la estructura y estimaciones del comportamiento de un software. Además, debido a que se están dando los primeros pasos en el campo del modelado y simulación de rendimiento basado en componentes, es preferible delegar tareas de extracción y estimaciones a una herramienta(s) con el fin de aprender de lo resultados obtenidos e ir introduciendo cambios paulatinamente; la creación manual de modelos de rendimiento puede llegar a ser muy compleja, consumir mucho tiempo y propensa a errores.

Kieker es la herramienta seleccionada para el estudiar el comportamiento del rendimiento del sistema a partir de las entradas/rastros que se registran en una bitácora. Creada y mantenida por el grupo de ingeniería de software de la Universidad de Kiel en conjunto con el grupo de sistemas de software de la Universidad de Stuttgart, Kieker se define como una herramienta para monitoreo del rendimiento y análisis dinámico de aplicaciones y se desarolló como parte de las actividades de enseñanza e investigación de ambas universidades. Otras instituciones académicas y de la industria también han realizado contribuciones.

Kieker ofrece adaptadores de monitoreo (*monitoring adapters*) escritos en lenguaje Java, aunque también ofrece adaptadores en otros lenguajes. Los dos principales componentes de Kieker son: `Kieker.Monitoring` y `Kieker.Analysis`. `Kieker.Monitoring` es el responsable de la instrumentación del código, recolección de datos y registro(*logging*). El componente `Kieker.Analysis` es el responsable de leer, analizar y visualizar los datos monitoreados.

Performance Model Extractor (PMX), es una herramienta que automatiza la extracción de modelos de rendmiento a partir de mediciones. PMX utiliza como entrada las bitácoras basadas en Kieker y es capaz de crear modelos basados en *Palladio Component Model* a partir de estas. Esta es una herramienta creada y mantenida por el grupo de ingeniería de software de la Universidad Würzburg en Alemania.

En conjunto, Kieker y PMX, proporcionan un marco de trabajo por medio del cual se puede obtener mediciones del rendimiento de una aplicación y luego, a partir de estas, extraer un modelo de rendimiento basado en PCM. La selección de estas herramientas y el enfoque de medición y extracciòn de modelos de rendimiento se seleccionó luego de estudiar el *enfoque de ingeniería de rendimiento declarativo* propuesto en [6].

Estrategia de extracción de modelo de rendmiento para *Image-Handler* Debido a que las funciones Lambda se ejecutan en contenedores que son tanto inaccesibles como efímeros para los diseñadores e implementadores, estrategias tradicionales en donde se crean bitácoras en la misma computadora que ejecuta la aplicación y se van monitoreando utilizando alguna herramienta especializada o mediante *Secure Socket Channel*(`SSH`) deben ser replanteadas. Para este tipo de software, se hace necesario registrar los eventos asociados al

comportamiento del rendimiento en una computadora que se tenga control.

Para lograr esto, se instrumentalizó el código fuente para que:

1. Generara registros del rendimiento de la ejecución utilizando las bibliotecas proporcionadas por Kieker, de acuerdo con su manual de usuario[7]. Se crean objetos de tipo `OperationExecutionRecord` los cuales contienen información acerca del rendimiento de una invocación sobre alguna parte del código, y luego estos objetos se publican a una cola de mensajería para su posterior tratamiento. En el listado 2.3 se puede ver un extracto del código de la función instrumentalizada para que genere objetos `OperationExecutionRecord`.

2. En lugar de publicar los registros en alguna bitácora local, los registros del rendimiento se publican en una cola *Java Message Service*(JMS).

3. Una vez que arriban los mensajes a la cola JMS, otra aplicación, un consumidor de mensajes, procesa los mensajes de la cola y los va almacenando en una bitácora.

```
1  public class ImageHandlerKieker implements RequestStreamHandler {
2      private static final IMonitoringController MONITORING_CONTROLLER;
3      static {
4          MONITORING_CONTROLLER = MonitoringController.getInstance();
5      }
6      .
7      .
8      .
9
10     @Override
11     public void handleRequest(InputStream inputStream, OutputStream
       outputStream, Context context) throws IOException {
12
13         final long tin = MONITORING_CONTROLLER.getTimeSource().getTime
       ();
14         handleRequestInternal(inputStream, outputStream, context);
15         final long tout = MONITORING_CONTROLLER.getTimeSource().getTime
       ();
16         final OperationExecutionRecord e = new OperationExecutionRecord
       ("public void "+ this.getClass().getName()+".handleRequest(
       InputStream, OutputStream, Context)",
```

```
                OperationExecutionRecord.NO_SESSION_ID,
                OperationExecutionRecord.NO_TRACE_ID,
                tin, tout,
                InetAddress.getLocalHost().getHostName(),
                0,
                0);
        MONITORING_CONTROLLER.newMonitoringRecord(e);
    }
    .
    .
    .
```

Listing 2.3: Extracto de la clase `ImageHandler.java` instrumentalizada con Kieker

La figura 2.5 muestra los involucrados en el proceso de publicación de mediciones de rendimiento del código de *Image-Handler* hacia una bitácora externa.

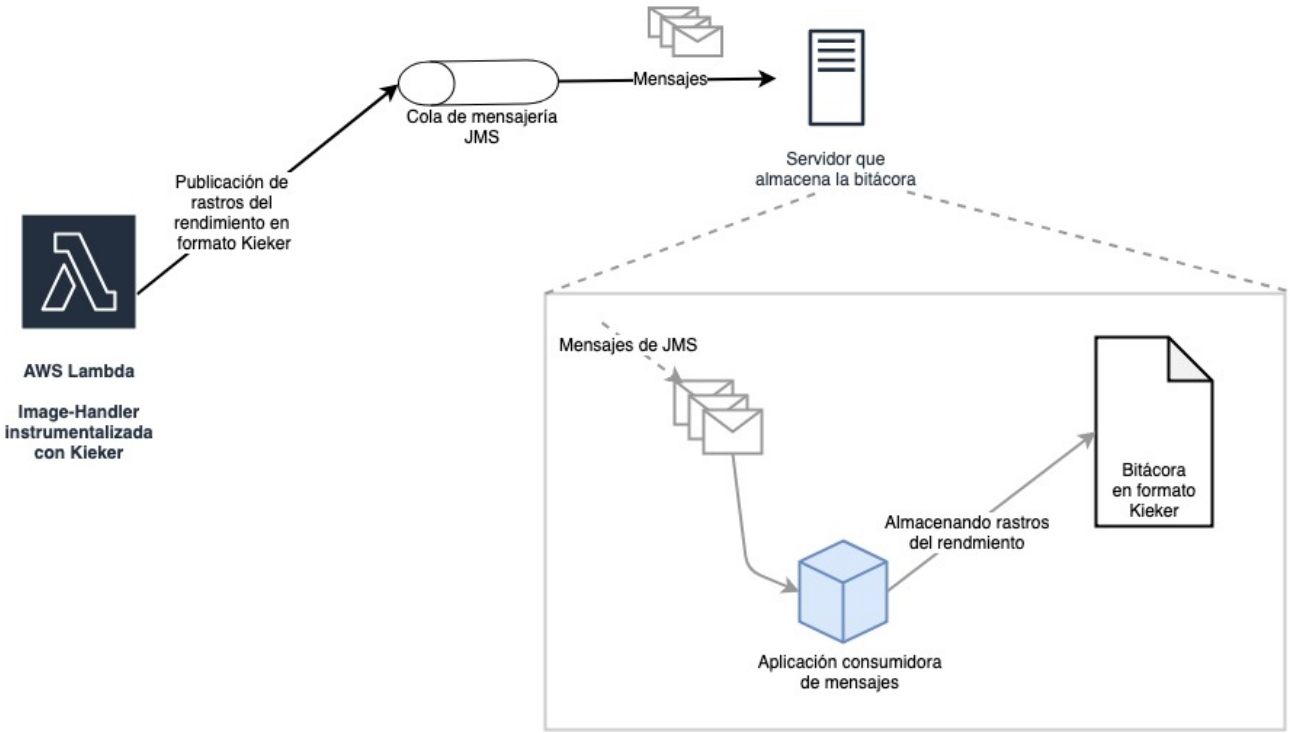

Figura 2.5: Publicando mediciones del rendimiento de la función Lambda.

Una vez que la función Lambda haya sido ejercitada y se obtenga una bitácora de Kieker, se toma esta bitácora como entrada para PMX, con el fin de obtener una instancia de PCM. Esto se aprecia en la figura 2.6. Se espero que, al utilizarse

datos de mediciones directas del rendimiento de la función Lambda, PMX pueda generar una instacia de PCM que sea confiable y de la que se puedan obtener simulaciones que representativas del comportamiento de la función Lambda.

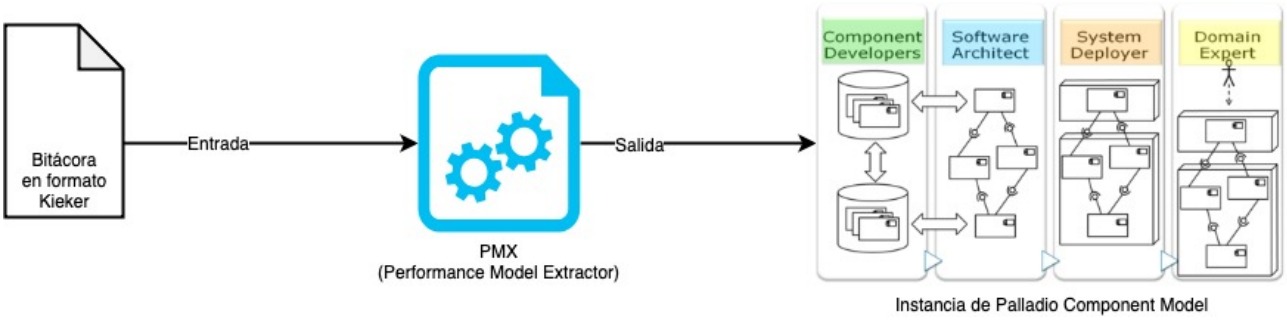

Figura 2.6: Convirtiendo una bitácora de Kieker a una instancia de PCM por medio de PMX.

Bibliografía

[1] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," *CoRR*, vol. abs/1706.03178, 2017.

[2] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance engineering for microservices: Research challenges and directions," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, (New York, NY, USA), pp. 223–226, ACM, 2017.

[3] M. Boyd, "Serverless architecture: Five design patterns," March 2017. `https://thenewstack.io/serverless-architecture-five-design-patterns/`.

[4] A. W. Services, "AWS lambda - resources: Reference architectures," 2018. `https://aws.amazon.com/lambda/resources/reference-architectures/`.

[5] A. W. Services, "Serverless image handler – AWS answers," 2018. `https://aws.amazon.com/answers/web-applications/serverless-image-handler/`.

[6] J. Walter, S. Eismann, J. Grohmann, D. Okanovic, and S. Kounev, "Tools for declarative performance engineering," in *Companion of the 2018 ACM/SPEC*

International Conference on Performance Engineering, ICPE '18, (New York, NY, USA), pp. 53–56, ACM, 2018.

[7] K. Project, "Kieker 1.13 user guide," Oct 2017. http://kieker-monitoring.net/documentation/, obtenido el 15 de Abril del 2019.