

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: 97af827

Date: 2019-05-24 23:55:41 -0600

Author: Martin Flores

Email: martin.flores@bodybuilding.com

Committer: Martin Flores

Committer email: martin.flores@bodybuilding.com

Dedicatoria

Agradecimientos

Resumen

Abstract

Índice

1. Introducción	1
2. Implementación de una <i>FaaS</i>: manejador de imágenes	4
2.1. <i>Manejador de imágenes</i>	6
2.1.1. Manejador de imágenes para SPE	7
2.2. Implementación del <i>manejador de imágenes</i>	10
2.2.1. Función Lambda: <i>Image-Handler</i> (IM-Simple)	11
Principales interacciones dentro de <i>Image-Handler</i>	16
2.2.2. Versiones alternas de <i>Image-Handler</i>	16
Versión instrumentalizada para Kieker y PMX (IM-KP) . . .	17
Versión instrumentalizada para AWS X-Ray (IM-XRay) . . .	20
2.3. Estrategia de extracción de modelo de rendimiento para <i>Image-Handler</i>	22
Versión instrumentalizada para AWS X-Ray	24

2.4. Diseño Experimental	26
2.4.1. Utilizando <i>Image-Handler</i> para redimensionar imágenes de distintos tamaños	26
Ejercitando la función Lambda con solicitudes de redimen- sionamiento en una imagen pequeña(~100Kb) .	27
Bibliografía	41

Índice de figuras

2.1. Arquitectura del manejador de imágenes	5
2.2. Arquitectura del manejador de imágenes propuesto para el estudio	7
2.3. Carga de trabajo sugerida para el manejador de imágenes	8
2.4. Secuencia de acciones llevadas a cabo por <i>Image-Handler</i>	16
2.5. <i>Image-Handler</i> publicando eventos de rendimiento al servicio AWS	
X-Ray	22
2.6. Publicando mediciones del rendimiento de la función Lambda. . .	24
2.7. Convirtiendo una bitácora de Kieker a una instancia de PCM por	
medio de PMX.	25
2.8. <i>Image-Handler</i> publicando eventos de rendimiento al servicio AWS	
X-Ray	29

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 capítulo ?. 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 Lambda¹ 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].

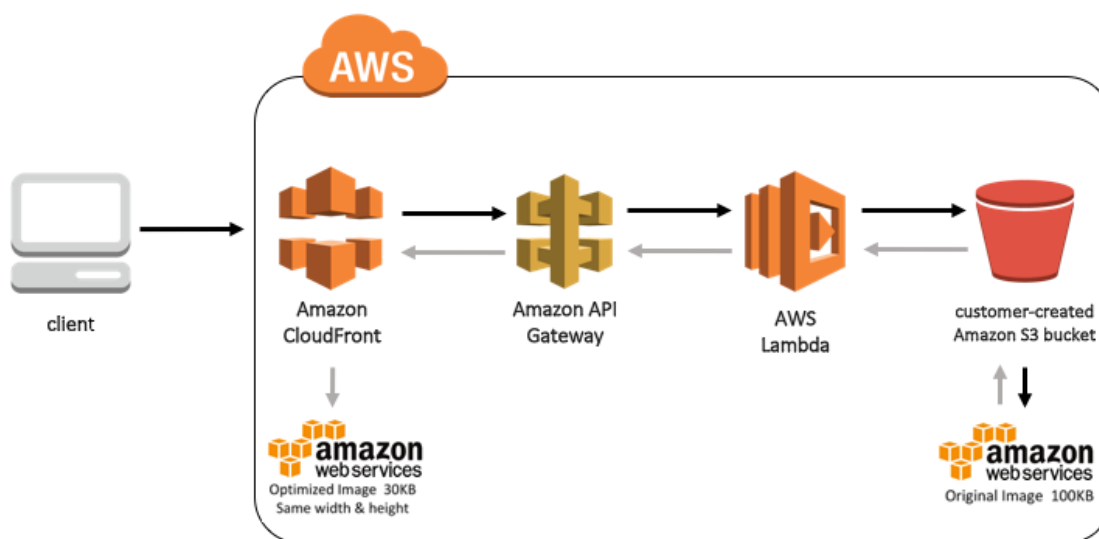


Figura 2.1: Arquitectura del manejador de imágenes. Tomado de [5]

¹En 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 respecto 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.

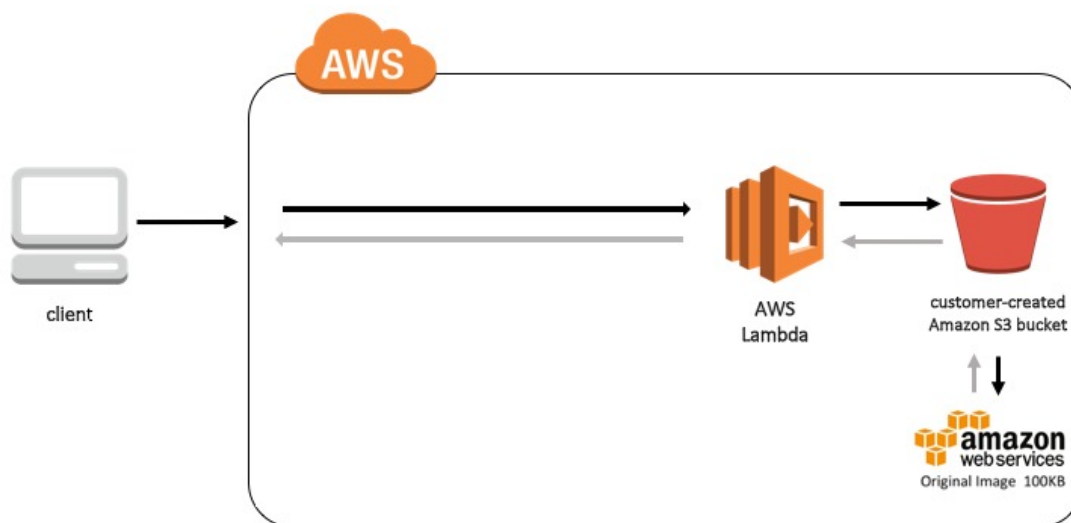


Figura 2.2: Arquitectura del manejador de imágenes propuesto para el estudio.

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 almacenamiento 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).

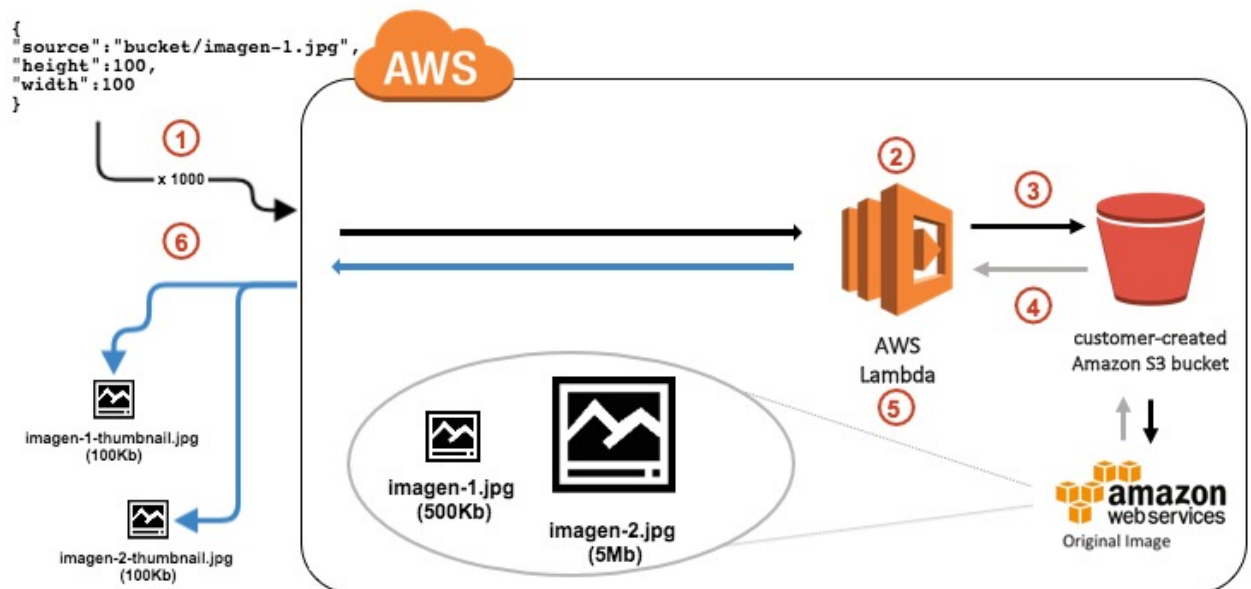


Figura 2.3: Carga de trabajo sugerida para el manejador de imágenes

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 Udemy⁶. Esta 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* (IM-Simple)

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/aws-labs/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 redimensionamiento (la entrada) que viene 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 redimensionar la imagen original.
2. Obtener la imagen del servicio Amazon S3 y posteriormente aplicar la operación de redimensionamiento 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={value}&height={value}
```

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).

```
1 {
2   "headers": {
3     "Accept": "*/*",
4     "User-Agent": "HTTPIe/1.0.2",
5     "Connection": "keep-alive",
6     "X-Forwarded-Proto": "http",
7     "Host": "localhost:3000",
8     "Accept-Encoding": "gzip, deflate",
9     "X-Forwarded-Port": "3000"
10  },
11  "pathParameters": {
12    "image": "original-pic.jpg"
13  },
14  "path": "/images/original-pic.jpg",
15  "isBase64Encoded": true,
16  "requestContext": {
17    "accountId": "123456789012",
18    "path": "/images/{image+}",
19    "resourceId": "123456",
20    "stage": "prod",
21    "requestId": "c6af9ac6-7b61-11e6-9a41-93e8deadbeef",
22    "identity": {
23      "cognitoIdentityPoolId": null,
24      "accountId": null,
25      "caller": null,
26      "apiKey": null,
27      "sourceIp": "127.0.0.1",
28      "cognitoAuthenticationType": null,
29      "cognitoAuthenticationProvider": null,
30      "userArn": null,
31      "userAgent": "Custom User Agent String",
32      "user": null
33    },
34    "resourcePath": "/images/{image+}",
35    "httpMethod": "GET",
36    "extendedRequestId": null,
37    "apiId": "1234567890"
38  },
39  "resource": "/images/{image+}",
40  "httpMethod": "GET",
41  "body": null,
42  "queryStringParameters": {
43    "width": "50",
44    "height": "66"
45  },
46  "stageVariables": null
47 }
```

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.

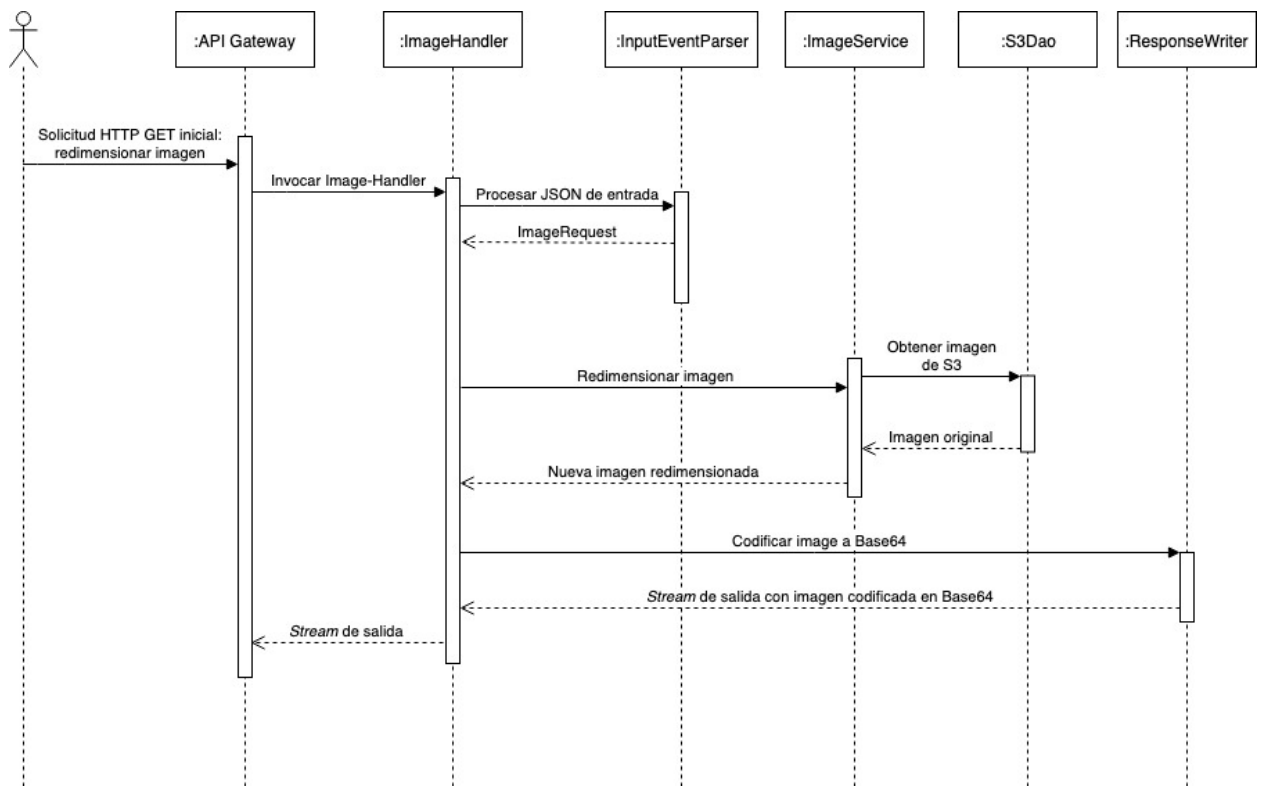


Figura 2.4: Secuencia de acciones llevadas a cabo por *Image-Handler*

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.

En la primer version, el código se modificó para generar rastros del rendimiento de la función Lambda y extraer a partir de estos un modelo de rendimiento en PCM. La segunda versión fue modificada para generar rastros de rendimiento pero, a diferencia de la versión anterior, los datos obtenidos en esta versión fueron de mayor utilidad para afinar las estimaciones de rendimiento de los componentes involucrados en la función.

Versión instrumentalizada para Kieker y PMX (IM-KP)

Uno de los principales objetivos de este trabajo es el de obtener un modelo de rendimiento 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 los resultados obtenidos e ir introduciendo cambios paulatinamente; la creación manual de modelos de rendimiento puede llegar a ser muy compleja, consumir mucho tiempo y ser propensa a errores.

Para lograr esto, se seleccionaron dos herramientas, Kieker y PMX, las cuales en conjunto 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 extracción de modelos de rendimiento se seleccionó luego de estudiar el *enfoque de ingeniería de rendimiento declarativo* propuesto en [6].

Kieker se utiliza para el estudiar el comportamiento del rendimiento del sistema a partir de las entradas/rastros que se registran en una bitácora. Kieker ofrece adaptadores de monitoreo (*monitoring adapters*) escritos en lenguaje Java (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.

En esta versión de *Image-Handler* se modificó el código original para generar registros del rendimiento de la ejecución de la función utilizando las bibliotecas proporcionadas por Kieker, utilizando como referencia lo especificado en el manual de usuario de Kieker[7]. Se crean objetos de tipo `OperationExecutionRecord` los cuales son los que contienen la información acerca del rendimiento de una invocación sobre alguna parte del código. En el listado 2.3 se puede ver un extracto del código de la función instrumentalizada para que genere objetos `OperationExecutionRecord`.

Adicionalmente se configuró la biblioteca para que publique los objetos `OperationExecutionRecord` a modo eventos a una cola de mensajería *Java Message Service* (JMS), en lugar de una bitácora local.

```
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
12     outputStream, Context context) throws IOException {
13
14         final long tin = MONITORING_CONTROLLER.getTimeSource().getTime
15         ();
16         handleRequestInternal(inputStream, outputStream, context);
17         final long tout = MONITORING_CONTROLLER.getTimeSource().getTime
18         ();
19         final OperationExecutionRecord e = new OperationExecutionRecord
20         ("public void "+ this.getClass().getName()+".handleRequest(
21         InputStream, OutputStream, Context)",
22         OperationExecutionRecord.NO_SESSION_ID,
23         OperationExecutionRecord.NO_TRACE_ID,
24         tin, tout,
25         InetAddress.getLocalHost().getHostName(),
26         0,
27         0);
28         MONITORING_CONTROLLER.newMonitoringRecord(e);
29     }
30     .
31     .
32     .
33 }

```

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

Performance Model Extractor (PMX), es una herramienta que automatiza la extracción de modelos de rendimiento 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.

Los aspectos relacionados con la estrategia de cómo se obtuvo un modelo de rendimiento a partir de las bitácoras de Kieker y PMX se dan a conocer en la Sección 2.3.

Versión instrumentalizada para AWS X-Ray (IM-XRay)

La principal motivación detrás de esta nueva versión de *Image-Handler* es la de contar con datos del rendimiento de la función Lambda que no pudieron llegar a ser estimados durante el proceso de extracción del modelo utilizando PMX. En la Sección 2.3 se brindan los detalles de lo observado durante el proceso de extracción del modelo con PMX y en dónde encajan los resultados arrojados por esta nueva versión en el modelo.

Para esta versión, Se siguió un enfoque similar al de la Sección 2.2.2 pero en lugar de utilizar la biblioteca de Kieker, se utilizó la biblioteca AWS SDK (*Software Development Kit, SDK*) para crear las trazas y *subsegmentos* de trazas, que son vistas más específicas del comportamiento de la aplicación. Con el uso AWS X-Ray se busca:

1. Obtener datos específicos del rendimiento de la función.
2. Averiguar si las nuevas mediciones logran brindar información acerca de la infraestructura AWS Lambda y su impacto en la ejecución de *Image-Handler*.
3. Exportar los datos de rendimiento a algún formato conocido para su manipulación.

Se modificó la función *Image-Handler* para que puede generar trazas de AWS X-Ray en los mismos puntos en el código en los que se agregó la instrumentalización para Kieker. Un extracto de este código se muestra en el listado 2.4. Cada invocación a las operaciones de procesamiento de la entrada, redimensionamiento y entrega de la respuesta se realizan utilizando el método `AWSRay.createSubsegment`.

```

1 public class ImageHandlerXRay implements RequestStreamHandler {
2     .
3     .
4     .
5     @Override
6     public void handleRequest(InputStream inputStream, OutputStream
7     outputStream, Context context) throws IOException {
8         ImageRequest imageRequest = AWSXRay.createSubsegment("input
9     event", new Function<Subsegment, ImageRequest>() {
10        @Override
11        public ImageRequest apply(Subsegment subsegment) {
12            return inputEventParser().processInputEvent(inputStream
13        );
14        }
15    });
16
17    InputStream imageResized = AWSXRay.createSubsegment("resize
18    event", new Function<Subsegment, InputStream>() {
19        @Override
20        public InputStream apply(Subsegment subsegment) {
21            return imageService().getImageFrom(imageRequest);
22        }
23    });
24
25    AWSXRay.createSubsegment("write response", () -> {
26        imageHandlerResponseWriter().writeResponse(imageResized,
27        outputStream, imageRequest);
28    });
29
30    .
31    .
32    .
33 }

```

Listing 2.4: Extracto de la clase `ImageHandler.java` instrumentalizada con AWS X-Ray

En la figura 2.5 se muestran los principales involucrados en la generación de trazas de rendimiento en AWS X-Ray. A la función *Image-Handler* se le agrega la biblioteca de AWS X-Ray para crear las trazas. Estas trazas se envían al servicio AWS X-Ray que es el recolecta estas trazas y con base en ellas se pueden obtener mapas de la interacción de los servicios que componen la función y desgloses de los subsegmentos que componen la traza.

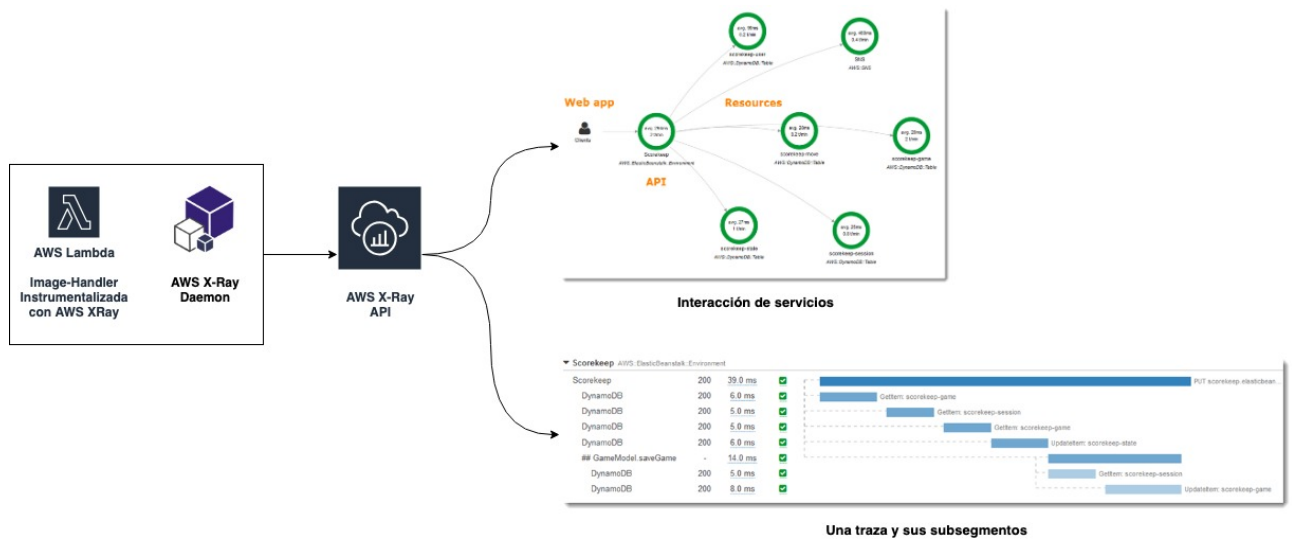


Figura 2.5: *Image-Handler* publicando eventos de rendimiento al servicio AWS X-Ray

2.3. Estrategia de extracción de modelo de rendimiento para *Image-Handler*

Debido a que las funciones Lambda se ejecutan en contenedores que son tanto inaccesibles como efímeros para los diseñadores e implementadores, y, sobre los cuales no se tiene ningún control, estrategias tradicionales en donde se crean bitácoras en la misma computadora en donde se ejecuta la aplicación y se van monitoreando utilizando alguna herramienta especializada o simplemente 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 o servicio externo en el cual se tenga control para acceder a los resultados y manipularlos.

Para lograr extraer un modelo PCM a partir de las bitácoras de Kieker, se llevaron a cabo las siguientes actividades:

1. Creación de la versión Image-PK (Sección 2.2.2): Versión de *Image-Handler*

instrumentalizar el código fuente con las bibliotecas de Kieker para generar bitácoras del rendimiento de la función.

2. Provisionar una nueva máquina virtual en AWS la cual se va a:

- Ejecutar una cola JMS.
- Ejecutar una aplicación consumidora de mensajes de la cola JMS.
- Almacenar la bitácora de registros de rendimiento de la función Lambda.

3. Configurar la biblioteca de Kieker para señalar que la publicación de los registros de rendimiento de la función se hagan a través de la cola JMS en la máquina virtual del punto #2.

4. Creación de una aplicación consumidora de mensajes para que una vez que arriben los mensajes a la cola JMS, esta procese los mensajes de la cola y los almacene en una bitácora.

La figura 2.6 muestra los involucrados en el proceso de publicación de mediciones de rendimiento del código de *Image-Handler* hacia una bitácora externa.

Una vez que la función Lambda se ejercitó y se haya obtenido 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 2.7. Se espera que, al utilizarse datos de mediciones directas del rendimiento de la función Lambda, PMX pueda generar una instancia de PCM que sea confiable y de la que se puedan obtener simulaciones que representativas del comportamiento de la función Lambda.

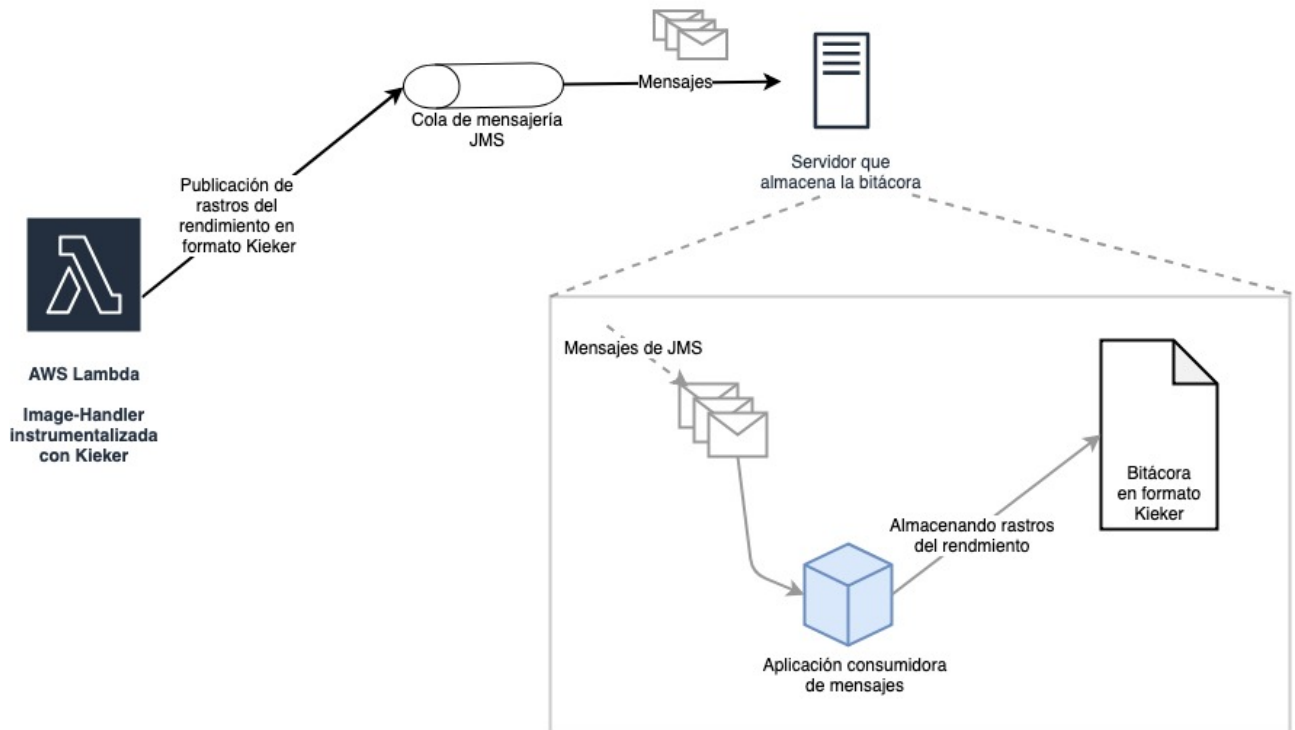


Figura 2.6: Publicando mediciones del rendimiento de la función Lambda.

Versión instrumentada para AWS X-Ray

Una actividad recurrente durante el modelado y simulación de arquitecturas de software, es la del afinamiento del modelo. Esta tarea es necesaria para que el modelo en cual se está trabajando pueda llegar a convertirse una representación cercana del comportamiento de un software en escenarios reales.

Durante el trabajo con el modelo PCM obtenido en la Sección 2.2.2 se notó que las estimaciones hechas por PMX no correspondían al comportamiento observado y que el esfuerzo necesario para obtener estas estimaciones a partir de las bitácoras de Kieker iba a ser grande, principalmente porque el formato de las bitácoras de Kieker incluye muchos más datos que solo las estimaciones de rendimiento y porque se empezó a tener la sensación de que estos datos de

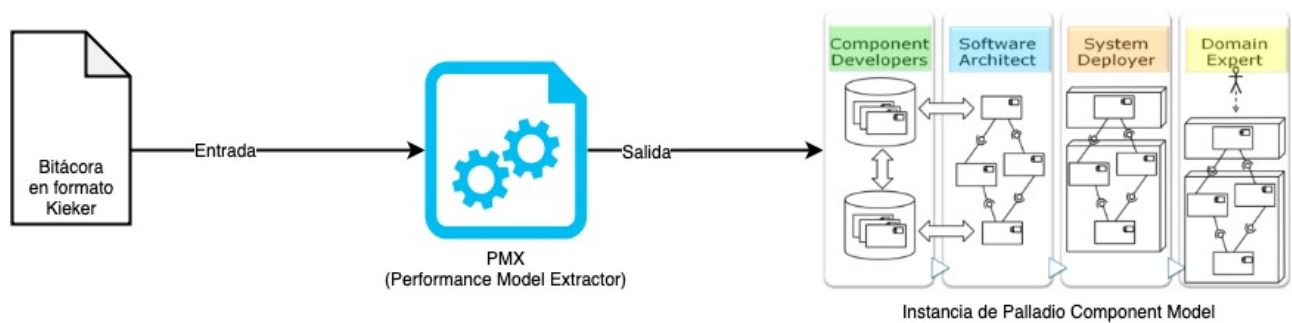


Figura 2.7: Convirtiendo una bitácora de Kieker a una instancia de PCM por medio de PMX.

alguna u otra forma no estaban “contando toda la historia” de lo que estaba pasando con la función Lambda. Por ejemplo, los datos de rendimiento de Kieker no podían explicar tiempos de retraso que se observaban al inicio de la ejecución de la función Lambda y, sin esos datos, se podía llegar a generar un modelo poco preciso.

Por esta razón se comenzó a explorar herramientas alternativas para obtener mediciones del rendimiento de la función Lambda y se eligió Amazon X-Ray⁸ para este fin. AWS XRay ayuda a los desarrolladores a analizar y depurar aplicaciones en producción distribuidas, tal y como lo son las basadas en arquitecturas de microservicios. Con AWS X-Ray se puede ver cómo es que la aplicación y sus servicios asociados se están ejecutando para identificar y resolver problemas de rendimiento y errores en general.

AWS X-Ray recolecta datos de las solicitudes que hacen a cada uno de los servicios de la aplicación y los agrupa en unas unidades llamadas trazas (*traces*). Luego, utilizando estas trazas, es posible ver mapas de la interacción de los servicios, latencias y meta datos para analizar el comportamiento o identificar problemas.

⁸<https://aws.amazon.com/xray/>

2.4. Diseño Experimental

En esta sección se detallan los experimentos realizados para validar si el modelo y la simulaciones sobre el mismo, logran dar caracterizaciones relevantes del comportamiento de la función *Image Handler* en distintos escenarios.

2.4.1. Utilizando *Image-Handler* para redimensionar imágenes de distintos tamaños

Este es el caso que se menciona en la Sección 2.1.1 y se muestra en la figura 2.3. Se probó la función Lambda con dos imágenes: una de tamaño pequeño ($\sim 100\text{Kb}$) y otra de tamaño grande ($\sim 5\text{Mb}$). El objetivo es comprobar cómo los distintos tamaños de las imágenes influyen en el tiempo de respuesta de la función.

Intuitivamente, se espera que, cuando se hagan solicitudes de redimensionamiento de imágenes de mayor tamaño tomen mayor tiempo en ser procesadas y que lo contrario suceda con las imágenes de menos tamaño. Los resultados obtenidos brindan una referencia inicial para saber cómo es que los componentes de software asociados al redimensionamiento trabajan y qué posibles mejoras podrían realizarse.

Las cargas de trabajo para este experimento son de tipo *cerrada*, lo que quiere decir que una solicitud se ejecuta solamente hasta que la anterior se termina. Esto va orientado a tener mejor trazabilidad de lo que ocurre con la función.

Ejercitando la función Lambda con solicitudes de redimensionamiento en una imagen pequeña(~100Kb)

Este, al ser el primer experimento realizado, sentó las bases del marco de trabajo en modelaje y simulación que se aplicaron en otros experimentos.

Se utilizó la siguiente configuración para la ejecución del experimento:

1. Se utilizó una imagen de tamaño de 19Kb y de 295 pixeles de ancho y alto.
2. La función *Image-Handler* se ejercitó con una carga de trabajo *cerrada* de 100 solicitudes de redimensionamiento para convertir la imagen del punto #1 a una imagen de 50 pixeles de ancho y 60 de alto.
 - a) En primera instancia, las 100 solicitudes se hicieron usando la versión de la función *Image-Handler* instrumentalizada con Kieker (Sección 2.2.2), para obtener una bitácora con las operaciones realizadas
 - b) Posteriormente, esta bitácora fue procesada por PMX para obtener una instancia de PCM.
3. Una vez obtenido la instancia de PCM, se ejecutó otra carga de trabajo *cerrada* de 100 solicitudes de redimensionamiento pero esta vez utilizando la versión de la función *Image-Handler* instrumentalizada con AWS X-Ray (Sección 2.3). Esto para obtener métricas más detalladas sobre el uso de la función y así para afinar las estimaciones de los componentes que conforman la instancia de PCM.
 - a) Una vez obtenidas estas mediciones, se realizaron análisis de frecuencias para determinar la distribución de las cargas de trabajo en cada componente.

4. Se realizaron simulaciones sobre la instancia de PCM utilizando el motor de simulaciones provisto por *Palladio Workbench*.
5. Se ejecutó una carga de trabajo *cerrada* de 100 solicitudes de redimensionamiento utilizando la versión original de la función *Image-Handler*.
6. Por último se compararon y analizaron los resultados del punto #4 y #5 para determinar si, de acuerdo a lo obtenido, las simulaciones hechas sobre el modelo lograban caracterizar el rendimiento de la función Lambda.

Como resultado del trabajo llevado a cabo en el punto #2, se obtuvieron los componentes mostrados en la figura 2.8. En PCM a este diagrama se le conoce como diagrama de repositorio.

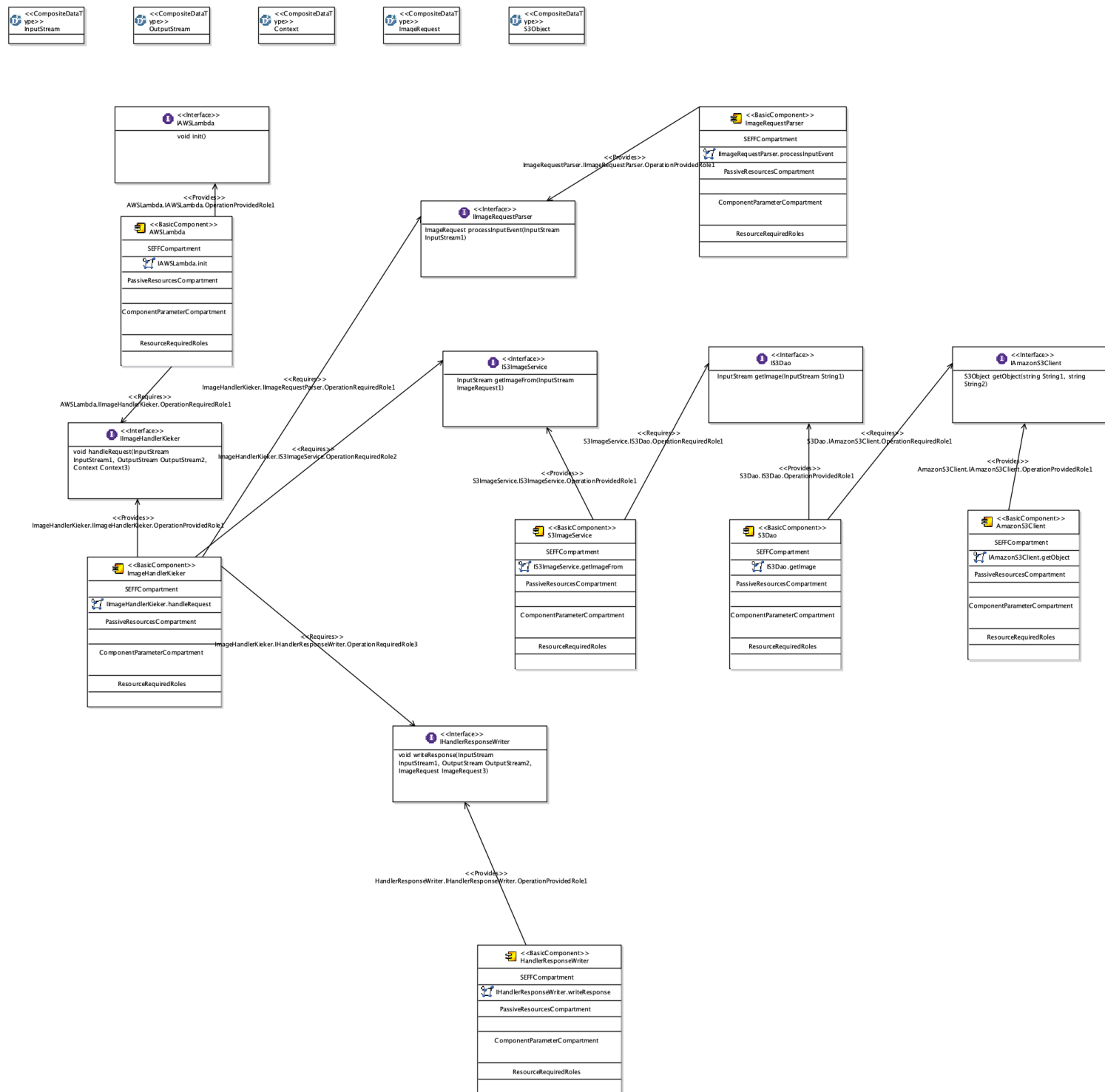


Figura 2.8: *Image-Handler* publicando eventos de rendimiento al servicio AWS X-Ray

A partir de la bitácora proporcionada como entrada, PMX logró identificar 6 componentes principales:

- ImageHandlerKieker: El punto de entrada de la función.
- ImageRequestParser: Encargado de tomar la solicitud de redimensionamiento, analizarla y convertirla en un objeto que pueda ser utilizado por el resto de componentes.
- S3ImageService: Contiene la lógica de:
 1. Cómo obtener una imagen y
 2. Cómo aplicar la redimensión sobre la misma.
- S3Dao: El componente que sabe cómo obtener una imagen el servicio AWS S3
- AmazonS3Client: Contiene las operaciones de comunicación de bajo nivel con el servicio AWS.
- HandlerResponseWriter: Convierte la imagen redimensionada a una representación en Base 64 y prepara la respuesta de la función.

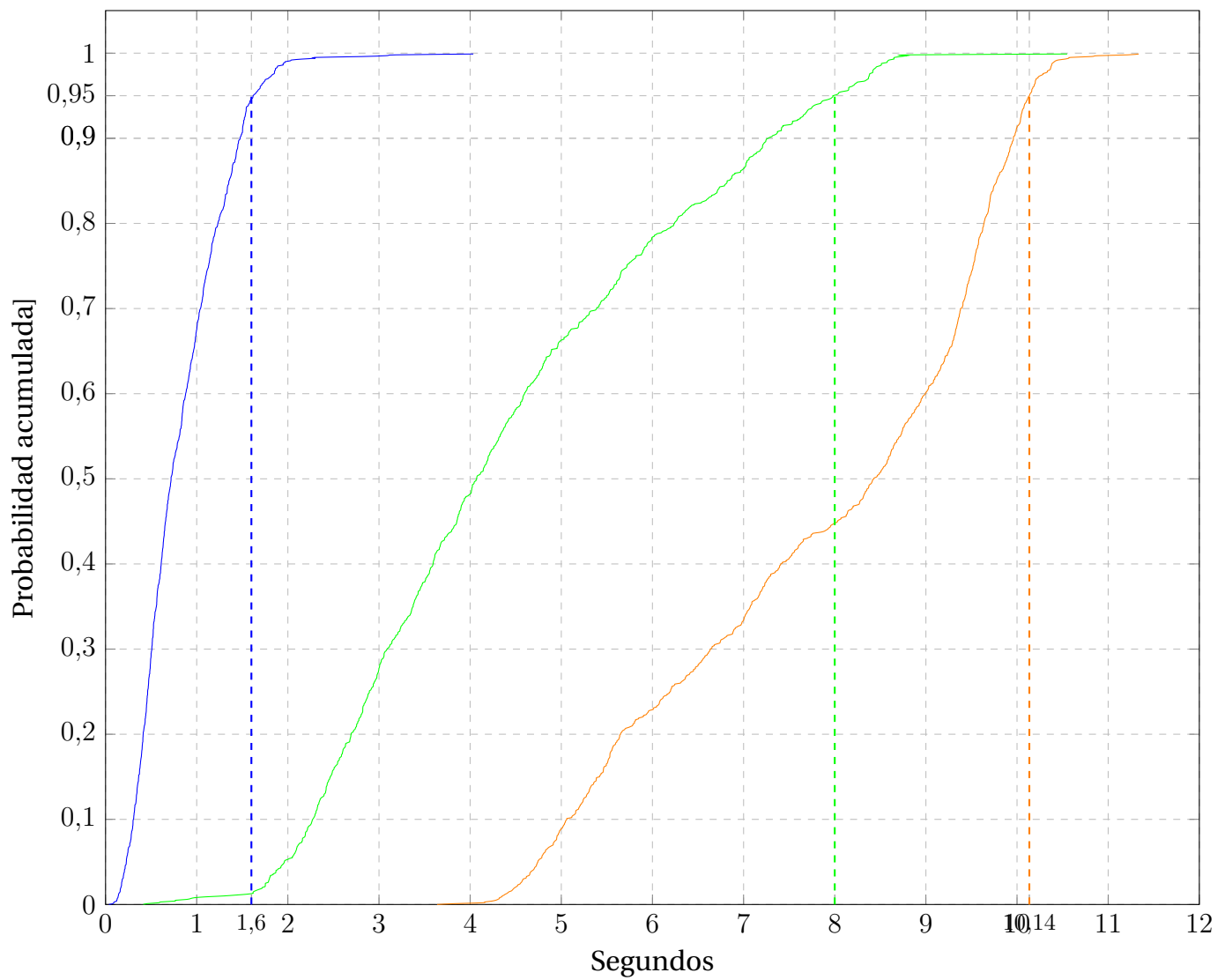
Cada uno de ellos expone su funcionalidad por medio de una interfaz. En el modelado y simulación basado en componentes, los componentes se conciben como piezas intercambiables los cuales exponen sus operaciones por medio de interfaces y delegan los detalles de implementación a componentes concretos (Como por ejemplo *BasicComponents*).

Durante el análisis del modelo generado por PMX, se pudo notar que si bien el modelo generado representaba muy bien la intención detrás de los componentes del código fuente, no era detallado en las estimaciones del uso de cada

componente. En PCM, a cada componente se le puede especificar su flujo de trabajo, estado e invocaciones a otros componentes, por medio de *Service Effect Specifications* (SEFF).

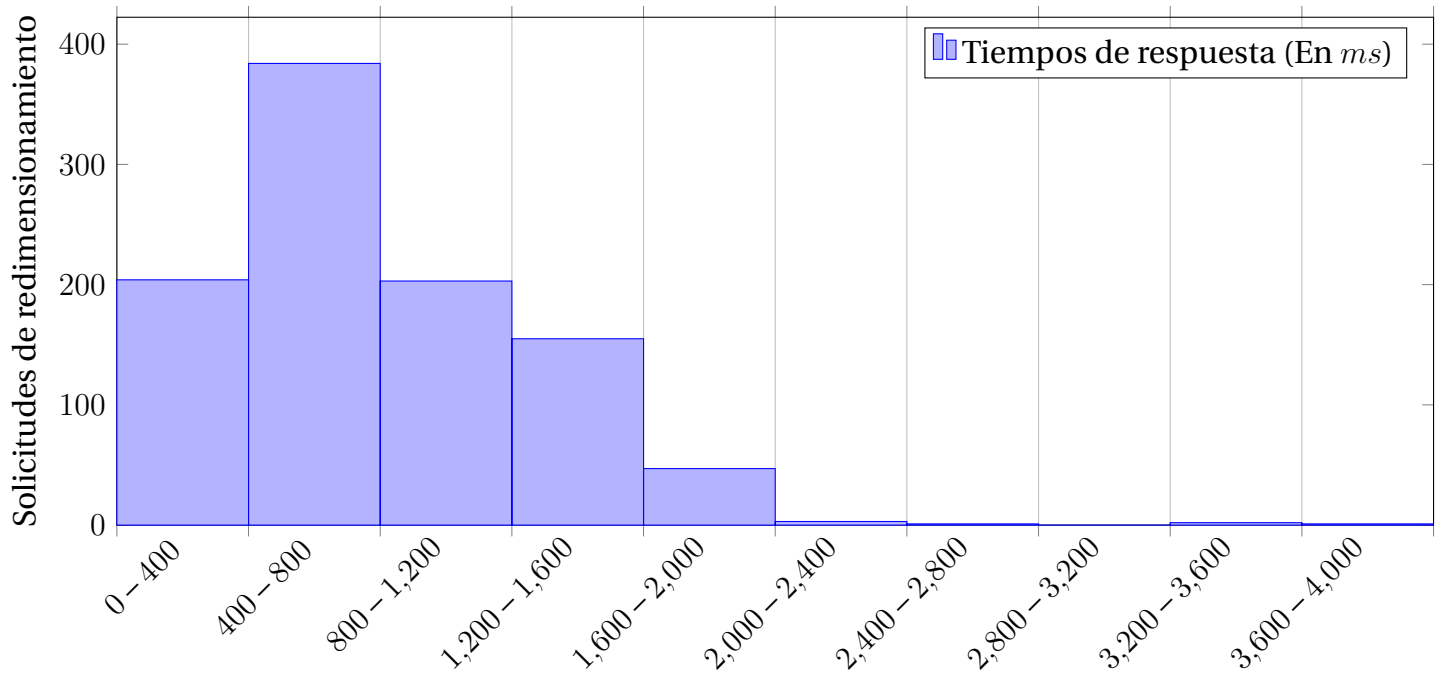
Los SEFFs generados por PMX no eran lo suficientemente detallados y cuando se realizaban simulaciones, los resultados no lograban acercarse a la observado en la función Lambda. Por esta razón fue se recurrió al trabajo del punto #3, para obtener datos “crudos” del rendimiento de la función Lambda e incluir esto en los *SEFFs* del modelo como distribuciones de frecuencia.

1000 Simulaciones: Función de probabilidad acumulada para los tres escenarios de pruebas

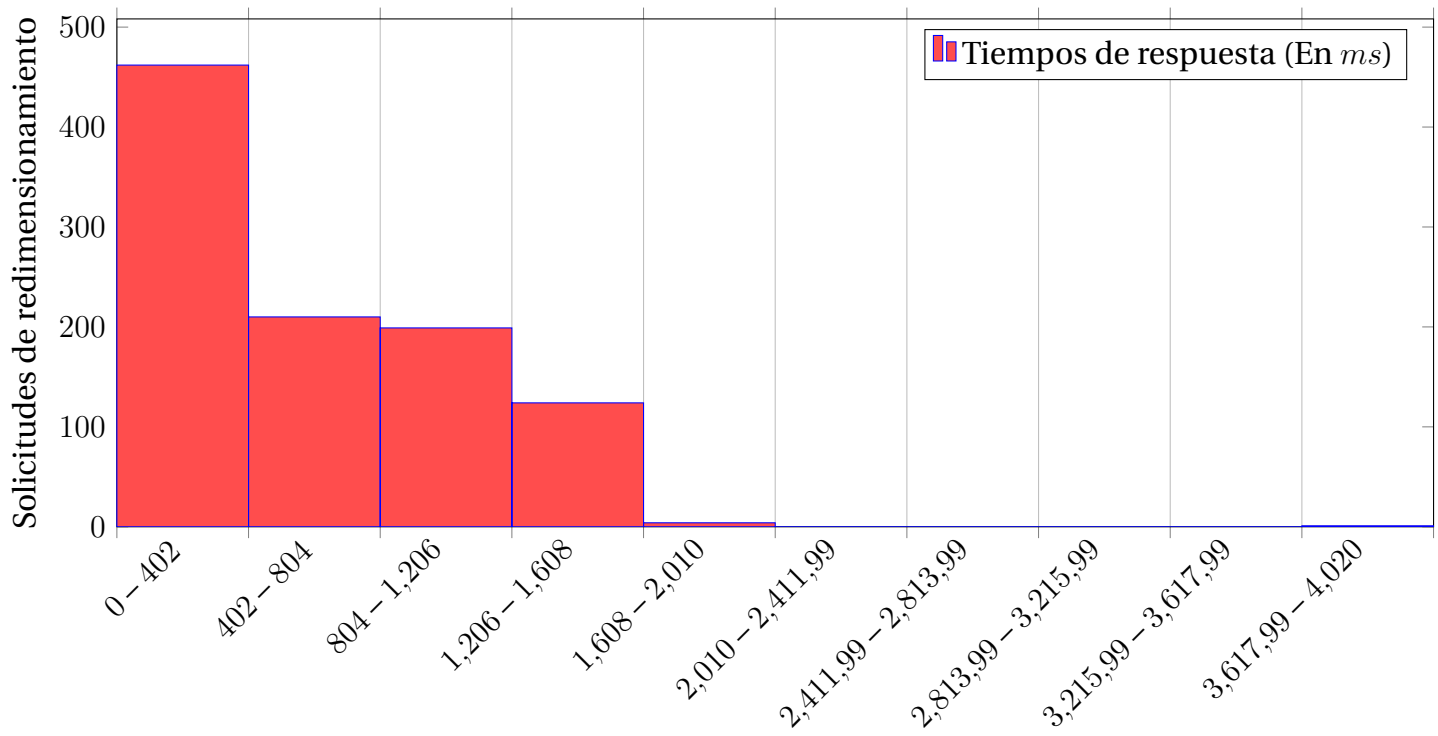


— Hasta 500Kb — De 500Kb a 1Mb — Mayor a 1Mb

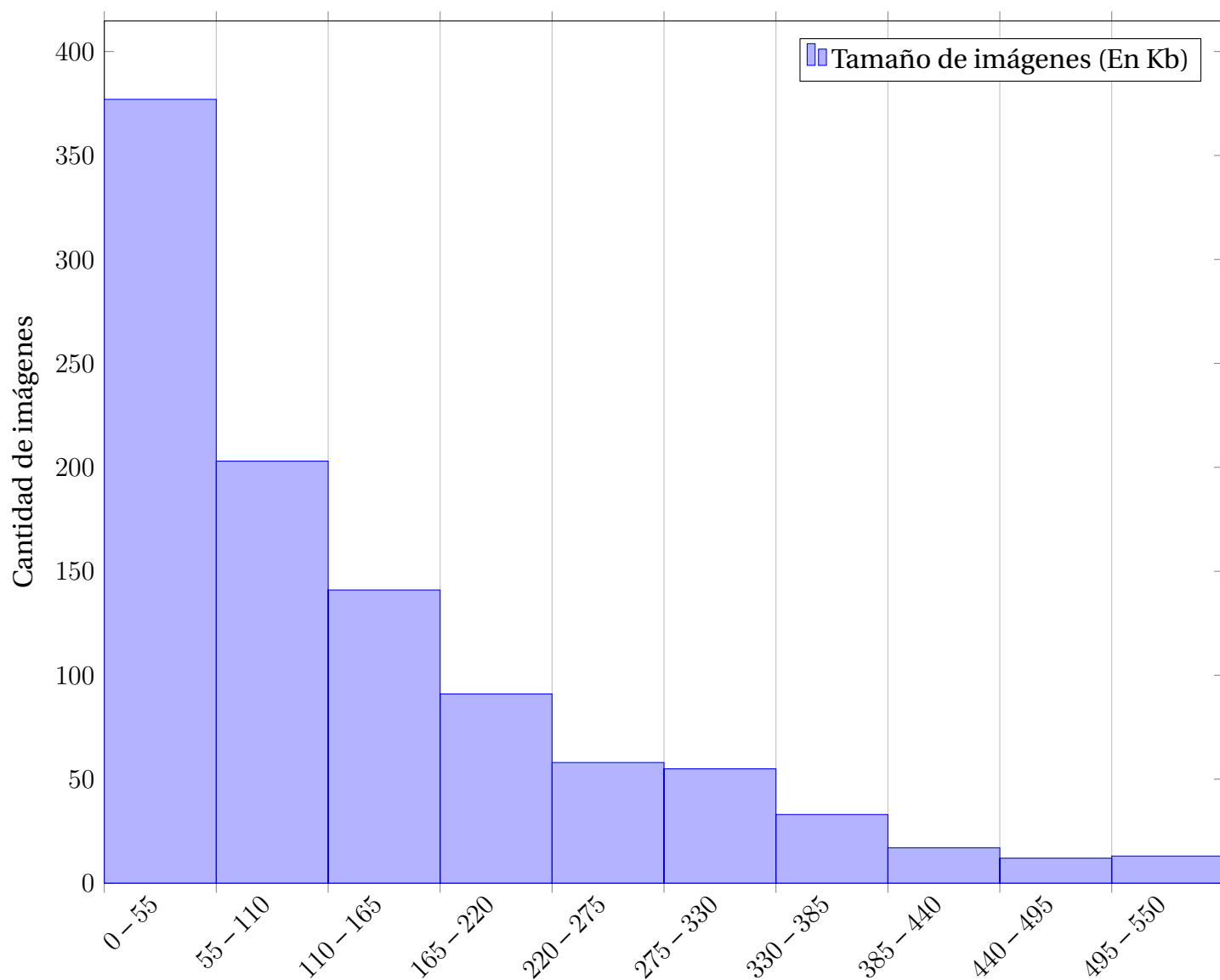
Distribución de los tiempos de respuesta en solicitudes de redimensionamiento
de imágenes de tamaño $\leq 500Kb$ en las simulaciones de Palladio



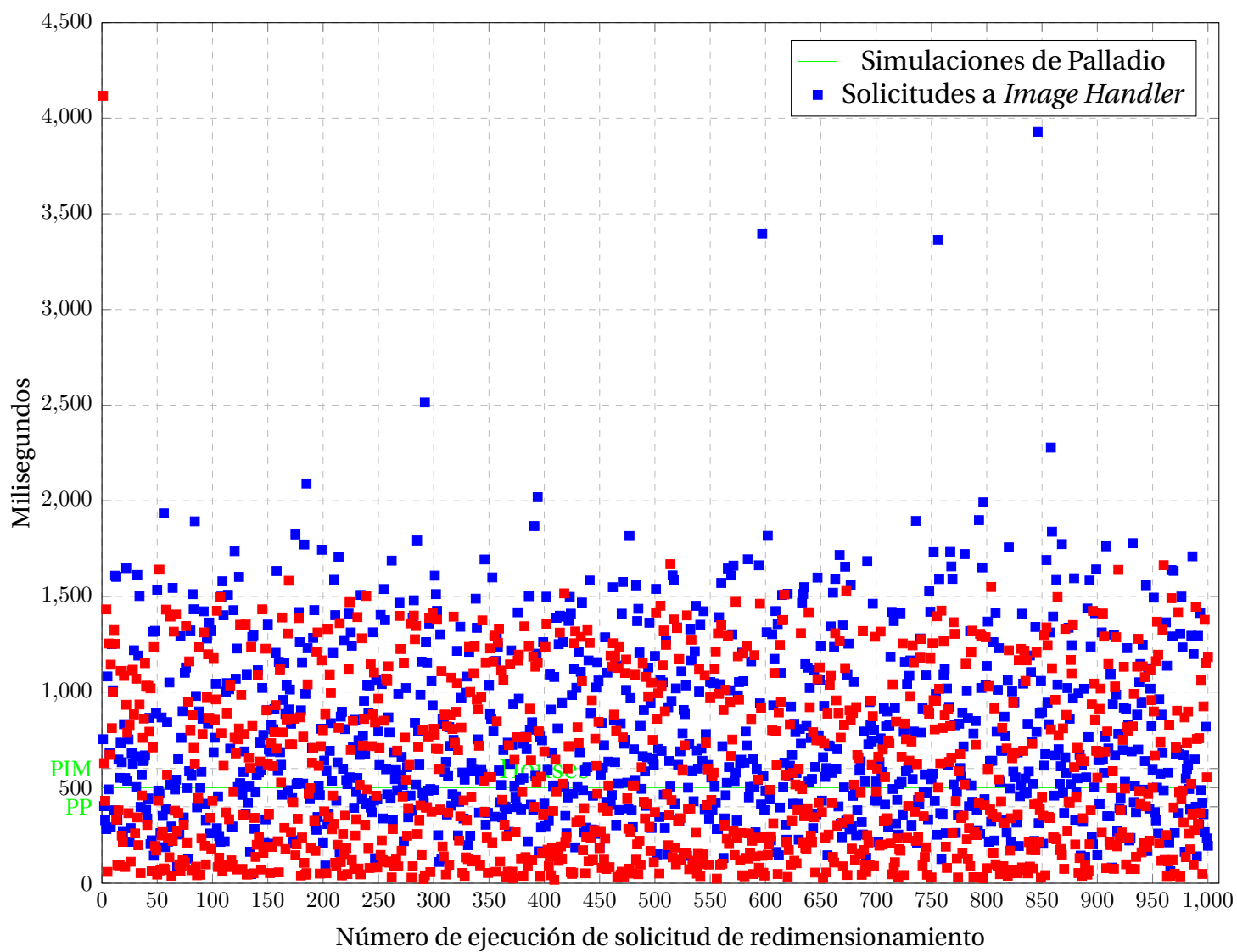
Distribución de los tiempos de respuesta en solicitudes de redimensionamiento
de imágenes de tamaño $\leq 500Kb$ en *Image Handler*



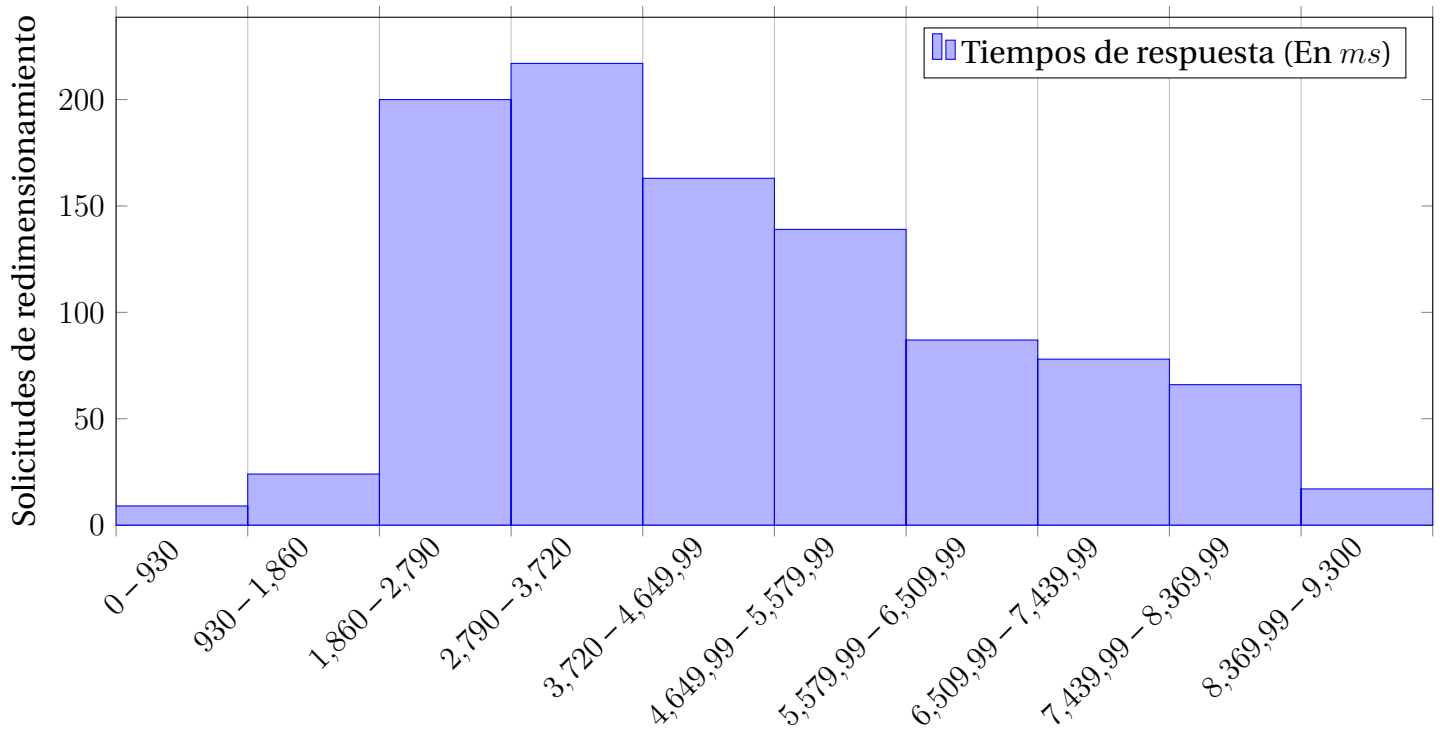
Distribución del tamaño de imágenes de tamaño $\leq 500Kb$



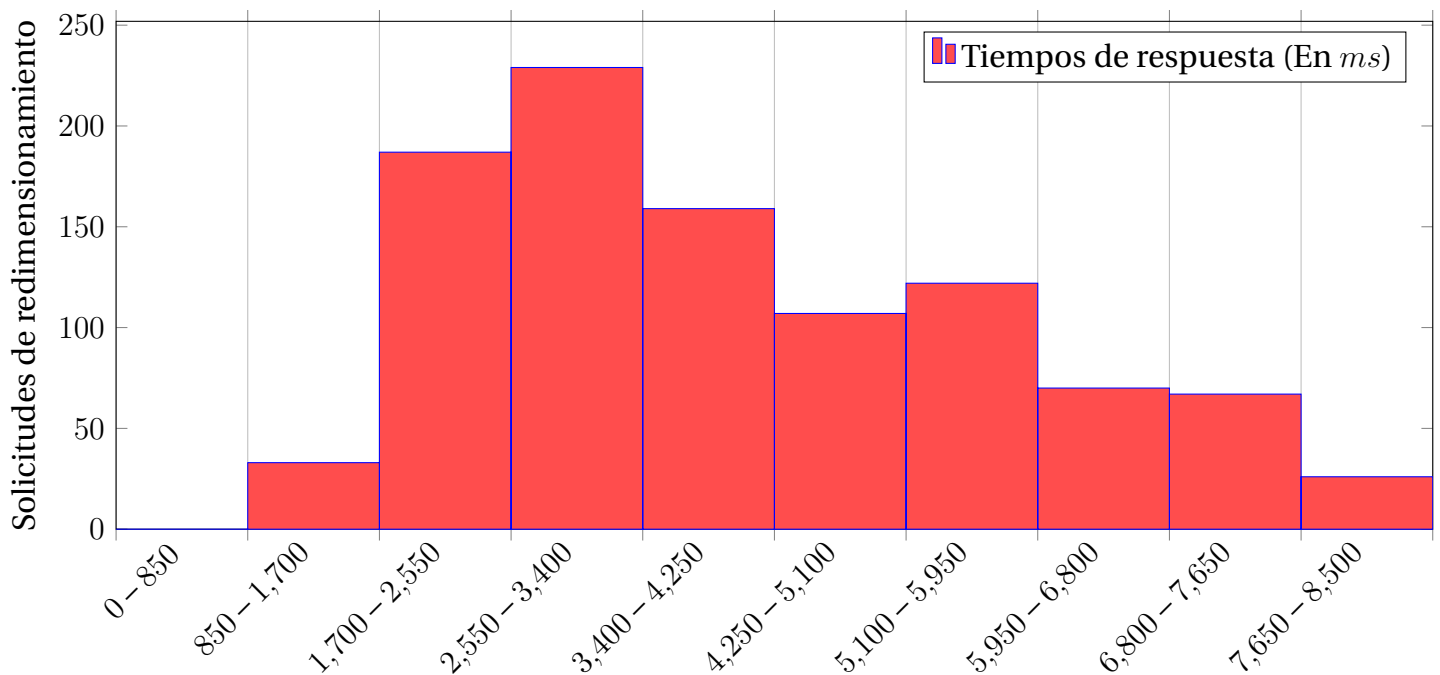
1000 Solicitudes de redimensionamiento de imágenes de tamaño $\leq 500Kb$



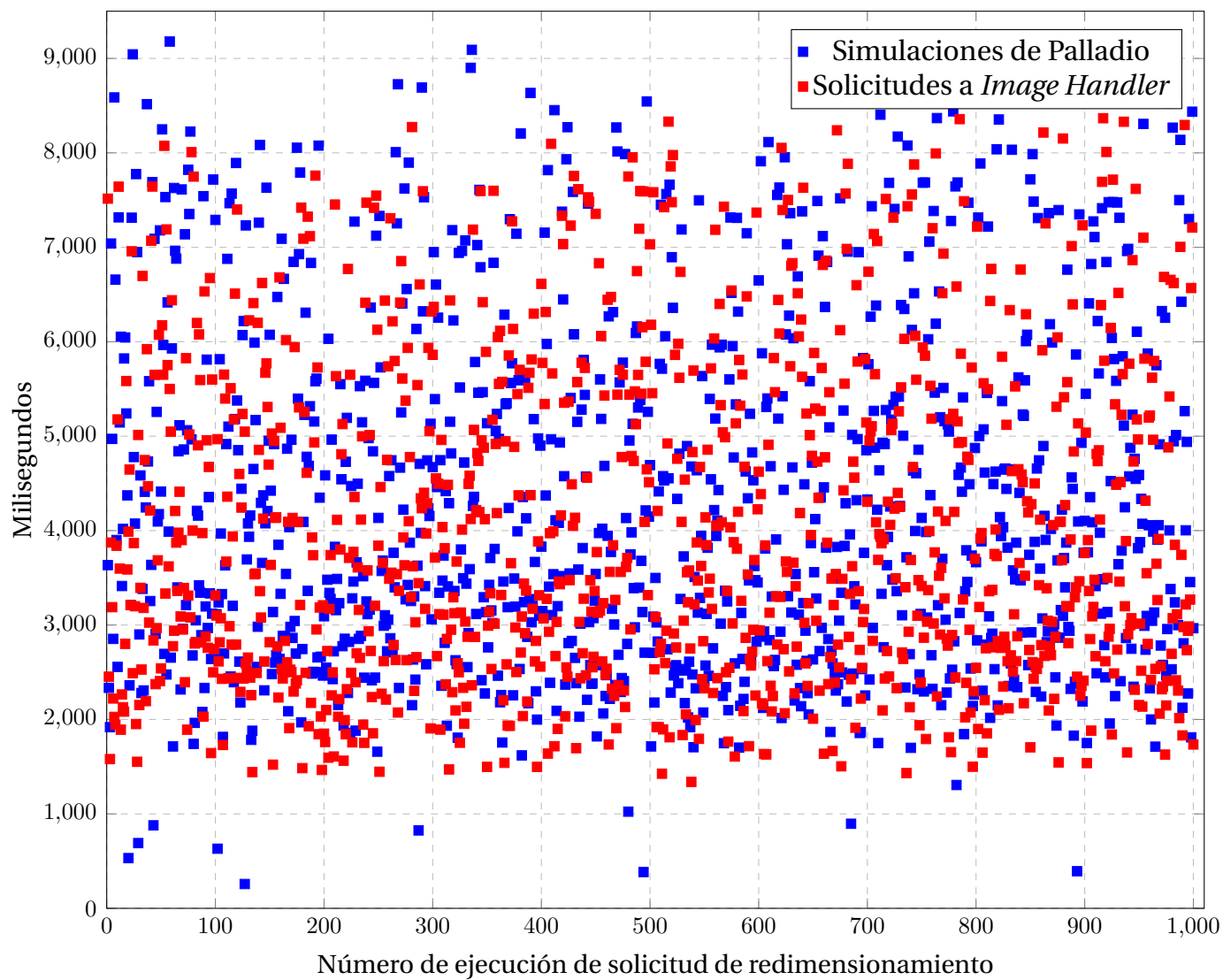
Distribución de los tiempos de respuesta en solicitudes de redimensionamiento
de imágenes de tamaño $500Kb \leq x \leq 1Mb$ en las simulaciones de Palladio



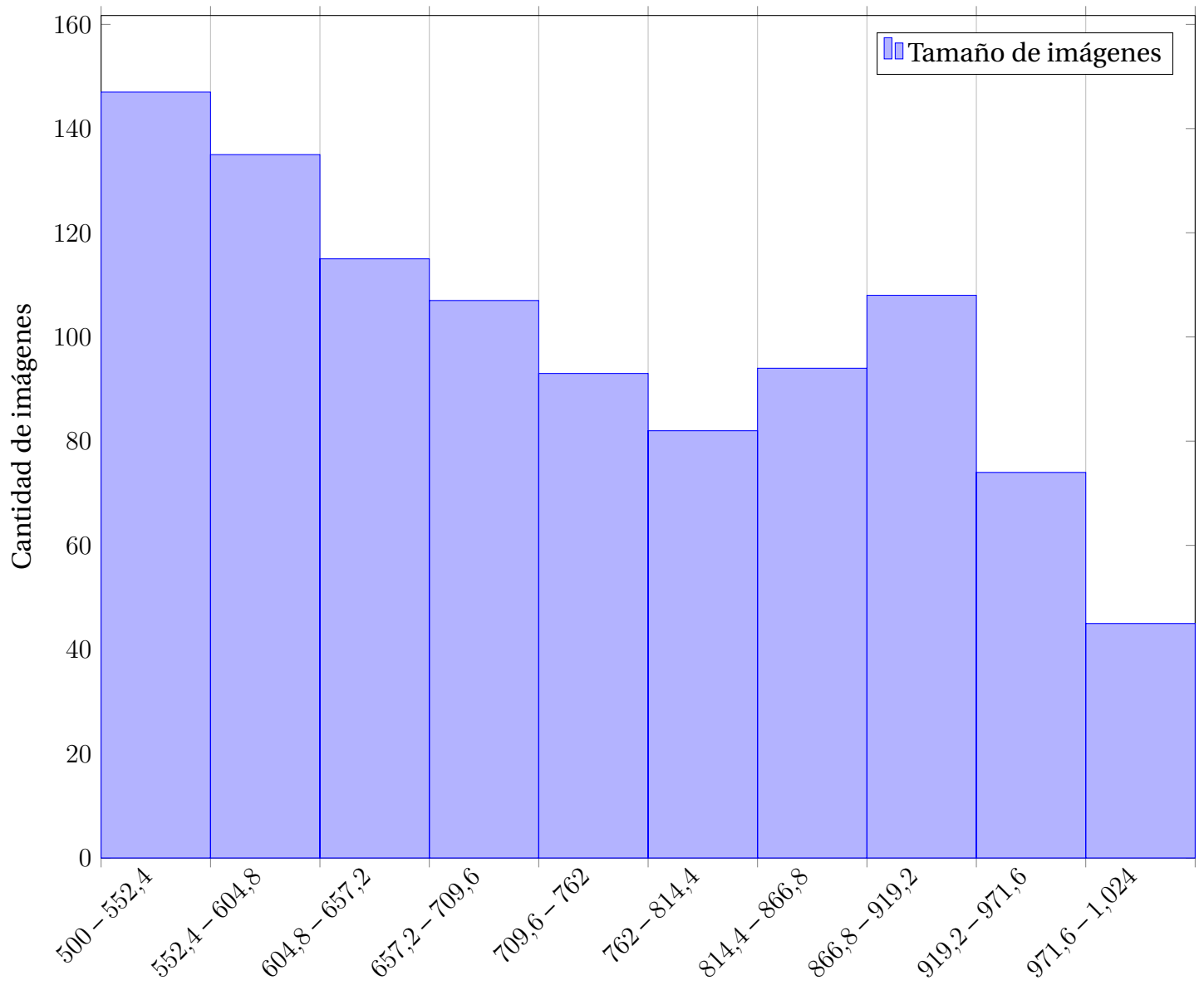
Distribución de los tiempos de respuesta en solicitudes de redimensionamiento
de imágenes de tamaño $500Kb \leq x \leq 1Mb$ en *Image Handler*



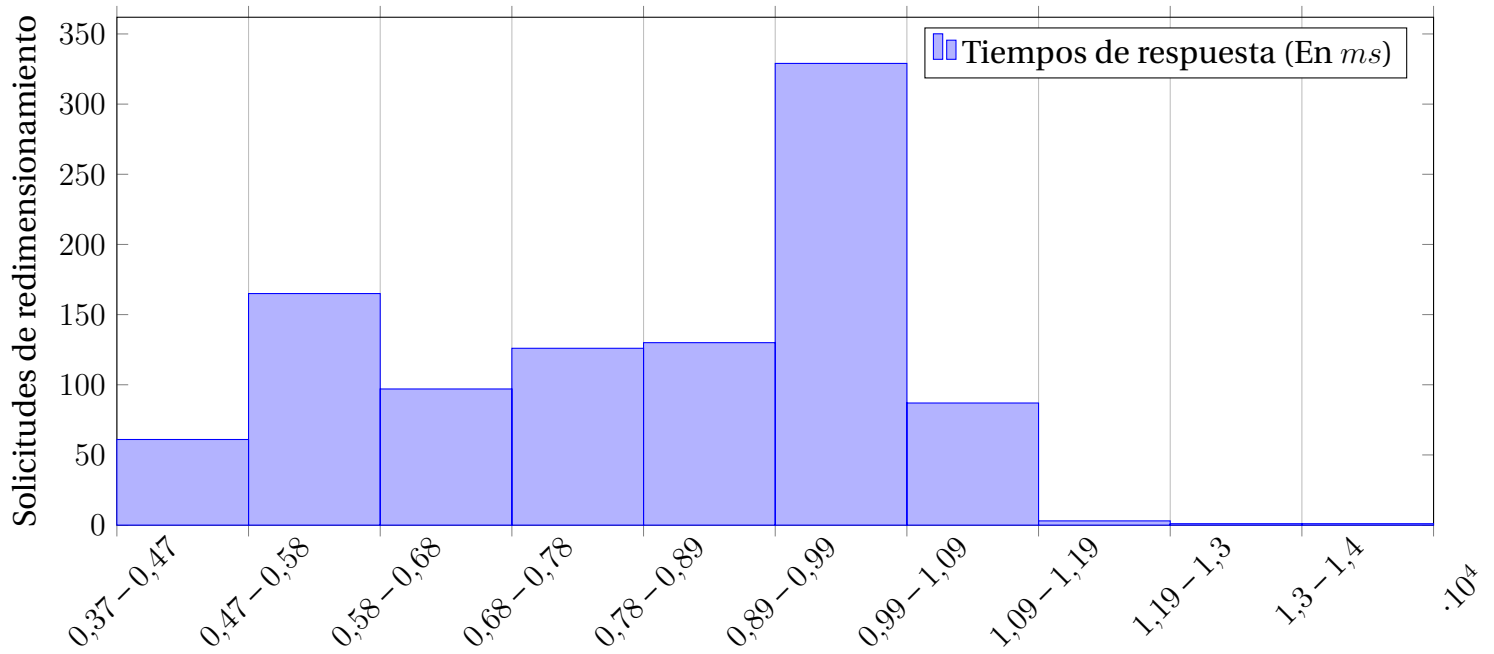
Solicitudes de redimensionamiento de imágenes de tamaño $500Kb \leq x \leq 1Mb$



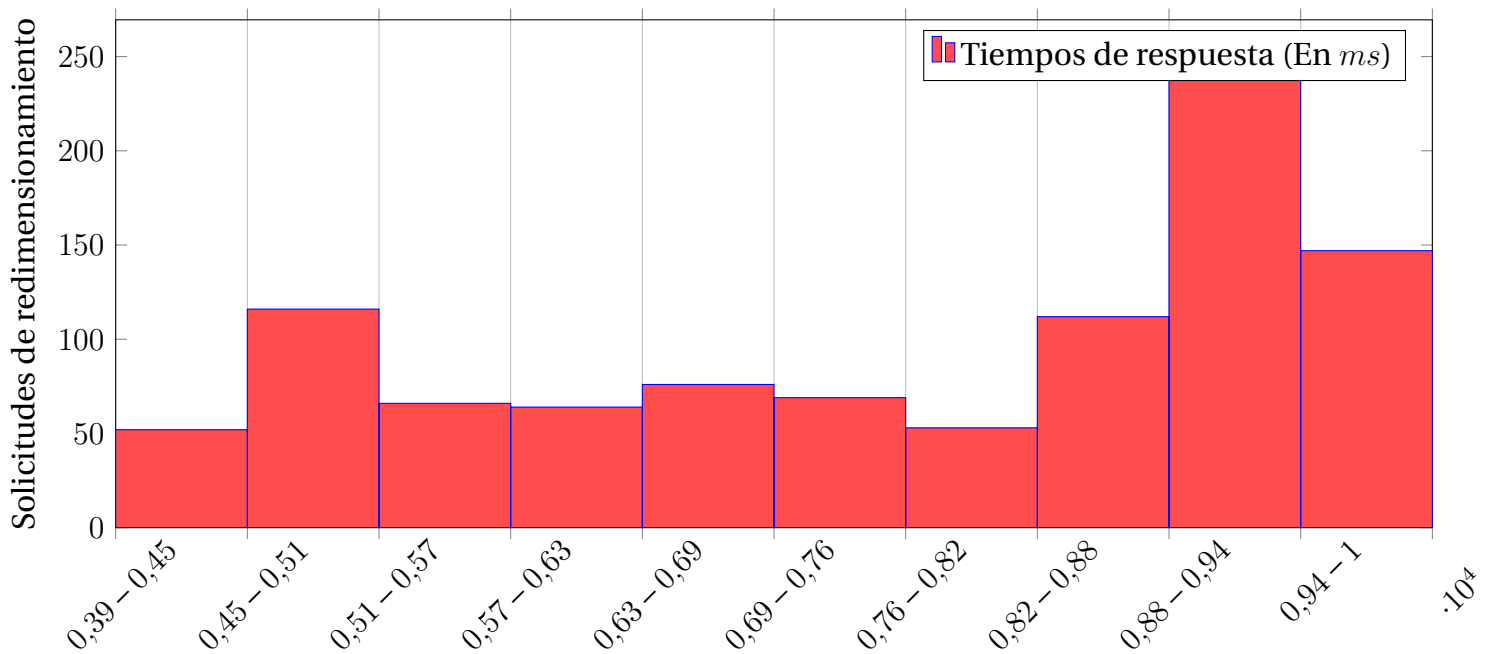
Distribución del tamaño de imágenes de tamaño $500Kb \leq x \leq 1Mb$



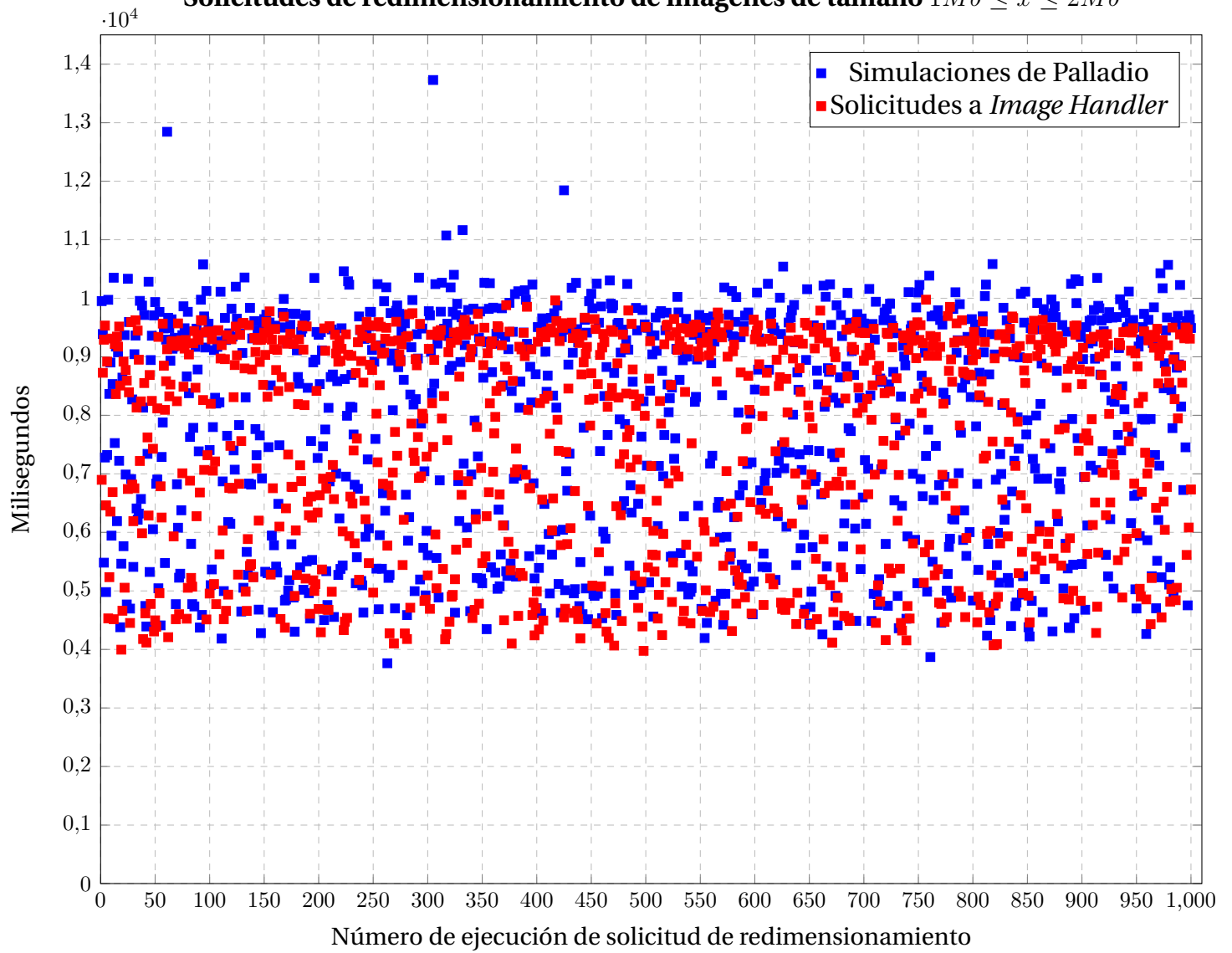
Distribución de los tiempos de respuesta en solicitudes de redimensionamiento
de imágenes de tamaño $1Mb \leq x \leq 2Mb$ en las simulaciones de Palladio



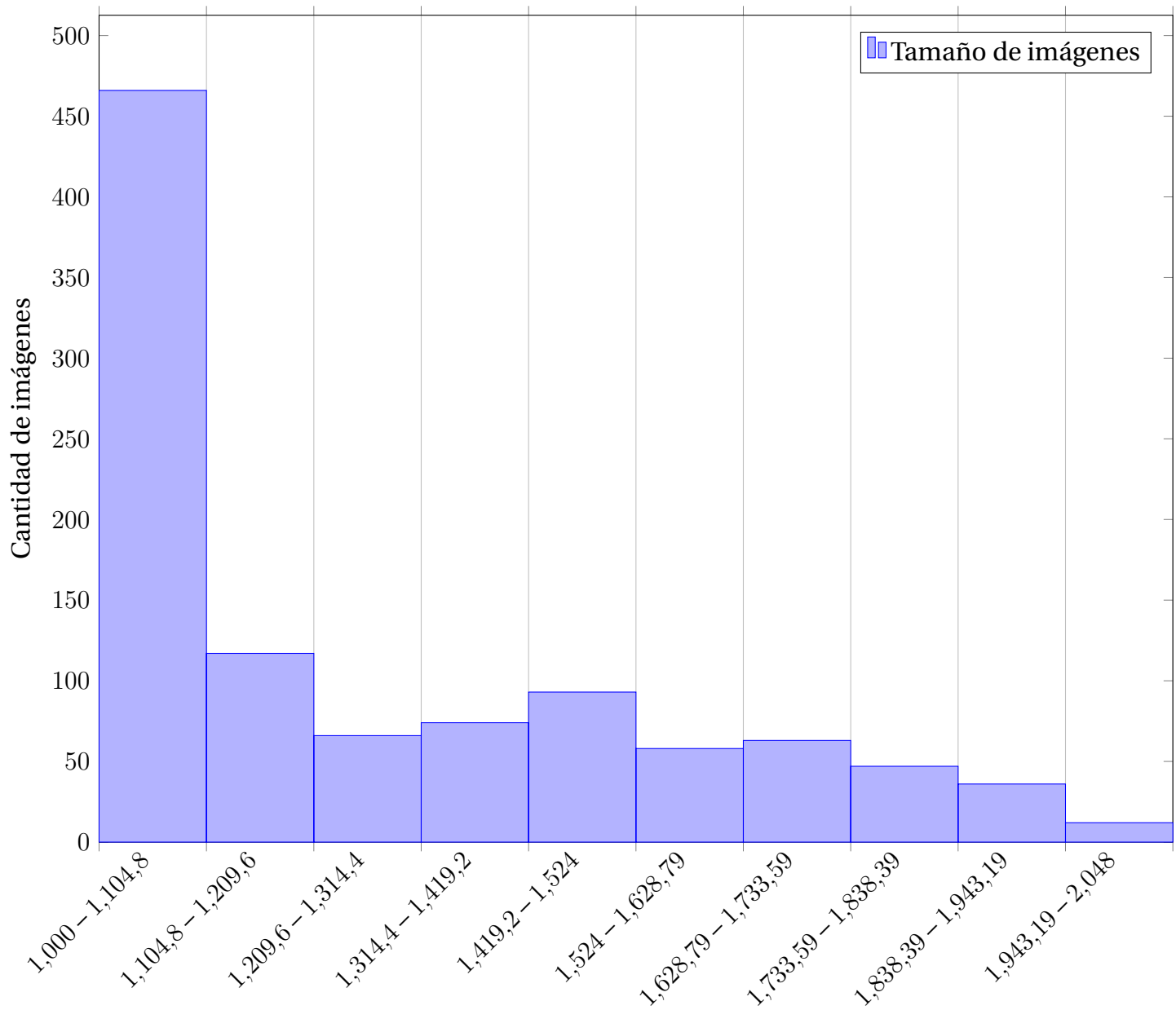
Distribución de los tiempos de respuesta en solicitudes de redimensionamiento
de imágenes de tamaño $1Mb \leq x \leq 2Mb$ en *Image Handler*



Solicitudes de redimensionamiento de imágenes de tamaño $1Mb \leq x \leq 2Mb$



Distribución del tamaño de imágenes de tamaño $1Mb \leq x \leq 2Mb$



Hasta 500Kb			
Solicitud de redimensionamiento	Image Handler	Palladio	Diferencia
Tiempo promedio	583.842ms	793.808ms	209.965ms
Desviación estándar	460.659ms	465.441ms	4.782ms
Varianza	212206.961	216635	–
Mediana	466.715ms	680.482ms	.–
Coefficiente de variación	0.987	0.683	–
Entre 500Kb a 1Mb			
Tiempo promedio	4073.600ms	4348.029ms	274.428ms
Desviación estándar	1731.974ms	1844.893ms	112.919ms
Varianza	2999736.844	3403633.84	–
Mediana	3658.825ms	3989.406ms	–
Coefficiente de variación	0.473	0.462	–
Entre 1Mb y 2Mb			
Tiempo promedio	7539.139ms	7796.913ms	257.773ms
Desviación estándar	1816.152ms	1914.258ms	98.106ms
Varianza	3298410.017	3664385	–
Mediana	8200.875ms	8310.293ms	–
Coefficiente de variación	0.221	0.230	–

Cuadro 2.1: Resumen de datos estadísticos

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.