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: 16fd310

Date: 2019-08-05 21:49:13 -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

Lista de Tablas

Glosario

UNADECA Universidad Adventista de Centro América. 126

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 2 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 3 se define el problema a resolver. En el capítulo 4 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 6. El alcance del proyecto se define en el capitulo **??**. Los entregables que se generarán a partir de esta propuesta se listan en el capítulo 7. La metodología de trabajo se indica en el capítulo **??**. La propuesta concluye en el capítulo **??**, donde se presenta el cronograma de actividades.

Capítulo 2

Antecedentes

2.1. Marco conceptual

2.1.1. Ingeniería de rendimiento de software (*Software Performance Engineering*)

Una descripción comúnmente utilizada para definir ingeniería de rendimiento de software (*Software Performance Engineering* - SPE) es la brindada en Woodside et al. [3]: *"Ingeniería de rendimiento del software representa toda la colección de actividades de ingeniería de software y análisis relacionados utilizados a través del ciclo de desarrollo de software que están dirigidos a cumplir con los requerimientos de rendimiento"*.

De acuerdo con estos mismos autores, los enfoques para ingeniería de rendimiento puede ser divididos en dos categorías: basadas en mediciones y basadas en modelos. La primera es la más común y utiliza pruebas, diagnóstico y ajustes una vez que existe un sistema en ejecución que se puede medir; es por esto

que solamente puede ser utilizada conforme se va acercando el final del ciclo de desarrollo de software. En contraste con el enfoque basado en mediciones, el enfoque basado en modelos se centra en las etapas iniciales del desarrollo. Como el nombre lo indica, en este enfoque los modelos son clave para hacer predicciones cuantitativas de cuán bien una arquitectura puede cumplir sus expectativas de rendimiento.

Se han propuesto otras clasificaciones de enfoques para SPE pero, con respecto de la evaluación de sistemas basados en componentes, en [4] se deja la clasificación a un lado debido a que se argumenta que la mayoría de enfoques de modelaje toman alguna medición como entrada y a la mayoría de los métodos de medición los acompaña algún modelo.

Ingeniería de rendimiento basada en mediciones

Los enfoques basados en mediciones son los que prevalencen en la industria[5] y son típicamente utilizados para verificación(¿cumple el sistema con sus requerimientos de rendimiento?) o para localizar y reparar *hot-spots* (cuáles son las partes que tienen peor rendimiento en el sistema). La medición de rendimiento se remonta al inicio de la era de la computación, lo que ha generado una amplia gama de herramientas como generadores de carga y monitores para crear cargas de trabajo ficticias y llevar a cabo la medición de un sistema respectivamente.

Las pruebas de rendimiento aplican técnicas basadas en medición y usualmente esto es hecho luego de las pruebas funcionales o de carga. Las pruebas de carga validan el funcionamiento de un sistema bajo cargas de trabajo pesadas, mientras que las pruebas de rendimiento son usadas para obtener datos

cuantitativos de características de rendimiento, como tiempos de respuesta, *throughput* y utilización de hardware para una configuración de un sistema bajo una carga de trabajo definida.

Ingeniería de rendimiento por medio de modelado

La importancia del modelado del rendimiento está motivada por el riesgo de que se presenten problemas graves de rendimiento y la creciente complejidad de sistemas modernos, lo que hace difícil abordar los problemas de rendimiento en el nivel de código[6]. Cambios considerables en el diseño o en las arquitecturas pueden ser requeridos para mitigar los problemas de rendimiento. Por esta razón, la comunidad de investigación de modelado de rendimiento intenta luchar contra el enfoque de "arreglar las cosas luego" durante el proceso de desarrollo. Con la aplicación de un modelo del rendimiento de software se busca encontrar problemas de rendimiento y alternativas de diseño de manera temprana en el ciclo de desarrollo, evitando así el costo y la complejidad de un rediseño o cambios en los requerimientos.

Las herramientas de modelado de rendimiento ayudan a predecir la conducta del sistema antes que este sea construido, o bien, evaluar el resultado de un cambio antes de su implementación. El modelado del rendimiento puede ser usado como una herramienta de alerta temprana durante todo el ciclo de desarrollo con mayor precisión y modelos cada vez más detallados a lo largo del proceso. Al iniciar el desarrollo, un modelo no puede ser validado contra un sistema real, por esto el modelo representa el conocimiento incierto del diseñador. Como consecuencia de esto el modelo incluye suposiciones que no necesariamente se van a dar en el sistema real, pero que van a ser útiles para obtener una abstracción del comportamiento del sistema. En estas fases inicia-

les, la validación se obtiene mediante el uso del modelo, y existe el riesgo de conclusiones erróneas debido a su precisión limitada. Luego, el modelo puede ser validado contra mediciones en el sistema real (o parte de este) o prototipos de este, lo cual hace que la precisión del modelo se incremente.

En [7] se sugiere que los métodos actuales deben superar diversos retos antes de poder aplicarlos en sistemas existentes que enfrentan cambios en su arquitectura o requerimientos. Primero, debe quedar claro cómo se obtienen los valores para los parámetros del modelo y cómo se pueden validar los supuestos. Estimaciones basadas en la experiencia para estos parámetros no son suficientes, por lo que es necesario hacer mediciones en el sistema existente para formular predicciones precisas. Segundo, la caracterización de la carga del sistema en un entorno de producción es problemática debido a los recursos compartidos (bases de datos, hardware). Tercero, deben desarrollarse métodos para capturar parámetros del modelo dependientes de la carga. Por ejemplo un incremento en el tamaño de la base de datos probablemente incrementará las necesidades de procesador, memoria y disco en el servidor.

Técnicas comunes de modelado incluyen redes de colas y también extensiones de estas como redes de colas en capas y varios tipos de redes de Petri, y álgebras de procesos estocástica.

Modelado de Rendimiento

En SPE, la creación y evaluación de modelos de rendimiento es un concepto clave para evaluar cuantitativamente el rendimiento del diseño de un sistema y predecir el rendimiento de diversas alternativas de diseño. Un modelo de rendimiento captura el comportamiento relevante de diversos elementos (y sus

interacciones) para identificar el efecto de cambios en la configuración o en la carga de trabajo sobre el rendimiento de un sistema. Un buen modelo permite predecir los efectos de tales cambios sin necesidad de implementar y ejecutar un sistema en un ambiente de producción, que podrían ser no solamente tareas costosas sino también un desperdicio en caso que el hardware con el que se cuenta demuestre ser insuficiente para soportar la intensidad de la carga de trabajo.[8]

La forma del modelo de rendimiento puede comprender desde funciones matemáticas a formalismos de modelado estructural y modelos de simulación. Estos modelos varían en sus características clave; por ejemplo, los supuestos de modelado de los formalismos, el esfuerzo de modelado requerido y el nivel de abstracción.

En cuanto a técnicas de simulación, a pesar que estas permiten un estudio más detallado de los sistemas que modelos analíticos, la construcción de un modelo de simulación requiere de conocimiento detallado tanto de desarrollo de software como de estadística[5]. Los modelos de simulación también requieren usualmente de mayor tiempo de desarrollo que los modelos analíticos. En [3] se menciona que “la construcción de un modelo de simulación es cara, algunas veces comparable con el desarrollo de un sistema, y, los modelos de simulación detallados puede tardar casi tanto en ejecutarse como el sistema”.

2.1.2. Modelado y simulación de rendimiento basado en componentes

En este enfoque, modelos clásicos de rendimiento tales como redes de colas, redes de Petri estocásticas o álgebras de procesos estócasticas son aplicadas

para modelar y analizar el rendimiento de software basado en componentes. La ingeniería de software basada en componentes se considera un sucesor del desarrollo de software orientado a objetos, en esta los componentes de software son unidades de composición a las que se le define una especificación y una interfaz. A partir de estas, los arquitectos de software construyen e integran componentes de software entre sí, creando de esta manera sistemas más complejos.[4]

El reto en los modelos de rendimiento para componentes, es que el rendimiento de un componente de software en un sistema en ejecución depende del contexto en que está instalado y su perfil de uso, el cual es usualmente desconocido por el desarrollador del componente que creal el modelo de un componente individual.

Rendimiento de componentes de software

Szyperski[9] define un componente de software como: *"Un componente es una unidad de composición en aplicaciones de software, que posee un conjunto de interfaces y un conjunto de requerimientos (una especificación), y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio*".

Los componentes de software presentan los principios de ocultación de la información y la separación de responsabilidades. Fomentan la reutilización y preparan el sistema para que se puedan efectuar cambios de partes individuales. Permiten además una división de trabajo entre los desarrolladores de componentes y los arquitectos de software, lo que reduce la complejidad de la tarea de desarrollo. Los componentes de caja negra (*black-box*) solamente re-

velan sus interfaces a los clientes, mientras que los componentes de caja blanca (*white-box*) permiten ver y modificar el código fuente de la implementanción del componente como tal. Los componentes compuestos agrupan varios componentes en unidades más grandes.

Factores que influyen en el rendimiento de un componente Especificar el rendimiento de componentes reutilizables es difícil porque esto no solamente va a depender de la implementación del componente, sino también del contexto en donde este se encuentre instalado. Los factores que influyen en el rendimiento de los componentes son:

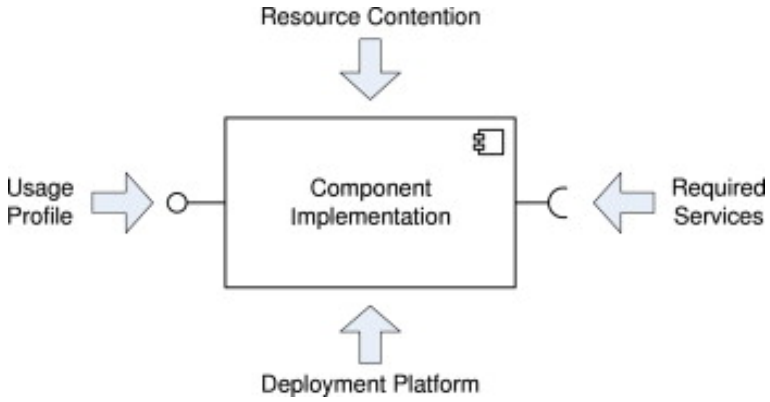

Figura 2.1: Factores que influyen en el rendimiento de un componente. Tomado de [4]

- *Implementación del componente*: los desarrolladores pueden implementar la funcionalidad especificada en una interfaz de diferentes formas. Dos componentes pueden proporcionar el mismo servicio pero presentar tiempos de ejecución diferentes cuando se ejecutan con los mismos recursos y con las mismas entradas.

- *Servicios requeridos*: cuando el componente A invoca el servicio del componente B, el tiempo de ejecución de B suma al tiempo de ejecución de A. Por lo tanto, el tiempo total de ejecución de un componente depende del tiempo de ejecución de los otros componentes/servicios que necesita.

- *Plataforma de instalación*: los arquitectos de software instalan un componente de software en diferentes plataformas. Una plataforma de instalación puede incluir varias capas de software (como por ejemplo contenedores de componentes o máquinas virtuales) y hardware (procesador, dispositivos de almacenamiento y red, etc)

- *Perfil de uso*: clientes pueden invocar los servicios del componente con diferentes parámetros de entrada. El tiempo de ejecución de un servicio puede cambiar dependiendo de los valores de lo parámetros de entrada.

- *Contención de recursos*: un componente de software típicamente no se ejecuta como un solo proceso aislado en una plataforma determinada. Los tiempos de espera inducidos para acceder a recursos limitados se suman al tiempo de ejecución de un componente de software.

Enfoques de ingeniería de rendimiento para software basado en componentes propuestos

La encuesta llevada a cabo en [4] proporciona una clasificación de enfoques de medición y predicción de rendimiento para sistemas de software basados en componentes. Otra clasificación es la que se expone en [10], donde se presenta una revisión de métodos de predicción de rendimiento basado en modelos para sistemas en general, pero no analiza los requerimientos propios para sistemas basados en componentes.

De acuerdo con [4] durante los últimos diez años, los investigadores han propuesto muchos enfoques para evaluar el rendimiento (tiempos de respuesta, *throughput*, utlización de recursos) de sistemas de software basados en componentes. Estos enfoques abarcan tanto predicción como medición del rendi-

miento. Los primeros analizan el rendimiento esperado de un diseño de software basado en componentes para evitar problemas de rendimiento en la implementación del sistema, lo que podría llevar a costos substanciales para rediseñar la arquitectura. Los otros analizan el rendimiento observable de sistemas de software basados en componentes implementados para entender sus propiedades, determinar su capacidad máxima, identificar componentes críticos y para resolver cuellos de botella.

Métodos de evaluación de rendimiento Los enfoques se reunieron en dos grandes grupos: enfoques principales que proporcionan procesos de evaluación de rendimiento completo y enfoques suplementarios que se centran en aspectos específicos como medición de componentes individuales o modelaje de las propiedades de rendimiento de los conectores de un componente.

Enfoques principales

- **Enfoques de predicción basados en UML**: los enfoques en este grupo se enfocan en la predicción de rendimiento en tiempo de diseño para sistemas de software basado en componentes modelados con el Lenguaje de Modelado Unificado (UML por sus siglas en inglés). UML 2.0 tiene la noción de componente de software como una clase extendida. UML permite modelar el comportamiento de un componente con diagramas de secuencia, actividad y colaboración. La asignación de los componentes puede ser descrita mediante diagramas de despliegue(*deployment*).
    - CB-SPE - *Component-Based Software Performance Engineering*
- **Enfoques de predicción basados en Meta-Modelos propietarios**: Los en-

foques en este grupo apuntan a las predicciones de rendimiento de tiempo de diseño. En lugar de usar UML como lenguaje de modelado para desarrolladores y arquitectos, estos enfoques tienen meta-modelos propietarios[4].

  - CBML - *Component-Based Modeling Language*
  - PECT - *The Prediction Enabled Component Technology*
  - COMQUAD - *Components with Quantitative properties and Adaptivity*
  - KLAPPER
  - ROBOCOP
  - Palladio

- **Enfoques de predicción centrados en *middleware***: hacen énfasis en la influencia del *middleware* en el rendimiento de un sistema basado en componentes. Por lo tanto miden y modelan el rendimiento de plataformas *middleware* como JavaEE o .Net. Se basan en suponer que la lógica de negocio de los componentes tiene poco impacto en el rendimiento general del sistema y por eso no requieren un modelado detallado.

  - NICTA

- **Enfoques basados en especificaciones formales**: estos enfoques siguen teorías fundamentales de especificación de rendimiento y no toman en cuenta marcos de trabajo (*frameworks*) de medición y predicción.

  - RESOLVE
  - HAMLET

- **Enfoques de monitoreo para sistemas implementados**: suponen que un sistema basado en componentes ha sido implementado y puede ser pro-

bado. El objetivo es encontrar problemas de rendimiento en un sistema en ejecución, identificar cuellos de botella y adaptar el sistema para que pueda lograr los requerimientos de rendimiento.

- COMPAS
- TESTEJB
- AQUA
- PAD

Enfoques Suplementarios

- **Enfoques de monitoreo para componentes implementados**: El objetivo de los enfoques de medición para implementaciones de componentes de software individuales es derivar especificaciones de rendimiento parametrizadas a través de mediciones múltiples. El objetivo es obtener el perfil de uso, dependencias y la plataforma de implementación a partir de la especificación de rendimiento, de modo que pueda usarse en diferentes contextos.
  - RefCAM
  - COMAERA
  - ByCounter
- **Enfoques de predicción con énfasis en conectores de componentes**: Estos enfoques suponen un lenguaje de descripción de componentes existente y se centran en modelar y medir el impacto en el rendimiento de las conexiones entre componentes. Estas conexiones pueden implementarse con diferentes técnicas *middleware*.

- Verdickt
- Grassi
- Becker
- Happe

Recientemente varios enfoques de predicción basados en meta-modelos propietarios han sido propuestos para la optimización del diseño de arquitecturas, modelado de calidad de servicio y escalabilidad. PerOpteryx[11] es un enfoque de optimización de diseño de arquitecturas que manipula modelos especificados en *Palladio Component Model*[6] y utiliza el algoritmo evolutivo multiobjetivo `NSGA-II`. Para análisis de rendimiento utiliza redes de colas en capas. *Descartes Modeling Language*[12] es un lenguaje de modelado de arquitecturas para modelar calidad de servicio y aspectos relacionados con la gestión de recursos de los sistemas, las infraestructuras y los servicios de tecnología de información dinámicos modernos. *CloudScale*[1] [13] es un enfoque de diseño y evolución de aplicaciones y servicios escalables en la nube. En *CloudScale* se identifica y gradualmente se resuelven problemas de escalabilidad en aplicaciones existentes y también permite el modelado de alternativas de diseño y el análisis del efecto de la escalabilidad en el costo. Cabe mencionar que estos últimos enfoques han sido influenciados en gran medida por el trabajo llevado a cabo en *Palladio Component Model*.

Modelado de Arquitecturas de Software con *Palladio Component Model*

El *Palladio Component Model* es un enfoque de modelaje para arquitecturas de software basados en componentes que permite predicción de rendimien-

[1] `http://www.cloudscale-project.eu/`

to basada en modelos. PCM contribuye al proceso de desarrollo de ingeniería basada en componentes y proporciona conceptos de modelaje para describir componentes de software, arquitectura de software, despliegue (*deployment*) de componentes y perfiles de uso de sistemas de software basados en componentes en diferentes submodelos (Figura 2.2).

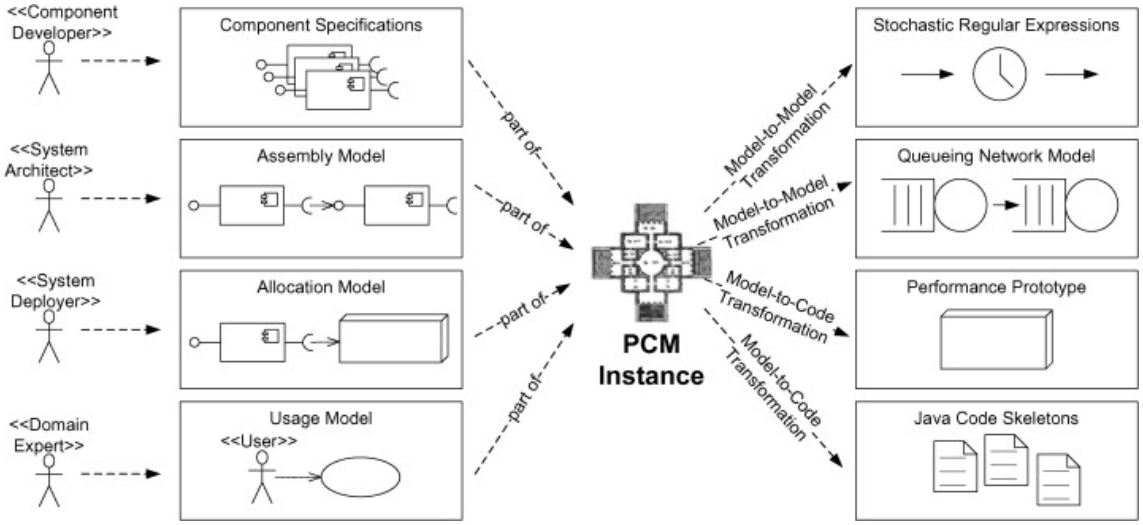

Figura 2.2: Instancia de un modelo PCM. Tomado de [14]

- **Especificaciones de componentes** son descripciones abstractas y paramétricas de los componentes de software. En las especificaciones de software se proporciona una descripción del comportamiento interno del componente así como las demandas de sus recursos en RDSEFFs (*Resource Demanding Service EFFect specifications*) utilizando una sintaxis similar a los diagramas de actividad de UML.

- **Un modelo de ensamblaje** (*assembly model*) especifica qué tipo de componentes se utilizan en una instancia de aplicación modelada y si las instancias del componente se replican. Además, define cómo las instancias del componente se conectan, para representar la arquitectura de software.

- El entorno de ejecución y los recursos, así como la instalación (*deployment*) de instancias de componentes para dichos contenedores de recursos se definen en un **modelo de asignación** (*allocation model*).

- El **modelo del uso** especifica la interacción de los usuarios con el sistema utilizando una sintaxis similar al diagrama de actividades de UML, para proporcionar una descripción abstracta de la secuencia y la frecuencia con que los usuarios activan las operaciones disponibles en un sistema.

Un modelo PCM abstrae un sistema de software en el nivel de arquitectura y se anota con consumos de recursos que fueron medidos previamente y otros que son estimados. El modelo puede entonces ser usado en transformaciones de modelo-a-modelo o modelo-a-texto a un modelo de análisis en particular (redes de colas o simulación de código) que puede ser resuelto analíticamente o mediante simulación para obtener resultados sobre el rendimiento y predicciones del sistema modelado. Los resultados del rendimiento y las predicciones pueden ser utilizadas como retroalimentación para evaluar y mejorar el diseño inicial, permitiendo así una evaluación de calidad de los sistemas de software con base en un modelo[8].

Modelado de Arquitecturas de Software con *Descartes Modeling Language*

El *Descartes Modeling Language* (DML) es un lenguaje de modelado de nivel de arquitectura utilizado para describir calidad de servicio (*QoS* por sus siglas en inglés) y aspectos relacionados con la gestión de recursos de sistemas de información dinámicos, infraestructuras y servicios. DML distingue explícitamente diferentes tipos de modelos que describen el sistema y sus procesos de adaptación desde un punto de vista técnico y lógico. Juntos, estos diferentes

tipos de modelos forman una instancia DML. La idea detrás del uso de estos modelos es separar el conocimiento acerca de la arquitectura del sistema y el comportamiento de su rendimiento (aspectos técnicos), del conocimiento de los procesos de adaptación del sistema (aspectos lógicos)[12].

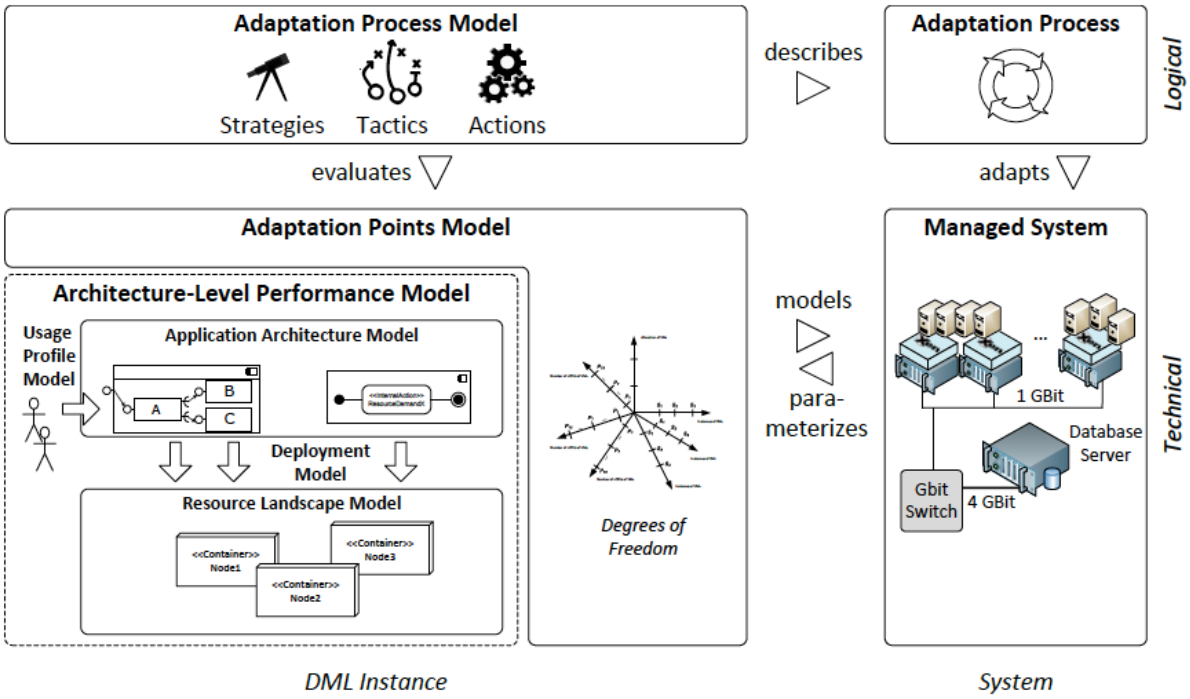

Figura 2.3: Relación de los diferentes modelos de una instancia DML y el sistema. Tomado de [12]

La figura 2.3 muestra la relación de los diferentes modelos que son parte de una instancia DML, el sistema gestionado (*managed system*) y el proceso de adaptación del sistema (*adaptation process*). En la esquina inferior derecha de la figura 2.3, se ve el sistema, el cual es gestionado por un proceso de adaptación, mostrado en la esquina superior derecha de la figura 2.3. En la esquina inferior izquierda se muestran modelos que reflejan los aspectos técnicos del sistema. Esos aspectos son los recursos de hardware y su distribución (*resource landscape model*), los componentes de software y el comportamiento relevante al rendimiento (*application architecture model*), la instalación de componentes de software en el hardware (*deployment model*), el comportamiendo del uso y

las cargas de trabajo de los usuarios del sistema (*usage profile model*) y los grados de libertad del sistema que pueden ser empleados para la adaptación del sistema en ejecución (*adaptation points model*). Por encima de estos modelos (esquina superior izquierda de la figura 2.3) se muestra el modelo de proceso de adaptación, que especifica un proceso de adaptación que describe cómo el sistema se adapta a cambios en su ambiente. El proceso de adaptación aprovecha las técnicas de predicción de rendimiento en línea para razonar sobre posibles estrategias, tácticas y acciones de adaptación.

2.1.3. Computación en nube

Según el NIST[15], el *Cloud Computing*, o la Computación en la Nube, es un modelo que, desde la perspectiva del consumidor, permite el acceso conveniente, por demanda, desde alguna red, a un conjunto de recursos computacionales específicos y configurables (por ejemplo, redes, servidores, almacenamiento, aplicaciones y servicios) que pueden ser rápidamente provisionados y entregados con un esfuerzo mínimo de administración o de interacción con el proveedor del servicio.

Modelos de entrega de servicio

La definición del NIST presenta 3 modelos básicos de servicio. Aunque los proveedores de servicios de Cloud han creado muchas variaciones de ofertas *"as-a-Service"*, estos 3 siguen siendo considerados los fundamentales. En la figura 2.4 se muestra lo que se conoce como infraestructura como servicio (IaaS), plataforma como servicio (PaaS) y software como servicio y (SaaS) y las áreas que abarcan en términos de infraestructura, plataforma y software dentro de

un esquema de computación en la nube.

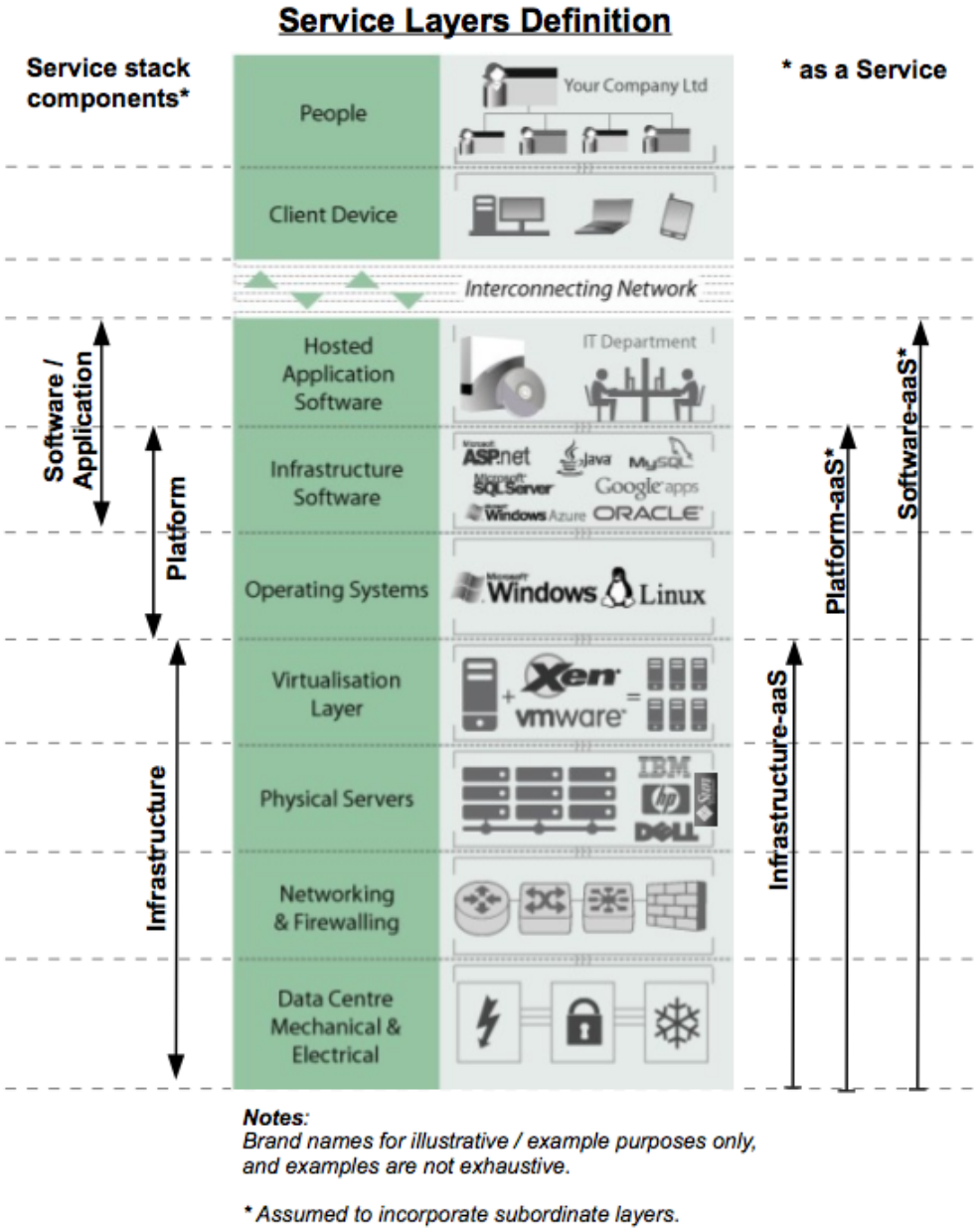

Figura 2.4: Niveles de servicio presentes en computación en la nube. Tomado de [16]

Software como Servicio (SaaS) El cliente usa una aplicación, que se ofrece como un servicio accedido remotamente con protocolos estandarizados, pero no controla el sistema operativo, los servidores de aplicación, el hardware o la infraestructura de red en la que opera. El proveedor instala, administra y mantie-

ne el software. El proveedor no necesariamente es dueño de la infraestructura física donde el software se está ejecutando. En general, se puede considerar un servicio para usuarios finales.

Plataforma como Servicio (PaaS) El cliente usa un sistema hospedado para sus aplicaciones. El cliente controla las aplicaciones que ejecuta en el ambiente (y posiblemente tenga algún control sobre el ambiente de hospedaje), pero no tiene acceso ni controla el sistema operativo, el hardware o la infraestructura de red donde ellas se ejecutan. La plataforma es típicamente un marco de ejecución aplicativo. El proveedor administra la infraestructura de software de la nube para la plataforma. En este caso, el cliente por lo general es un desarrollador de software.

Infraestructura como Servicio (IaaS) El cliente utiliza recursos computacionales fundamentales como poder de procesamiento, almacenamiento, comunicaciones y *middleware*. El cliente puede controlar el sistema operativo, el almacenamiento, implementar aplicaciones y posiblemente algunos componentes de red como cortafuegos (*firewalls*) y balanceadores de carga, pero la infraestructura física de red y de hardware no le pertenece. Los clientes habituales de estos servicios son administradores de sistemas.

Serverless y Function-as-a-Service

FaaS es un área de lo que hoy se conoce como computación sin servidores o *serverless computing*. Amazon.com, uno de los principales propulsores de esta tendencia define *serverless* como una tecnología la cual permite construir y ejecutar aplicaciones y servicios sin necesidad de pensar en los servidores. Las

aplicaciones *serverless* no requieren de aprovisionamiento, escalamiento y administración de ningún servidor. Se puede construir con ella casi cualquier tipo de aplicación o servicio, y todo lo que se requiere para ejecutar y escalar la aplicación con alta disponibilidad es gestionado por el proveedor del servicio[17].

FaaS se refiere a un tipo de aplicación *serverless* en donde la lógica del lado del servidor es escrita por un desarrollador pero, a diferencia de arquitecturas tradicionales, esta se ejecuta en contenedores de cómputo que no mantienen estado, son activados por medio de eventos, son efímeros (pueden durar solo una invocación) y son totalmente administrados por un tercero[18].

Cabe destacar que aunque el término *serverless* ha llegado a ser utilizado para referirse de forma directa a plataformas FaaS, hoy en día las aplicaciones de *serverless computing* abarcan otros tipos de servicios como almacenamiento, mensajería, análisis de datos, seguridad, entre otros.

Proveedores de servicios de *Function-as-a-Service*

Los principales proveedores de servicios de FaaS son: AWS Lambda, Google Functions, Microsoft Azure Functions e IBM Apache OpenWhisk Functions[19].

AWS Lambda (`https://aws.amazon.com/lambda/`) Proporcionado por Amazon Web Services. Soporta varios lenguajes como NodeJS, Java, C# y Python. Rango de asignación de memoria: mínimo 128Mb, máximo 1536Mb. Capacidad de disco (espacio en `/tmp`) 512Mb. La duración máxima de ejecución por solicitud es de 300 segundos.

El precio se establece en $0.20 por millón de solicitudes y $0.00001667 por

Gigabyte por segundo. 1 millón de solicitudes y 400.000 GBs por mes son gratuitos.

Google Functions (`https://cloud.google.com/functions`) Solamente soporta NodeJS. Está limitado a 1000 funciones con una duración máxima de 540 segundos por cuenta. El precio se establece en tres modelos diferentes. El primero es de $ 0,40 por millón de invocaciones, pero considera que 2 millones de invocaciones son gratuitas. El segundo modelo de precio es de $0.0000025 por GBs con 400,00 GBs por mes son gratis. El tercer modelo de precios, es de $0.0000100 GHz por segundo con 200,000 GHz por segundo al mes que son gratuitos.

Microsoft Azure Functions (`https://azure.microsoft.com/en-us/services/functions`) Soporta lenguajes como NodeJS, C#, F#, Python, PHP y Java. Azure está limitado a solo diez ejecuciones simultáneas por una función. No hay límites en el límite máximo de tiempo de ejecución. Utiliza dos modelos de precios diferentes. El primero es de $0.000016 GBs, con 400,000 GBs por mes que son gratuitos. El segundo modelo es de $0.20 por millón de ejecuciones, con 1 millón de ejecuciones por mes de forma gratuita.

IBM Apache OpenWhisk Functions (`https://www.ibm.com/cloud/functions`) Soporta lenguajes como NodeJS, Swift, Java, PHP, Go y Python. Se puede utilizar cualquier otro lenguaje de programación siempre y cuando de proporcione un contenedor de Docker para esto. Tiene un costo básico que es de $0.000017 por segundo de ejecución, por GB de memoria asignada.

Otros proveedores de servicios FaaS Se pueden encontrar otros proveedores y herramientas para utilizar el modelo de funciones en la nube. Entre ellos se encuentran:

- Apache OpenWhisk: `http://openwhisk.incubator.apache.org/`
- Webtask: `https://webtask.io/`
- OpenFaas: `https://github.com/openfaas/faas`
- IronFunctions: `https://github.com/iron-io/functions`
- Kubeless: `http://kubeless.io/`
- Fn Project: `http://fnproject.io/`
- Spotinst Functions: `https://spotinst.com/products/spotinst-functions/`

2.2. Trabajos relacionados

Ingeniería de rendimiento de software en aplicaciones en la nube

El estilo de arquitectura basado en microservicios es uno de los que ha logrado ganar mayor adopción y popularidad dentro de la comunidad de desarrolladores. Los microservicios, son una arquitectura de software que involucra la construcción y entrega de sistemas que se caracterizan por ser servicios pequeños, granulares, independientes y colaborativos [20]. Con respecto de SPE y microservicios se reporta que existen muchos retos en investigación que han sido poco o nada abordados.

En [2] se planean el monitoreo, pruebas y modelado del rendimiento como las tres áreas en donde se carece de investigación y desarrollo de SPE y microservicios. Se argumenta que aún hacen falta enfoques de SPE que tomen en cuenta las particularidades de los microservicios. Aderaldo et al.[21] señala una falta de investigación empírica repetible sobre diseño, desarrollo y evaluación de aplicaciones de microservicios y que esto dificulta la evaluación de este tipo de aplicaciones pues se cuenta con muy pocas aplicaciones y arquitecturas de referencia, así como de cargas de trabajo que contribuyan a caracterizar comportamiento.

El reporte de [22] también proporciona un listado de los retos asociados con microservicios y SPE, haciendo énfasis en actividades de SPE relacionadas con la integración de actividades de desarrollo y de operaciones de puesta en producción y mantenimiento del software. Se argumenta que a pesar del alto nivel de adopción de prácticas de integración continua, entrega continua y DevOps por la comunidad de ingeniería de software, ninguna toma en cuenta aspectos

relacionados con el rendimiento. Otros estudios, como el llevado a cabo en [23], indican que uno de los atributos de calidad que ha recibido mayor atención en la investigación en microservicios es el de la eficiencia del rendimiento pero vista mayoritariamente desde el punto de vista de la escalabilidad y mantenibilidad del código de los microservicios y su instalación.

Serverless y Function-as-a-Service

Los investigadores han empezado a describir y analizar FaaS a través de encuestas y experimentos[1, 24, 25], y también por análisis económicos[26, 27]. Sin embargo se reporta que aún no se conoce mucho acerca de SPE en FaaS. En [2] se menciona que al igual que con microservicios, FaaS también requiere de nuevas estrategias de modelado para capturar el comportamiento del código bajo estas infraestructuras. Los modelos de rendimiento tradicionales basados en la noción de máquinas independientes podría ser inadecuado.

En van Eyk et al.[28] se presenta un informe confeccionado por el *Standard Performance Evaluation Corporation RG Cloud Group*[2] (SPEC RG Cloud) sobre desafíos asociados a rendimiento en arquitecturas FaaS. Los principales temas son los que tienen que ver con evaluación y comparación de plataformas de FaaS, reducción del *overhead*, políticas de *scheduling*, la relación costo-rendimiento de una función y la predicción del rendimiento. El informe también señala que actualmente muchas de las tareas de evaluación y pruebas para FaaS se vuelven complicadas porque no se cuenta con aplicaciones, arquitecturas ni cargas de trabajo de referencia; este es un trabajo que pretende abordar el SPEC RG Cloud. Con respecto a predicción del rendimiento, se indica que la aplicación de modelos de rendimiento de sistemas de software tradicionales

[2] https://research.spec.org/working-groups/rg-cloud.html

en FaaS trae nuevos retos como la brecha de información (*information gap*) y el rendimiento de una función en particular. La brecha de información significa que el usuario de FaaS no está consciente de los recursos de hardware en los que las funciones son ejecutadas, mientras que, por otro lado, la plataforma de FaaS no tiene información acerca de los detalles de la implementación de la función. Tal y como en las aplicaciones tradicionales, la entrada (tamaño, estructura y contenido) influyen en el rendimiento de una función, el hecho de tener una infraestructura oculta hace necesario encontrar nuevos modelos que logren predecir de forma precisa el rendimiento de una función. Técnicas de modelado desarrolladas para sistemas de software se podrían aprovechar para FaaS, como por ejemplo modelado y simulación de arquitecturas de software basada en componentes.

La aplicación de *serverless computing* es una área activa de desarrollo. En trabajos previos [29, 30] se han estudiado arquitecturas alternativas de *serverless computing* con el fin de explotar aspectos de rendimiento y/o abordar retos técnicos que otras plataformas no han hecho. También se han investigado arquitecturas para recuperación de información[24] y *chatbots*[31] utilizando plataformas *serverless*. Muchas otras aplicaciones se han venido desarrollando en campos como *Machine Learning*, seguridad, Internet de las cosas, procesamiento de voz, sistemas de archivos, etc, y son solo una muestra del potencial de esta tecnología. Pese a este potencial, también se reporta que aún no se sabe mucho acerca de cuáles herramientas y arquitecturas tecnológicas se usan para producir, instalar y ejecutar funciones[32]. En [25] se examinan factores qué pueden influir en el rendimiento de plataformas de *serverless computing*. De acuerdo con esto, se logran identificar cuatro estados de una infraestructura *serverless*: *provider cold*, *VM cold*, *container cold*y *warm*. Además demuestra cómo el rendimiento de los servicios puede llegar a variar hasta 15 veces según

estos estados.

Posicionamiento de la investigación con respecto de la literatura consultada La investigación que se propone en este documento, pretende dar un aporte al área de ingeniería de rendimiento de software en aplicaciones en la nube. De acuerdo con el material recolectado, se pudo conocer que existe una necesidad por aplicar enfoques de modelado de rendimiento en el desarrollo de sistemas modernos pero que, por otro lado, los esfuerzos que se han llevado a cabo para esto no han logrado ganar popularidad, o bien, no consideran las particularidades de la computación en la nube. Por ejemplo, se reporta que no existen enfoques de modelado de rendimiento para microservicios, el cual es hoy en día, un estilo de arquitectura de software sumamente popular.

Es por esto que se considera que la realización de un estudio exploratorio para determinar los factores que influyen en el rendimiento de una función en la nube, puede brindar nuevo conocimiento sobre cómo aplicar enfoques de ingeniería de rendimiento conocidos a estas aplicaciones y además podría representar un marco de referencia inicial por medio del cual se pueda evaluar la adopción de estas tecnologías *a priori* y fundamentar el planeamiento de capacidades al dimensionar sistemas intensivos en software que vayan a usar microservicios en ambientes en la nube.

Capítulo 3

Definición del problema

De acuerdo con la revisión de la literatura, se carece de modelos de rendimiento que contribuyan a caracterizar el comportamiento de funciones en la nube alojadas en plataformas FaaS bajo distintas cargas de trabajo. Contar con tales modelos permitiría validar si las funciones en la nube pueden cumplir criterios de calidad de servicio especificados.

En las plataformas FaaS en las que se ejecutan funciones en la nube, la infraestructura tecnológica subyacente se oculta por completo de los desarrolladores y diseñadores. El conocimiento de la influencia de esta infraestructura y su configuración es vital para que los arquitectos de software puedan obtener predicciones significativas del comportamiento de una función pues, al omitirse la influencia que esta tiene, puede conducir a la generación de predicciones erróneas con respecto del rendimiento de una función. Una función que reporte tiempos de respuesta sumamente prolongados o bien la utilización de significativas cantidades de recursos puede generar grandes costos económicos y hasta llegar a ser rechazada por la plataforma FaaS.

Capítulo 4

Justificación

4.1. Innovación

Los aspectos más novedosos que se aportarán en esta tesis serán:

1. Proponer un método mediante el cual se pueden obtener estimaciones del rendimiento de una función en la nube

2. Realizar modelado y simulación basados en componentes para caracterizar el rendimiento de funciones en la nube sobre plataformas FaaS

3. Proporcionar, a partir de lo anterior, un modelo del rendimiento de una función en la nube

Con respecto de (1), en [28] se reporta que la predicción del rendimiento es uno de los principales retos de investigación en plataformas FaaS y que no se cuenta con modelos de rendimiento que consideren las características de FaaS. Esto abre la posibilidad de explorar la aplicabilidad de técnicas conocidas de

modelaje de rendimiento en sistemas de software a nuestro dominio de problema.

(2) El modelado y simulación de arquitecturas de software basadas en componentes, representa una alternativa atractiva para abordar este problema, pues es un enfoque en donde hay una comunidad de investigación y desarrollo activa que ha logrado generar diferentes tipos de herramientas para la estimación del rendimiento de sistemas informáticos.

(3) La obtención de un modelo de rendimiento de una función en la nube (3), representaría un aporte relevante para SPE y FaaS porque significaría que existe una forma por medio de la cual evaluar el comportamiento de una función sin que esta esté necesariamente instalada en una plataforma FaaS y además permitiría que cambios futuros que necesite esa función puedan ser modelados *a priori*, simularlos y obtener predicciones del impacto de los cambios.

4.2. Impacto

En distintos reportes sobre el estado de tecnologías *serverless* en la industria [33, 34, 35, 36] se señala que la adopción de este tipo de tecnologías va en franco aumento. En [34] se reporta una tasa de crecimiento del 25 % en su adopción con respecto del 2017. Cloud Foundry Foundation [36] indica que el 2017 "la mayoría" de sus encuestados no estaban usando tecnologías *serverless* mientras que para el 2018, solamente 43 % **no** lo estaba haciendo. En el informe anual del estado de tecnologías en la nube de DZone[33] se notó un incremento del 14 % con respecto del 2017 en el uso de tecnologías *serverless*.

Pese a que los niveles de adopción de esta tecnología van en aumento, estos

mismos reportes señalan incovenientes tales como:

1. Se tiene que depender de los niveles de servicio de un proveedor

2. Dificultad para monitorear y depurar

3. Preocupación por parte de los desarrolladores por el "cómo funciona" la función en la plataforma FaaS: ¿se estarán asignando los recursos adecuados para mi función en la plataforma FaaS? ¿Cómo y cuándo se hace?

4. Límites de tiempo de espera: dependiendo del tiempo de ejecución una función en la nube podría llegar a ser cancelada por la propia plataforma FaaS

Lo anterior refleja que aún existe una especie de "área gris" alrededor del uso de las tecnologías *serverless* y en particular funciones en la nube. Esto es lo que van Eyk et al.[28] llama brecha de la información: el usuario de FaaS no está consciente de los recursos de hardware en los que las funciones son ejecutadas, mientras que por otro lado, la plataforma de FaaS no tiene informacion acerca de los detalles de la implementacion de la funcion.

Analizar el rendimiento de una función en la nube y obtener un modelo de este contribuiría a tener un mejor entendimiento de esta tecnología y de cómo esta es gestionada por la plataforma FaaS. Esto permitiría a los arquitectos y diseñadores tener mayor control sobre los cambios en una función para que esta no solamente pueda cumplir con los requerimientos de calidad de servicio sino que también, al ser *serverless* un servicio que se cobra por demanda, colaboraría a reducir gastos, ya que una función que tenga un tiempo de ejecución menor generá menores costos por el uso de la plataforma FaaS.

4.3. Profundidad

Para lograr obtener un modelo de rendimiento de una función en la nube se plantean las siguientes actividades:

- Diseño e implementación de un caso de uso de referencia de una función en la nube
- Obtención un modelo:
  - Realizar pruebas de carga sobre la función en la nube seleccionada
  - Obtener métricas asociadas al rendimiento a partir de bitácoras de ejecución de la función
  - Utilizar las métricas obtenidas para suministrarlas como entrada a una herramienta de extracción de modelos de rendimiento. La herramienta generará como resultado un modelo de rendimiento
- Diseñar experimentos y ejecutar simulaciones sobre el modelo obtenido con el fin de validar las estimaciones o determinar si es necesario calibrar el modelo.

Capítulo 5

Hipótesis

En el modelo FaaS, la infraestructura tecnológica subyacente se oculta por completo de los diseñadores y desarrolladores, asimismo, la duración de la ejecución de una función en la nube determina el costo del servicio: entre mayor sea la duración, mayor será el costo y viceversa. Si bien se han realizado estudios [25] para evaluar factores que influyen en el rendimiento de servicios basados en computación *serverless*, el modelado del rendimiento de este tipo de aplicaciones se sigue presentando como un reto de investigación [2]. El modelado del rendimiento ha ganado considerable atención en la comunidad de ingeniería de rendimiento en las pasadas dos décadas, principalmente en sistemas de software basados en componentes [4]. Al finalizar y alcanzar los objetivos de la investigación se podrá contar con un marco de referencia que permitirá:

- Tener una función en la nube que servirá como prueba de concepto funcional para la evaluación del rendimiento
- Contar un método mediante el cual se pueda analizar el rendimiento de una función en la nube por medio de modelado y simulación.

Una vez alcanzados los objetivos, será posible responder a la pregunta:

¿Es posible estimar el rendimiento de una función en la nube por medio de modelado y simulación basados en componentes?

Capítulo 6

Objetivos

6.1. Objetivo general

Diseñar un método para modelar el rendimiento de funciones en la nube alojadas en plataformas *Function-as-a-Service* por medio del modelado y la simulación basados en componentes, con el fin de evaluar los factores que pueden influir en su comportamiento.

6.2. Objetivos específicos

1. Revisar el estado del arte de trabajos relacionados con enfoques de predicción y medición del rendimiento en sistemas de software como servicio.

2. Sintetizar un caso de uso de una función en la nube considerada como de referencia, con el propósito de analizar su comportamiento.

3. Elaborar, conforme a un diseño experimental, pruebas de rendimiento so-

bre el caso de uso seleccionado a fin de obtener datos base.

4. Analizar los datos experimentales mediante herramientas de extracción de modelos de rendimiento de software.

5. Proponer y validar modelos de rendimiento a partir de los experimentos realizados.

6. Formular una guía metodológica para dar a conocer aspectos de rendimiento en funciones en la nube a partir de la experiencia obtenida.

Capítulo 7

Entregables

7.1. Revisión de literatura

Alineado con objetivo específico 1

Se pretende identificar los resultados de otros estudios relacionados con el modelado de rendimiento de software, así como retos y oportunidades de investigación que existan en esta área. Las preguntas de investigación inicialmente propuestas son las siguientes:

PI1 ¿Cuáles enfoques de predicción y medición del rendimiento en sistemas de software basados en componente se han propuesto?

PI2 ¿Cuáles enfoques de predicción y medición de rendimiento de software se han utilizado para aplicaciones en la nube?

PI3 ¿Qué retos y oportunidades existen con estos enfoques en la actualidad?

PI4 ¿Qué herramientas hay disponibles para el modelado de rendimiento de software?

7.2. Implementación de un caso de uso de función en la nube

Alineado con objetivo específico 2

Identificar un caso de uso de que sea considerado como de referencia o de utilidad común en donde las funciones en la nube hayan mostrado ser un buenas candidatas para su solución. Se procederá a implementar este caso de uso e instalarlo en una plataforma FaaS. Esta será la función que servirá como base para futuros análisis.

7.3. Pruebas sobre el caso de uso

Alineado con objetivo específico 3

7.3.1. Diseño experimental

Planeamiento y definición de las variables por observar de las pruebas de carga.

7.3.2. Pruebas de carga

Diseño, ejecución y análisis de los resultados de las pruebas.

7.4. Modelado y análisis del rendimiento de la función

Alineado con objetivo específico 4

Se tomarán los datos generados por las pruebas anteriores para utilizarlas como entrada y, a partir de ellos, realizar una labor de análisis y modelado.

7.5. Modelo de rendimiento de la función en la nube

Alineado con objetivo específico 5

A partir de la función propuesta, pruebas y análisis del rendimiento por medio de herramientas de modelado, se propondrá un modelo que logre caracterizar el comportamiento de la función.

7.5.1. Simulaciones sobre el modelo propuesto

Para validar el modelo propuesto, se ejecutarán simulaciones sobre él y de esta forma determinar si los resultados de las simulaciones se corresponden con los obtenidos en las pruebas de rendimiento de la función.

7.6. Guía metodológica

Alineado con objetivo específico 6

Una vez realizado el estudio, se estructurará una descripción del método utilizado para estimar el rendimiento de una función en la nube.

Capítulo 8

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)[37] 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[38] para su plataforma FaaS, AWS Lambda. Dentro de estas arquitecturas se destaca el caso de uso de un manejador de imágenes (*Image Handler*)[39].

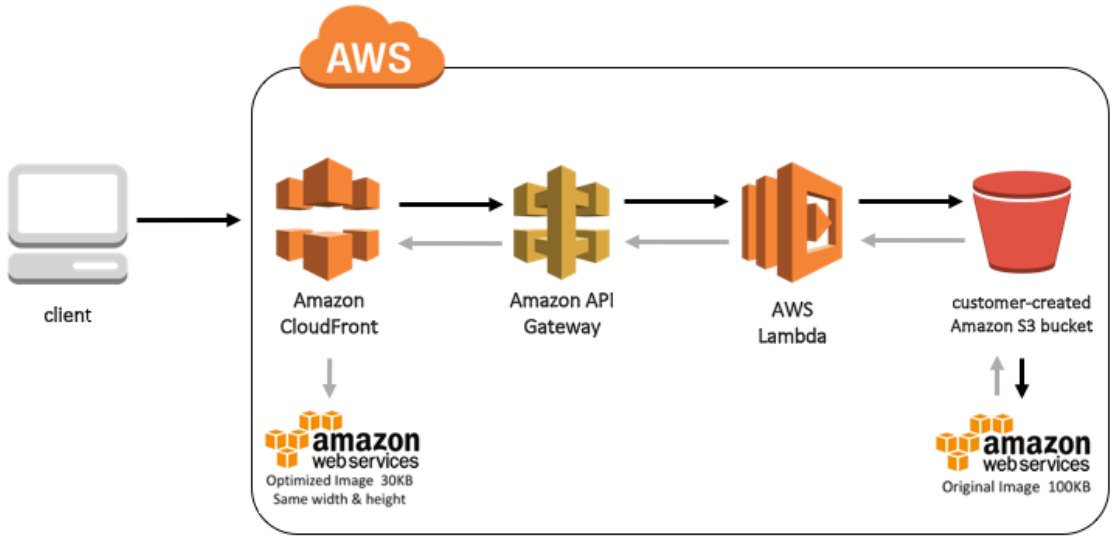

Figura 8.1: Arquitectura del manejador de imágenes. Tomado de [39]

[1] En la plataforma AWS Lambda

8.1. *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 8.1:

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.

8.1.1. Manejador de imágenes para SPE

Para este estudio se proponemos implementar una variación del manejador de imágenes de la sección 8.1, que se muestra en la figura 8.2.

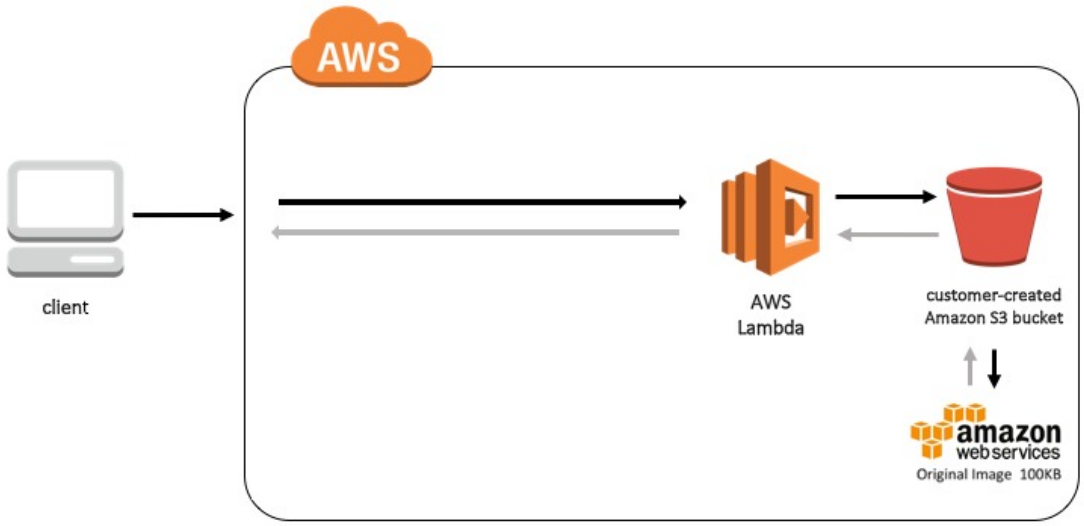

Figura 8.2: Arquitectura del manejador de imágenes propuesto para el estudio.

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 8.3

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).

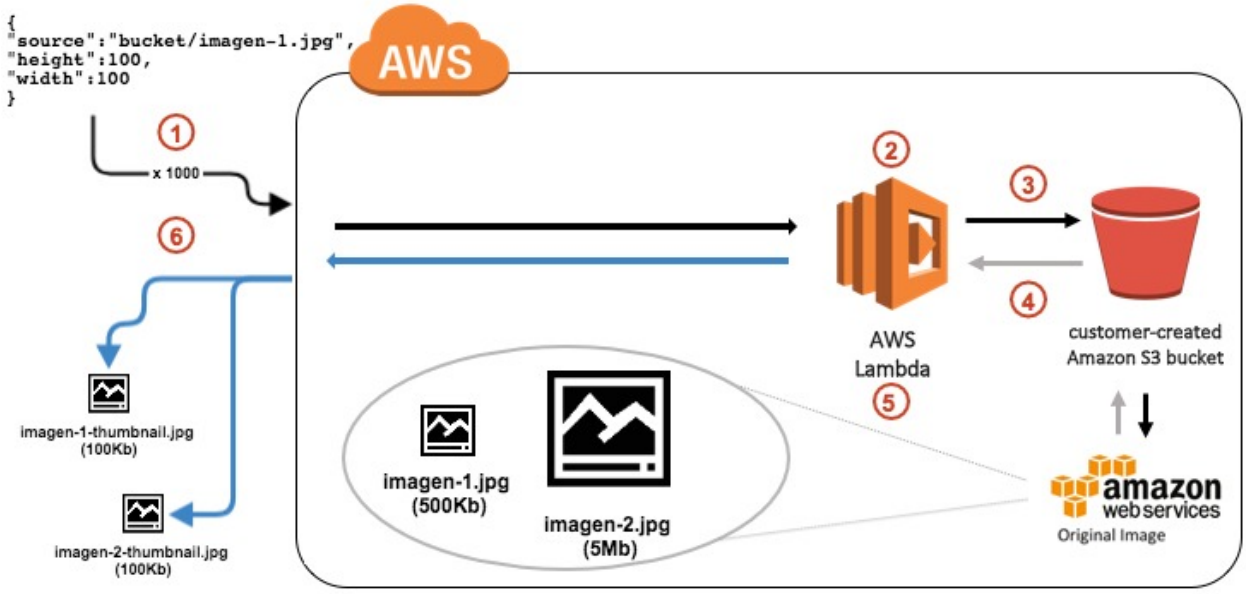

Figura 8.3: Carga de trabajo sugerida para el manejador de imágenes

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 8.3 muestra una sugerencia de dos posibles cargas de trabajo:

1. 100 solicitudes de cambio de tamaño de una imagen grande. En la figura 8.3, `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 8.3, `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 [37], 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 2.1.3) 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.

8.2. 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 8.1:

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]. 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.

8.2.1. 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/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 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 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 8.1. 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 8.1: 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 8.2. 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 8.2: Clase `ImageHandler.java`

Principales interacciones dentro de *Image-Handler*

La figura 8.4 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 8.2.1 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.

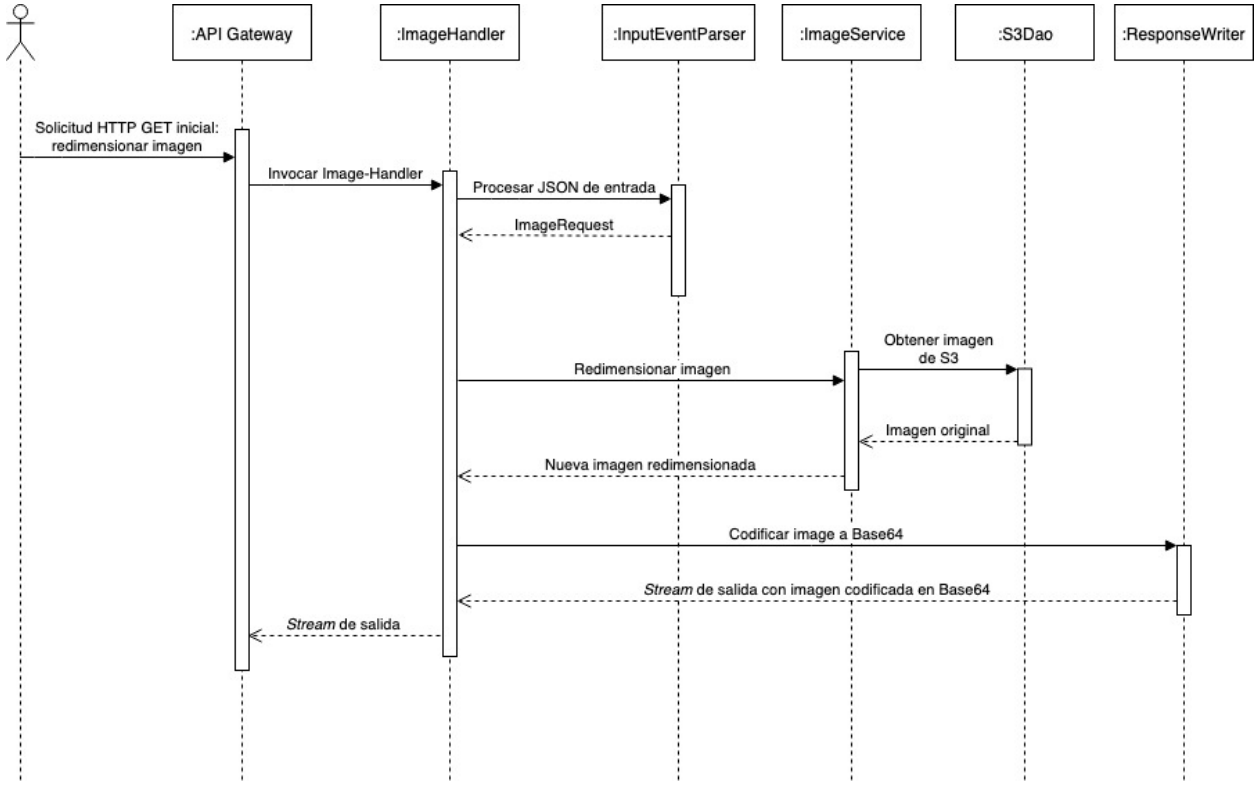

Figura 8.4: Secuencia de acciones llevadas a cabo por *Image-Handler*

8.2.2. 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 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 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 [40].

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[41]. 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 8.3 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
        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)",
17                      OperationExecutionRecord.NO_SESSION_ID,
18                      OperationExecutionRecord.NO_TRACE_ID,
19                      tin, tout,
20                      InetAddress.getLocalHost().getHostName(),
21                      0,
22                      0);
23          MONITORING_CONTROLLER.newMonitoringRecord(e);
24      }
25      .
26      .
27      .
28  }
```

Listing 8.3: Extracto de la clase `ImageHandler.java` instrumentalizada con Kieker

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.

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 8.3.

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 8.3 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 8.2.2 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 8.4. Cada invocación a las operaciones de procesamiento de la entrada, redimensionamiento y entrega de la respuesta se realizan utilizando el método `AWSRay.createSubsegment`.

```java
public class ImageHandlerXRay implements RequestStreamHandler {
    .
    .
    .
    @Override
    public void handleRequest(InputStream inputStream, OutputStream
    outputStream, Context context) throws IOException {
        ImageRequest imageRequest = AWSXRay.createSubsegment("input
    event", new Function<Subsegment, ImageRequest>() {
            @Override
            public ImageRequest apply(Subsegment subsegment) {
                return inputEventParser().processInputEvent(inputStream
    );
            }
        });

        InputStream imageResized = AWSXRay.createSubsegment("resize
    event", new Function<Subsegment, InputStream>() {
            @Override
            public InputStream apply(Subsegment subsegment) {
                return imageService().getImageFrom(imageRequest);
            }
        });

        AWSXRay.createSubsegment("write response", () -> {
            imageHandlerResponseWriter().writeResponse(imageResized,
    outputStream, imageRequest);
        });
    }
    .
    .
    .
}
```

Listing 8.4: Extracto de la clase `ImageHandler.java` instrumentalizada con AWS X-Ray

En la figura 8.5 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 desgloces de los subsegmentos que componen la traza.

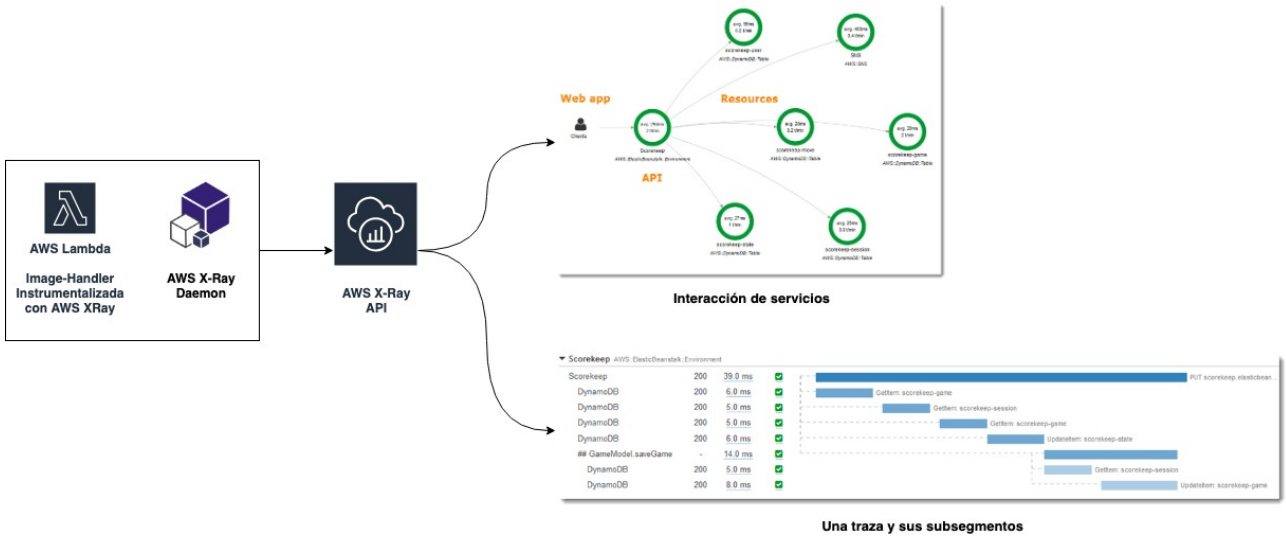

Figura 8.5: *Image-Handler* publicando eventos de rendimiento al servicio AWS X-Ray

8.3. 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, 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 la extracción del modelo PCM a partir de las bitácoras de Kieker, se llevaron a cabo las siguientes actividades:

1. Creación de la versión Image-PK (Sección 8.2.2): Versión de *Image-Handler*

con las bibliotecas de Kieker para generar bitácoras del rendimiento de la función.

2. Provisionar una nueva máquina virtual en AWS, en 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 indicar 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 en la máquina virtual creada en el punto #2. La figura 8.6 muestra los involucrados en el proceso de publicación de mediciones de rendimiento del código de *Image-Handler* hacia una bitácora externa.
5. Una vez obtenida una bitácora en formato Kieker, esta se usó como entrada para PMX. PMX inspecciona la bitácora, la procesa y retorna un archivo `.zip` con los archivos correspondientes a una instancia de PCM. Lo anterior se aprecia en la figura 8.7.

8.3.1. Modelo obtenido

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 redimiensión 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 las pruebas realizadas al modelo generado por PMX, se percató que si bien el modelo 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 acciones, estimaciones de rendimiento e invocaciones a otros componentes, por medio de *Service Effect Specifications* (SEFF).

En principio, las estimaciones incluidas en los SEFFs generados por PMX no lograron ser de utilidad para obtener predicciones representativas de lo observado en las ejecuciones de la función Lambda: mientras que en las invocaciones a *Image-Handler* se entregaban tiempos de respuesta distintos, las simulaciones sobre el modelo entregan siempre un dato fijo. Las estimaciones de rendimiento de los SEFFs se basaban en tiempos de cómputo constantes lo que hacía que el motor de simulaciones generara una predicción del tiempo de respuesta que era el mismo para todos los casos.

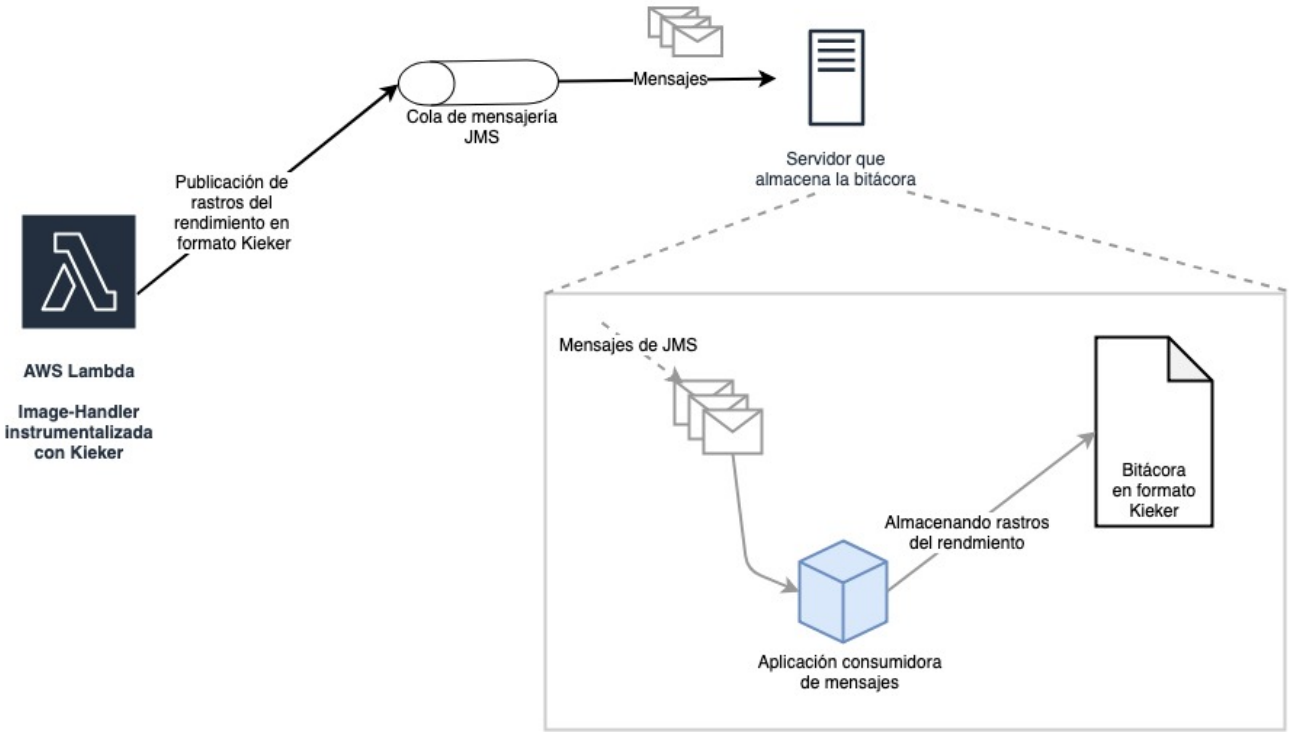

Figura 8.6: Publicando mediciones del rendimiento de la función Lambda.

Aporte de la versión IM-XRay a las simulaciones

Una actividad recurrente durante el modelado y simulación de arquitecturas de software, es la del afinamiento del modelo. Esta tarea es necesaria para

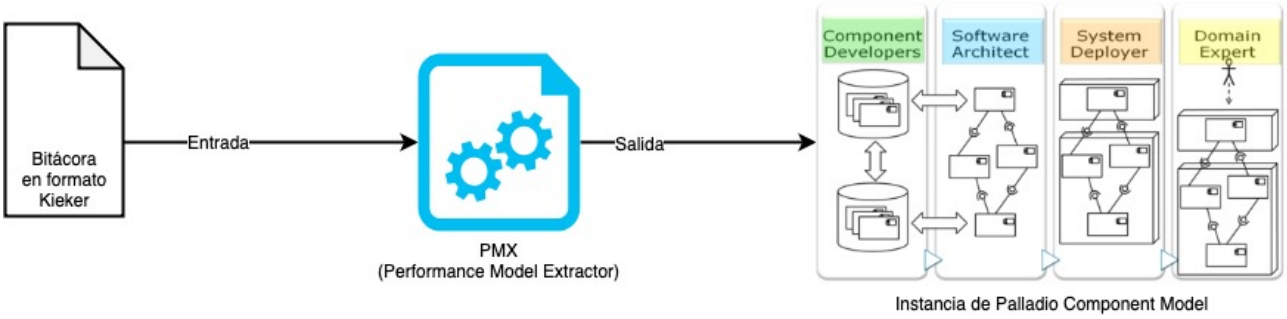

Figura 8.7: Convirtiendo una bitácora de Kieker a una instancia de PCM por medio de PMX.

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 8.2.2 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 podría llegar 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 alguna u otra forma no estaban "contando toda la historia" de lo que estaba pasando con la función Lambda. Por ejemplo, las 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 elegió Amazon X-Ray[8] 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

[8] `https://aws.amazon.com/xray/`

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 metadatos para analizar el comportamiento o identificar problemas.

Las trazas recolectadas en AWS X-Ray fueron de gran utilidad para refinar las estimaciones de rendimiento que cada uno de los componentes identificados realizaba. En el experimento #1, en la Sección 9.0.1, se tomaron los datos de rendimiento de cada componente y se realizaron sobre los mismos análisis de frencuencias con el fin de conocer cómo se distribuían estos datos con respecto a sus probabilidades. **Revisar párrafo con ITZ**. Las probabilidades fueron introducidas en los *SEFFs* de cada componente para la ejecución de simulaciones.

Capítulo 9

Diseño Experimental

En esta sección se detallan los experimentos realizados para:

- Validar si el modelo y la simulaciones sobre el mismo, logran caracterizar el comportamiento de la función *Image Handler* en distintos escenarios.
- Estudiar el comportamiento de la función Lambda cuando es invocada con cargas de trabajo y
- Comparar los resultados de las invocaciones de la función Lambda con los de la herramienta SAM CLI.

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

Este es el caso que se menciona en la Sección 8.1.1 y se muestra en la figura 8.3. Se realizaron invocaciones a la función Lambda con tres grupos de imágenes:

1. Imágenes de tamaño menor o igual a 500Kb.

2. Imágenes de tamaño mayor a 500Kb y menor a 1Mb.

3. Imágenes de tamaño mayor a 1Mb y menor a 2Mb.

En este experimento, el objetivo es comprobar por medio de mediciones directas y de simulaciones en un modelo, 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.

Invocaciones con imágenes menores a 500Kb

Para la realización de este experimento se contó con la siguiente configuración base:

- *Sujeto de prueba:* La función Lambda IM-Simple.

- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño menor a 500Kb alojadas en Amazon S3.

- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes con dimensiones aleatorias a IM-Simple.
- *Herramientas de medición:* Amazon Cloudwatch.

Configuración para la obtención de datos de rendimiento

- *Sujeto de prueba:* Las funciones Lambda IM-KP y IM-XRay
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño menor a 500Kb alojadas en Amazon S3.
- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes con dimensiones aleatorias a IM-PK y IM-XRay.
- *Herramientas de medición:* Kieker, PMX y Amazon X-Ray.

Para la realización de este experimento, se obtuvieron 1000 imágenes aleatorias de tamaño menor a 500Kb del servicio *Lorem Picsum*[1]. Se creó un *script* en `Bash` para acceder a la interfaz de programación (API) proporcionada por *Lorem Picsum* para descargar de forma aleatoria 1000 imágenes cuyo tamaño era menor a los 500Kb. La distribución del tamaño de las 1000 imágenes, en Kb, se aprecia en la figura 9.1. El mismo grupo de imágenes se utilizó para realizar solicitudes de redimensionamiento sobre IM-Simple, IM-KP y IM-XRay.

Medición Base: 1000 invocaciones de redimensionamiento de imágenes en IM-Simple. Se creó un *script* en `Bash` para ejecutar 1000 invocaciones de redimensionamiento en la función IM-Simple en las imágenes de tamaño menor a

[1] `https://picsum.photos`

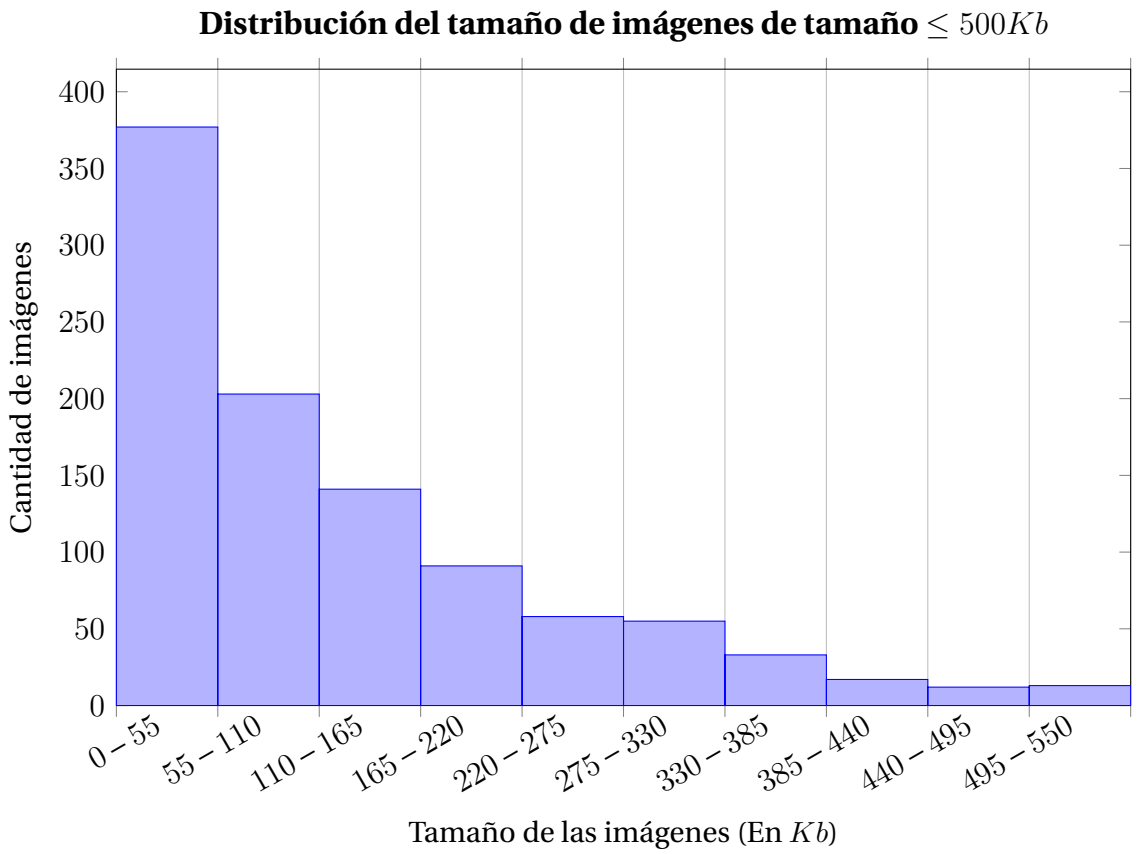

Figura 9.1: Distribución del tamaño de imágenes $\leq 500Kb$

500Kb. El *script* selecciona una imagen de forma aleatoria y luego ejecuta la solicitud de redimensionamiento utilizando dimensiones de ancho y alto de uso común para imágenes en miniatura (*thumbnails*) **AGREGAR APARTADO SOBRE LA ELECCION LOS THUMBNAILS**.

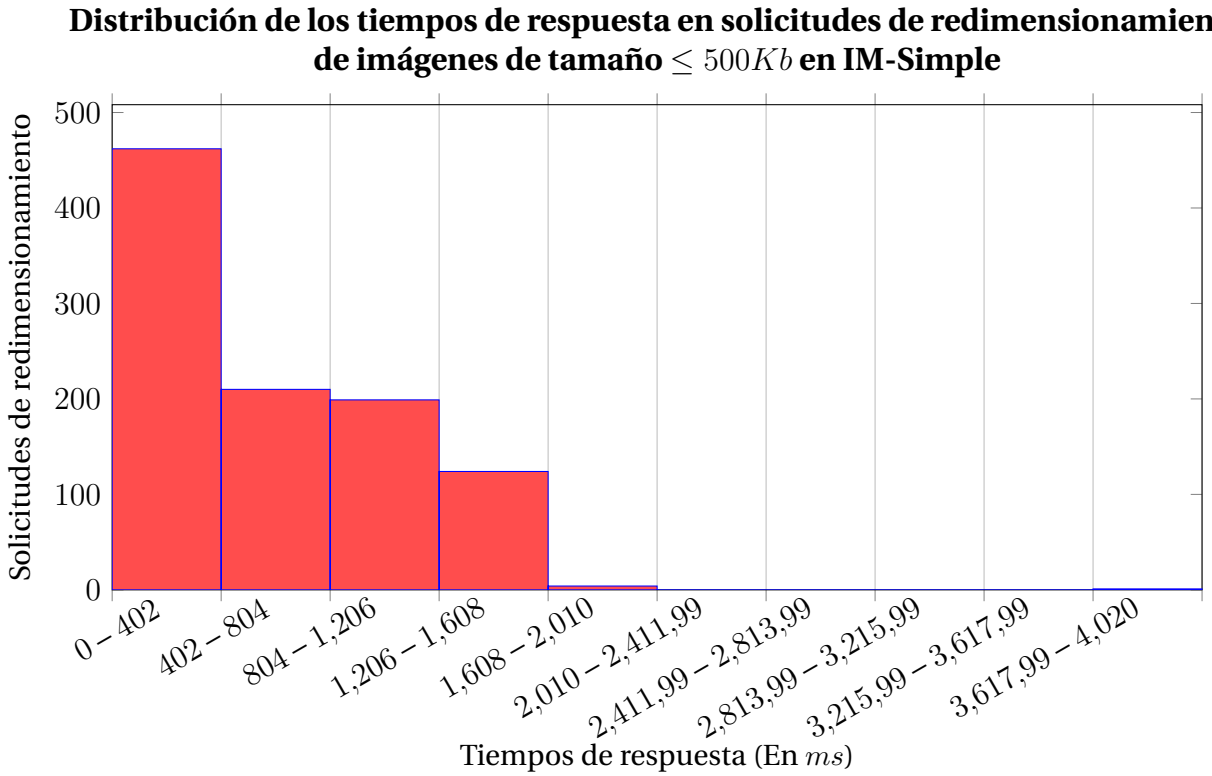

Figura 9.2: Distribución de los tiempos de respuesta en solicitudes de redimensionamiento de imágenes de tamaño $\leq 500Kb$ en IM-Simple

En la figura 9.2 se muestra la distribución de los tiempos de respuesta en las solicitudes de redimensionamiento en las imágenes de tamaño menor a 500Kb. Con excepción de la primera invocación, la cual tuvo una duración de 4 segundos, más del 97,5 % de las invocaciones no superó los 1,6 segundos.

Mediciones para obtención de modelo de rendimiento: 1000 invocaciones de redimensionamiento de imágenes en IM-PK y IM-XRay. Se utilizó el mismo *script* en `Bash` y la misma configuración para generar invocaciones a la función

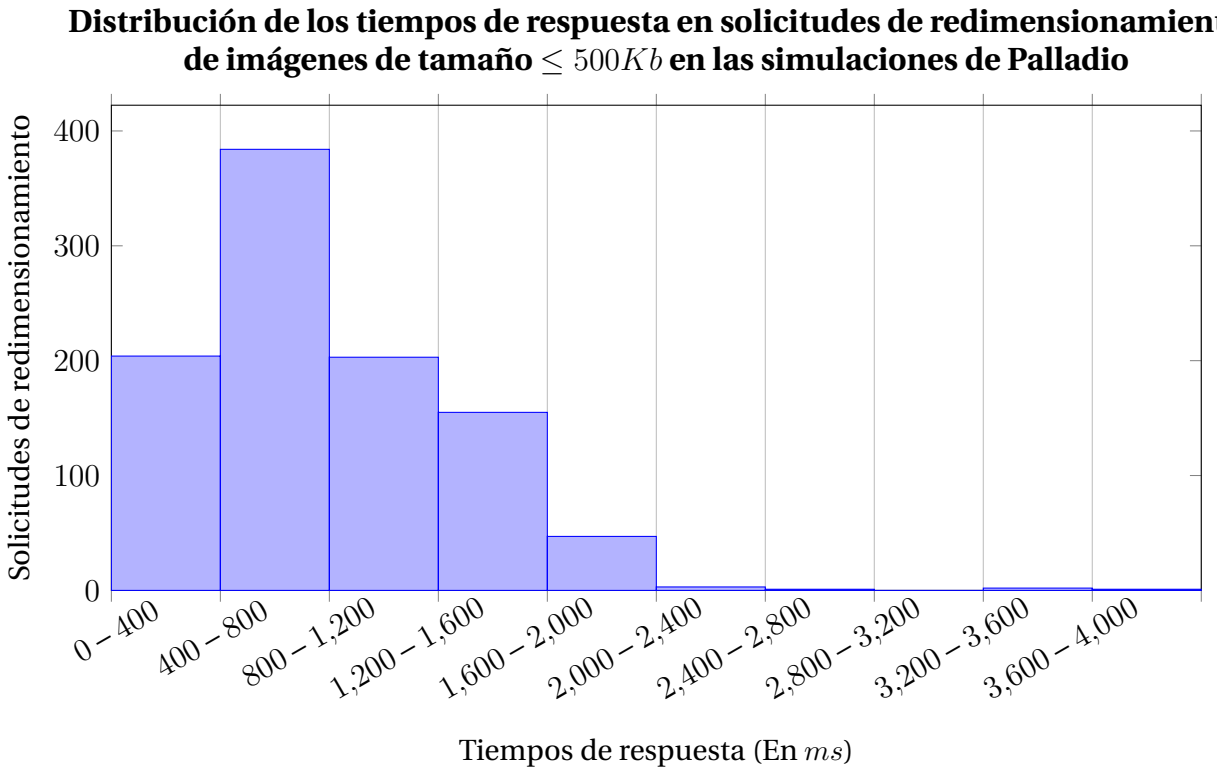

Figura 9.3: Distribución de los tiempos de respuesta en solicitudes de redimensionamiento de imágenes de tamaño $\leq 500Kb$ en las simulaciones de *Palladio Workbench*

Lambda descrita en la sección anterior.

En primera instancia se ejecutaron 1000 invocaciones a IM-PK para generar una bitácora de Kieker y a partir de la misma extraer un modelo de rendimiento PCM usando PMX. Tal y como se señala en la Sección 8.3.1, las estimaciones hechas por PMX sobre el rendimiento de los componentes del modelo se basaban en valores constantes. Fue por esta razón que, para contar con una versión alternativa de *Image Handler* que pudiera brindar otro nivel de detalle en las métricas de rendimiento, se introdujo IM-XRay.

Al igual que en caso anterior, se ejecutaron 1000 invocaciones a IM-XRay, y por medio de un *script* en `Bash`, se obtuvieron las trazas correspondientes a las 1000 invocaciones. Los nuevos datos fueron exportados a formato `.csv` e interpretados con el lenguaje `R`. En `R`, se calcularon distribuciones de frecuencia

Hasta 500Kb			
Solicitud de redimensionamiento	IM-Simple	PCM	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	. –
Coeficiente de variación	0.987	0.683	–

Tabla 9.1: Resumen de datos estadísticos

de la probabilidad en la que un componente lograba procesar una porción de la carga de trabajo total. Estos datos fueron incluidos en los *SEEFs* de cada componente del modelo. Por último se ejecutó una simulación en *Palladio Workbench* con los siguientes parámetros:

- Generación de 1000 mediciones.
- Carga de trabajo: *cerrada*. Se ejecuta una solicitud sobre el modelo hasta que la anterior termina.

En la figura 9.3, se muestra la distribución de los tiempos de respuesta en las solicitudes de redimensionamiento en las simulaciones de *Palladio Workbench* para imágenes de tamaño $\leq 500Kb$. En los resultados de las simulaciones, el 95 % de las invocaciones no superó los 1,6 segundos en procesar la solicitud de redimensionamiento.

En este punto, se cuenta con 1000 mediciones hechas sobre IM-Simple y un modelo al que se le simularon 1000 invocaciones. En la figura 9.4 se comparan los tiempos de respuesta obtenidos en IM-Simple y los de las simulaciones, y, en el Cuadro 9.1, un resumen de los datos estadísticos de los tiempos de respuesta en ambos sujetos de prueba.

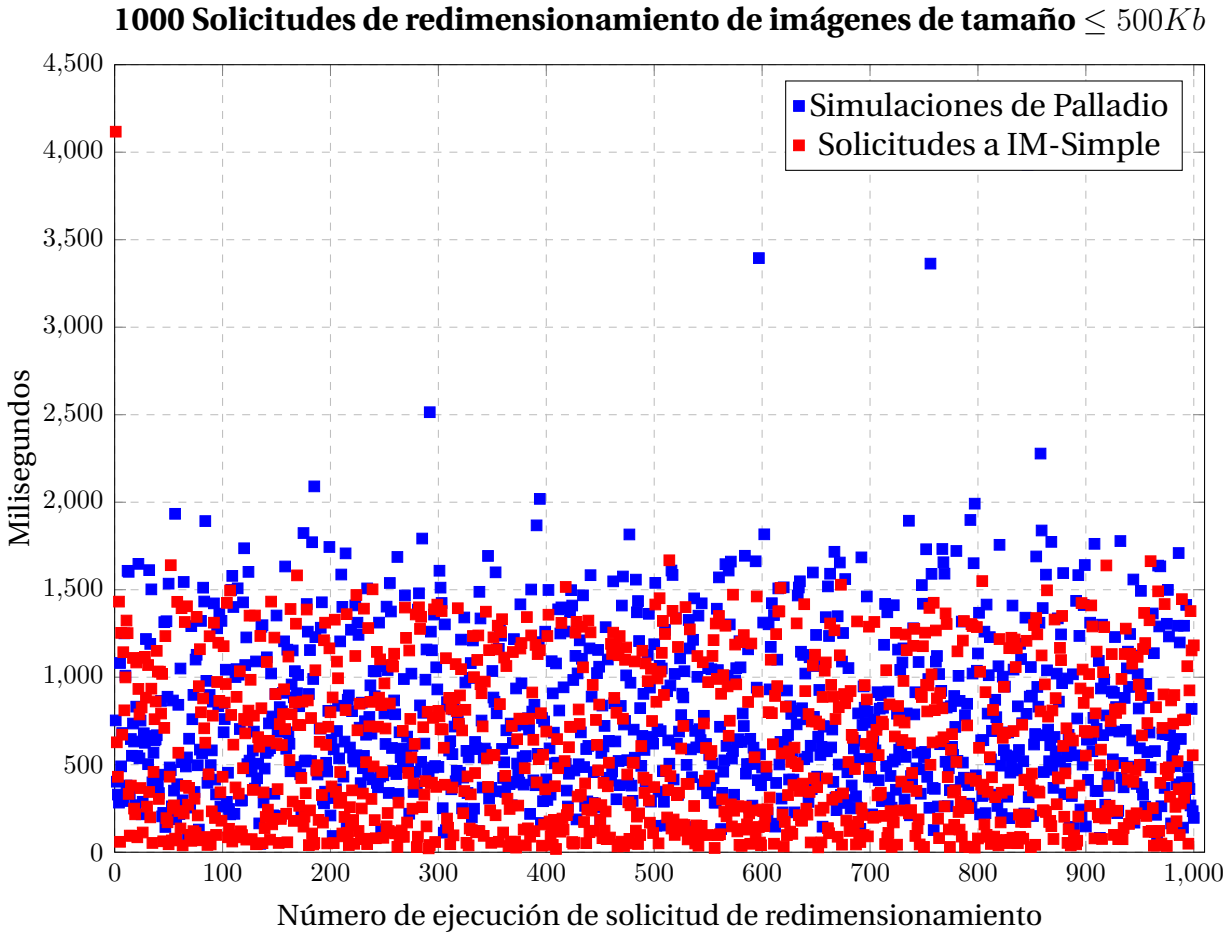

Figura 9.4: IM-Simple *vs* simulaciones en PCM: 1000 solicitudes de redimensionamiento de imágenes de tamaño $\leq 500Kb$.

Análisis de resultados

La Figura 9.4 muestra un panorama alentador. Las ejecuciones de las simulaciones en PCM presentan tiempos de respuesta muy similares a los que entrega IM-Simple. Hay una diferencia de 209,965ms en el tiempo promedio de los tiempos de respuesta de las simulaciones en PCM con respecto a los tiempos de IM-Simple. Preliminarmente, se valora que, debido a que la versión IM-XRay tiene activado el servicio de monitoreo AWS X-Ray y que fue esta versión de *Image Handler* utilizada como referencia para generar los tiempos procesamiento estimados para cada componente del modelo, instrumentalizar la fun-

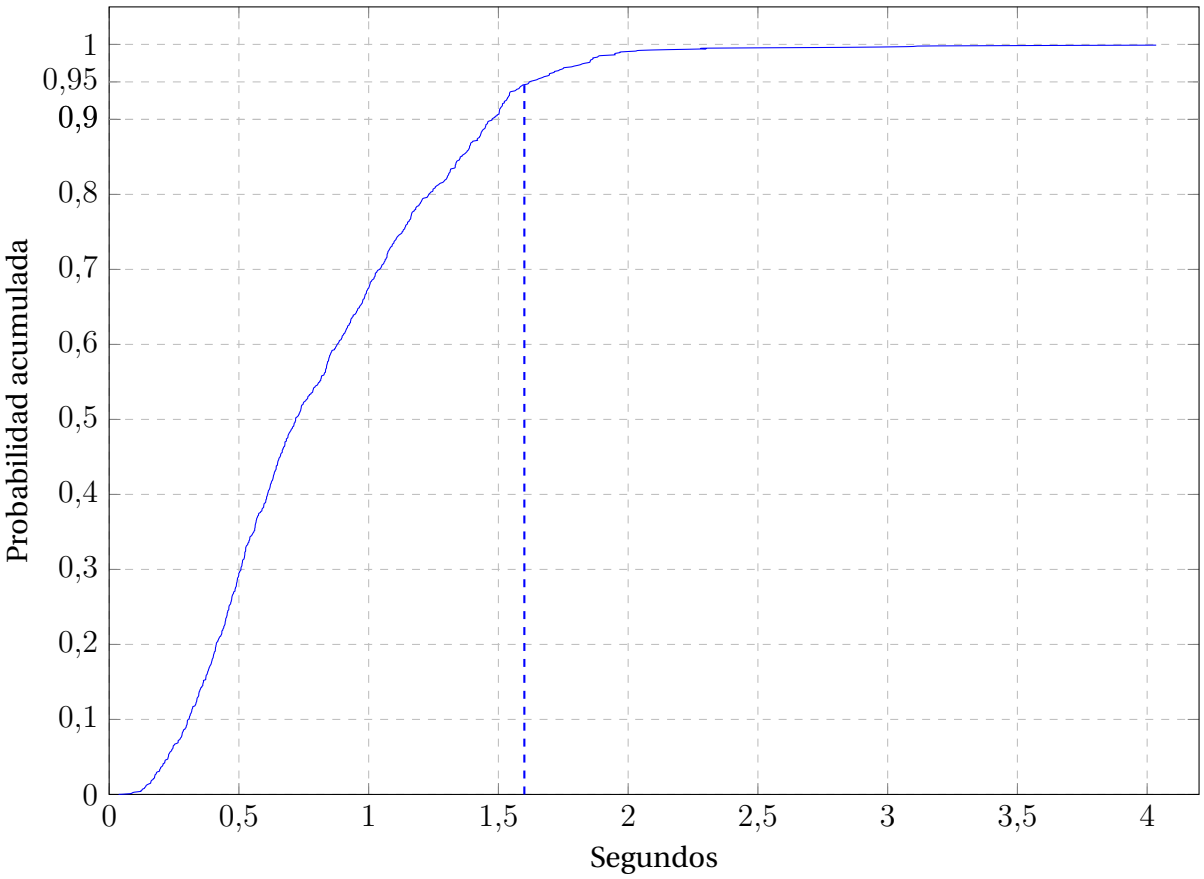

Figura 9.5: Probabilidad acumulada en solicitudes de redimensionamiento en imágenes de tamaño $\leq 500Kb$ en *Palladio Workbench*

ción Lambda con el servicio de monitoreo AWS X-Ray genera un costo adicional (*overhead*) en el procesamiento de la función. Cabe mencionar que la estrategia de monitoreo utilizada fue muy agresiva, pues para obtener nuevas métricas para los componentes del modelo PCM, se habilitaron muchos puntos de monitoreo dentro del código fuente. Además, se configuró la función Lambda para que monitoreara el 100 % de las invocaciones a solicitudes de redimensionamiento. Esta configuración de monitoreo no es la habitual que se utiliza para las funciones Lambda en producción, pero, para el caso de este estudio se necesitó contar con mayores niveles de detalle en las mediciones por lo que fue necesario sacar el mayor provecho al monitoreo. Mientras menor sean utiliza-

das las opciones de monitoreo de AWS X-Ray en la función Lambda, menor será el *overhead* experimentado, tal y como pasa con IM-Simple en donde no se experimenta *overhead* por concepto de monitoreo.

Los resultados muestran desviaciones estándar de 460,569ms y 465,445ms para IM-Simple y las simulaciones de PCM respectivamente. La desviación estándar de las simulaciones de PCM es solamente 4,782ms mayor que la de IM-Simple, lo que sugiere que la agrupación de los datos con respecto a su media aritmética serán muy semejantes.

Los coeficientes de variación para IM-Simple y las simulaciones de PCM fue de 0.987 y 0.683 respectivamente. Estos resultados apuntan a una mayor heterogeneidad entre los tiempos de respuesta y a que el tiempo promedio de procesamiento no se considere representativo para este conjunto de datos. Esta variabilidad viene dada por las diferencias de los tamaños de las imágenes utilizadas para este experimento, como se muestra en la Figura 9.1: imágenes de tamaño $\leq 500Kb$, en donde existen diferencias de hasta 500x, por lo que, por ejemplo, el tiempo de procesamiento de una imagen de tamaño de 5Kb será más rápido que una de tamaño de 490Kb.

Por último, de acuerdo con los resultados de las simulaciones, existe un 95 % de probabilidad de que el tiempo de procesamiento de una solicitud de redimensionamiento de una imagen de tamaño $\leq 500Kb$ tome 1,6 segundos o menos (Figura 9.5). En IM-Simple, se obtuvo un 97,5 % de probabilidad para el mismo caso (Figura 9.2). Para este caso en particular y, debido a la variabilidad de los tiempos de respuesta, se considera que el uso de esta probabilidad acumulada es más representativa a la hora de describir el comportamiento de la función Lambda.

Invocaciones con imágenes mayores a 500Kb y menores o igual a 1Mb

Para la realización de este experimento se contó con la siguiente configuración base:

- *Sujeto de prueba:* La función Lambda IM-Simple.
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño mayor a 500Kb y menor o igual a 1Mb alojadas en Amazon S3.
- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes con dimensiones aleatorias a IM-Simple.
- *Herramientas de medición:* Amazon Cloudwatch.

Configuración para la obtención de datos de rendimiento

- *Sujeto de prueba:* La función IM-XRay
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño mayor a 500Kb y menor o igual a 1Mb alojadas en Amazon S3.
- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes con dimensiones aleatorias a IM-XRay.
- *Herramientas de medición:* Amazon X-Ray.

Para este experimento no se realizó ningún trabajo sobre la versión IM-KP porque no se necesita extraer ningún modelo. El modelo fue extraído y generado en el experimento anterior y debido a que se usa es el mismo código con los mismos puntos de monitoreo, un eventual proceso de extracción daría como

resultado el mismo modelo del experimento anterior. Lo que sí fue necesario hacer, fue calibrar el modelo existente con los resultados de las mediciones a solicitudes de redimensionamiento en imágenes de tamaño mayor a 500Kb y menor o igual 1Mb, utilizando la versión IM-XRay.

Para la realización de este experimento, se obtuvieron 1000 imágenes aleatorias de tamaño mayor a 500Kb y menor o igual a 1Mb del servicio *Lorem Picsum*. Se ejecutó el *script* en `Bash` del experimento anterior para acceder a la API de *Lorem Picsum* para descargar de forma aleatoria 1000 imágenes cuyo tamaño fuera mayor a los 500Kb y menor o igual a 1Mb. La distribución del tamaño de las 1000 imágenes, en Kb, se aprecia en la figura 9.6. El mismo grupo de imágenes se utilizó para realizar solicitudes de redimensionamiento sobre IM-Simple y IM-XRay.

Medición Base: 1000 invocaciones de redimensionamiento de imágenes en IM-Simple. Se ejecutó el *script* en `Bash` del experimento anterior para ejecutar 1000 invocaciones de redimensionamiento en la función IM-Simple en las imágenes de tamaño mayor a 500Kb y menor o igual a 1Mb. El *script* selecciona una imagen de forma aleatoria y luego ejecuta la solicitud de redimensionamiento utilizando dimensiones de ancho y alto de uso común para imágenes en miniatura.

En la figura 9.7 se muestra la distribución de los tiempos de respuesta en las solicitudes de redimensionamiento en las imágenes de tamaño mayor a 500Kb y menor o igual a 1Mb. Más del 98 % de las invocaciones no superó los 8 segundos.

Mediciones para la calibración de modelo de rendimiento: 1000 invocaciones de redimensionamiento de imágenes en IM-XRay. Se utilizó el mismo *script*

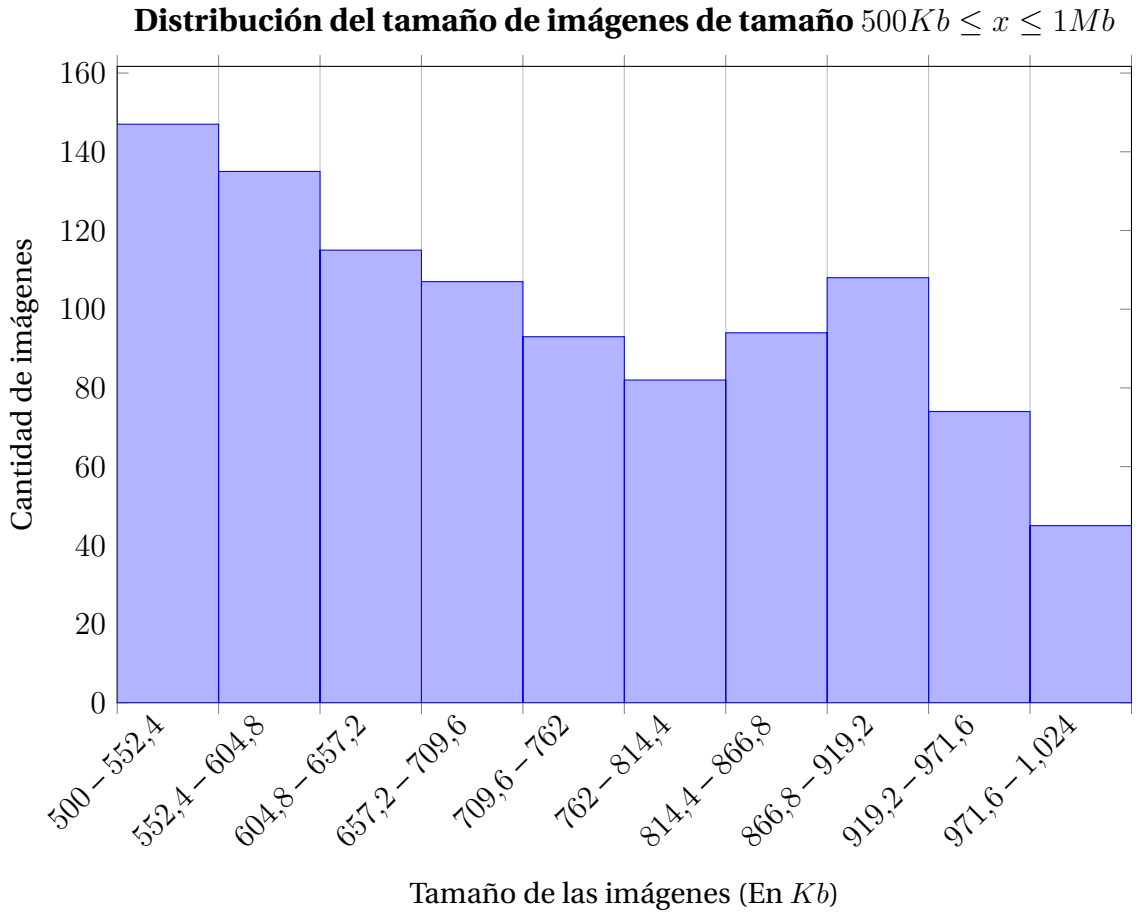

Figura 9.6: Distribución del tamaño de imágenes $500Kb \leq x \leq 1Mb$

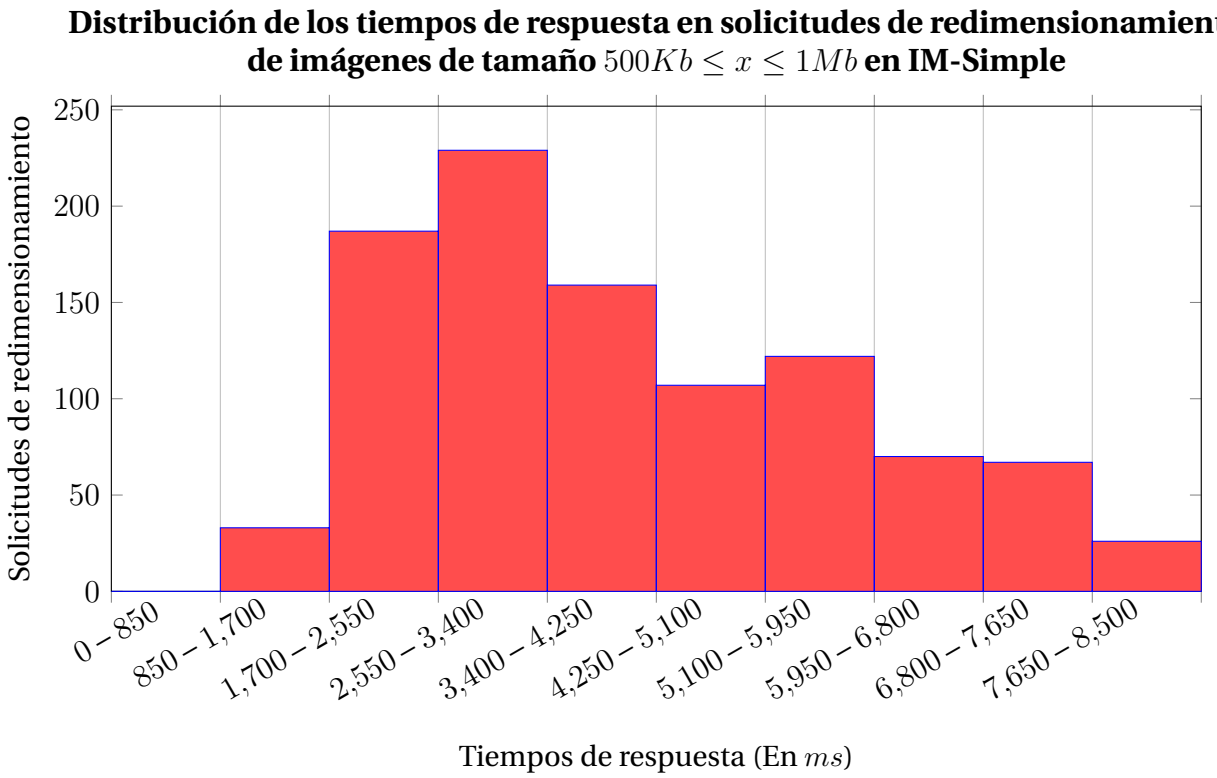

Figura 9.7: Distribución de los tiempos de respuesta en solicitudes de redimensionamiento de imágenes de tamaño $500Kb \leq x \leq 1Mb$ en IM-Simple

en Bash y la misma configuración para generar invocaciones a la función Lambda descrita en la sección anterior.

Se ejecutaron 1000 invocaciones a IM-XRay, y por medio de un *script* en Bash, se obtuvieron las trazas correspondientes a las 1000 invocaciones. Los nuevos datos fueron exportados a formato .csv e interpretados con el lenguaje R. En R, se calcularon distribuciones de frecuencia de la probabilidad en la que un componente lograba procesar una porción de la carga de trabajo total. Estos datos fueron incluidos en los *SEEFs* de cada componente del modelo. Por último se ejecutó una simulación en *Palladio Workbench* con los siguientes parámetros:

- Generación de 1000 mediciones.
- Carga de trabajo: *cerrada*. Se ejecuta una solicitud sobre el modelo hasta que la anterior termina.

En la figura 9.8, se muestra la distribución de los tiempos de respuesta en las solicitudes de redimensionamiento en las simulaciones de *Palladio Workbench* para imágenes de tamaño mayor a 500Kb y menor o igual a 1Mb. En los resultados de las simulaciones, el 95 % de las invocaciones no superó los 8 segundos en procesar la solicitud de redimensionamiento.

En este punto, se cuenta con 1000 mediciones hechas sobre IM-Simple y un modelo al que se le simularon 1000 invocaciones. En la figura 9.9 se comparan los tiempos de respuesta obtenidos en IM-Simple y los de las simulaciones, y, en el Cuadro 9.2, un resumen de los datos estadísticos de los tiempos de respuesta en ambos sujetos de prueba.

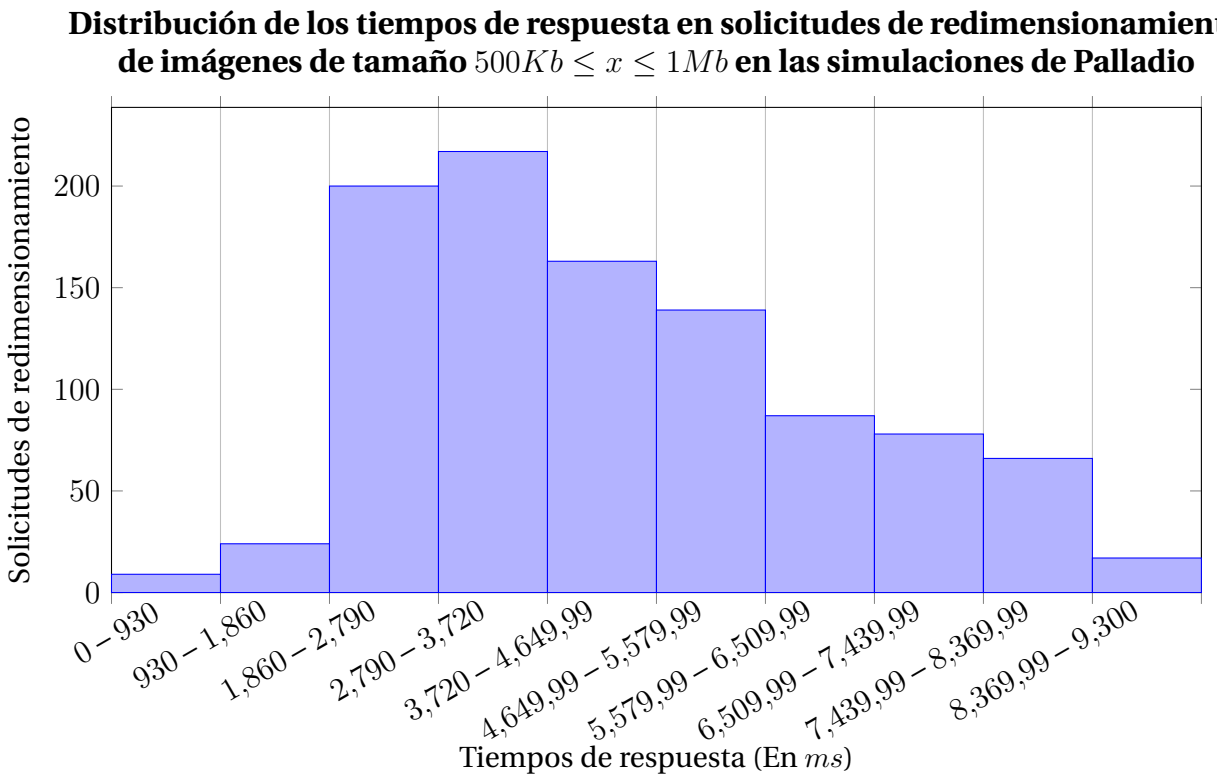

Figura 9.8: Distribución de los tiempos de respuesta en solicitudes de redimensionamiento de imágenes de tamaño $500Kb \leq x \leq 1Mb$ en las simulaciones de Palladio

Entre 500Kb a 1Mb			
Solicitud de redimensionamiento	Image Handler	Palladio	Diferencia
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	–
Coeficiente de variación	0.473	0.462	–

Tabla 9.2: Resumen de datos estadísticos

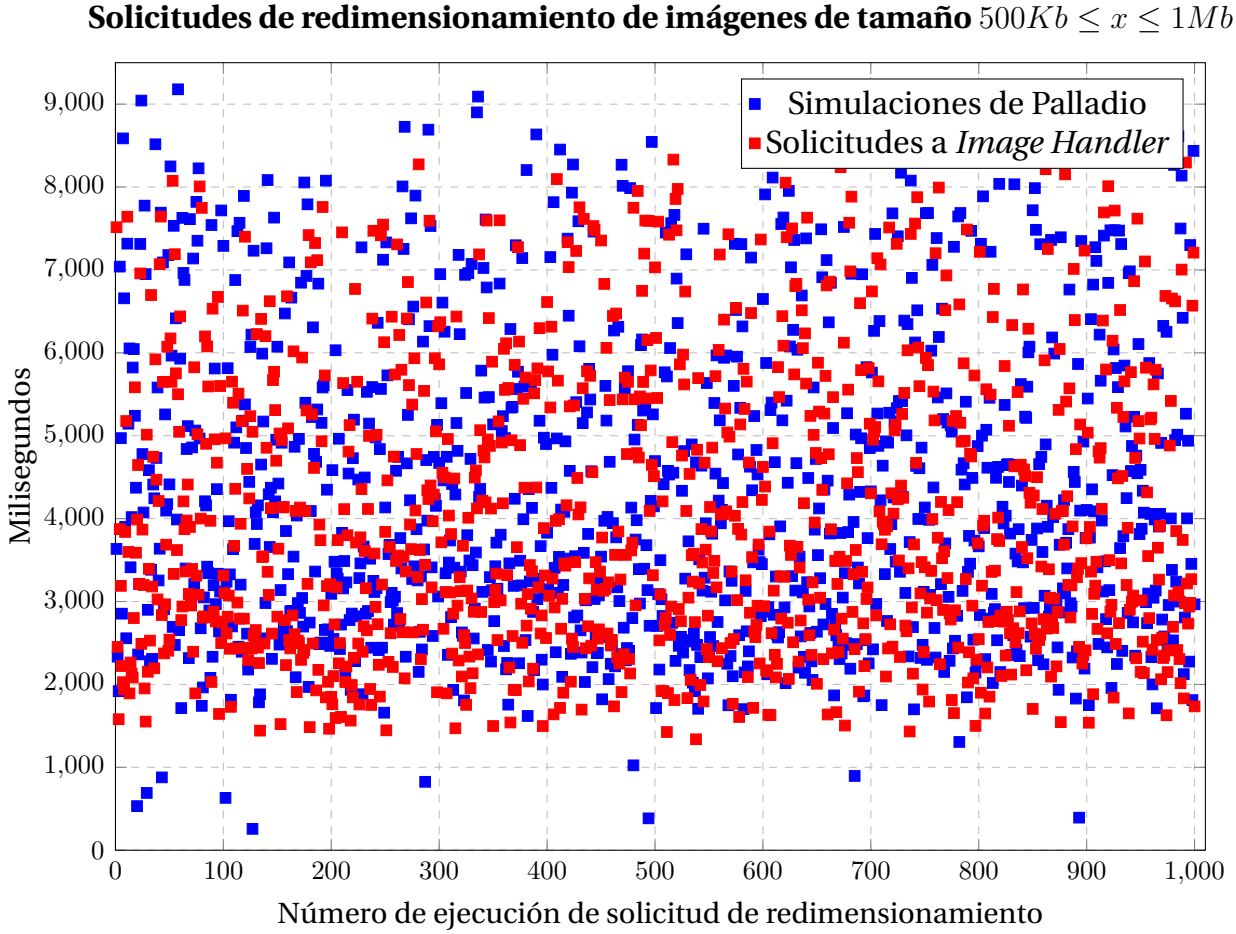

Figura 9.9: Solicitudes de redimensionamiento de imágenes de tamaño $500Kb \leq x \leq 1Mb$

Análisis de resultados

A primera vista, la comparación de los tiempos de respuesta de IM-Simple y las simulaciones en PCM mostradas en la Figura 9.9 muestran gran similitud. Hay una diferencia de 274,428ms en el tiempo promedio de los tiempos de respuesta en las simulaciones PCM con respecto a los tiempos de IM-Simple. En este experimento también se intuye que la estrategia de monitoreo de AWS X-Ray es la responsable de agregar este *overhead* en el procesamiento.

Los resultados muestran desviaciones estándar de 1731.974ms y 1844.893ms para IM-Simple y las simulaciones de PCM respectivamente. La desviación es-

tándar de las simulaciones de PCM es 112.919ms mayor que la de IM-Simple. Esto indica que la agrupación de los datos con respecto de su media aritmética es muy similar entre ambas muestras.

Los coeficientes de variación de ambas muestras bajaron con respecto a los obtenidos en el primer experimento, aún así, los valores de 0,473 para IM-Simple y de 0,462 para las simulaciones de PCM se consideran heterogéneos, no muy similares entre sí. Esto hace que el tiempo promedio de una solicitud pueda considerarse no tan representativo a la hora de caracterizar el comportamiento de la función Lambda bajo esta carga de trabajo. La variabilidad en los tiempos de respuesta bajó debido a que en los tamaños de las imágenes utilizadas (Figura 9.6) tienen una diferencia de hasta 2 veces: es decir, en este conjunto, la imagen más grande es el doble de grande que la imagen más pequeña.

Por último, de acuerdo con los resultados de las simulaciones, existe un 95 % de probabilidad de que el tiempo de procesamiento de una solicitud de redimensionamiento de una imagen de tamaño $500Kb \leq x \leq 1Mb$ tome 8 segundos o menos (Figura 9.10). En IM-Simple, se obtuvo un 98 % de probabilidad para el mismo caso (Figura 9.7). Para este caso, como en el experimento anterior, se considera más apropiado el uso de la probabilidad acumulada para describir el comportamiento de la función Lambda.

Invocaciones con imágenes mayores a 1Mb y menores o igual a 2Mb

Para la realización de este experimento se contó con la siguiente configuración base:

- *Sujeto de prueba:* La función Lambda IM-Simple.

Probabilidad acumulada: solicitudes de redimensionamiento en imágenes $500Kb \leq x \leq 1Mb$ en PCM

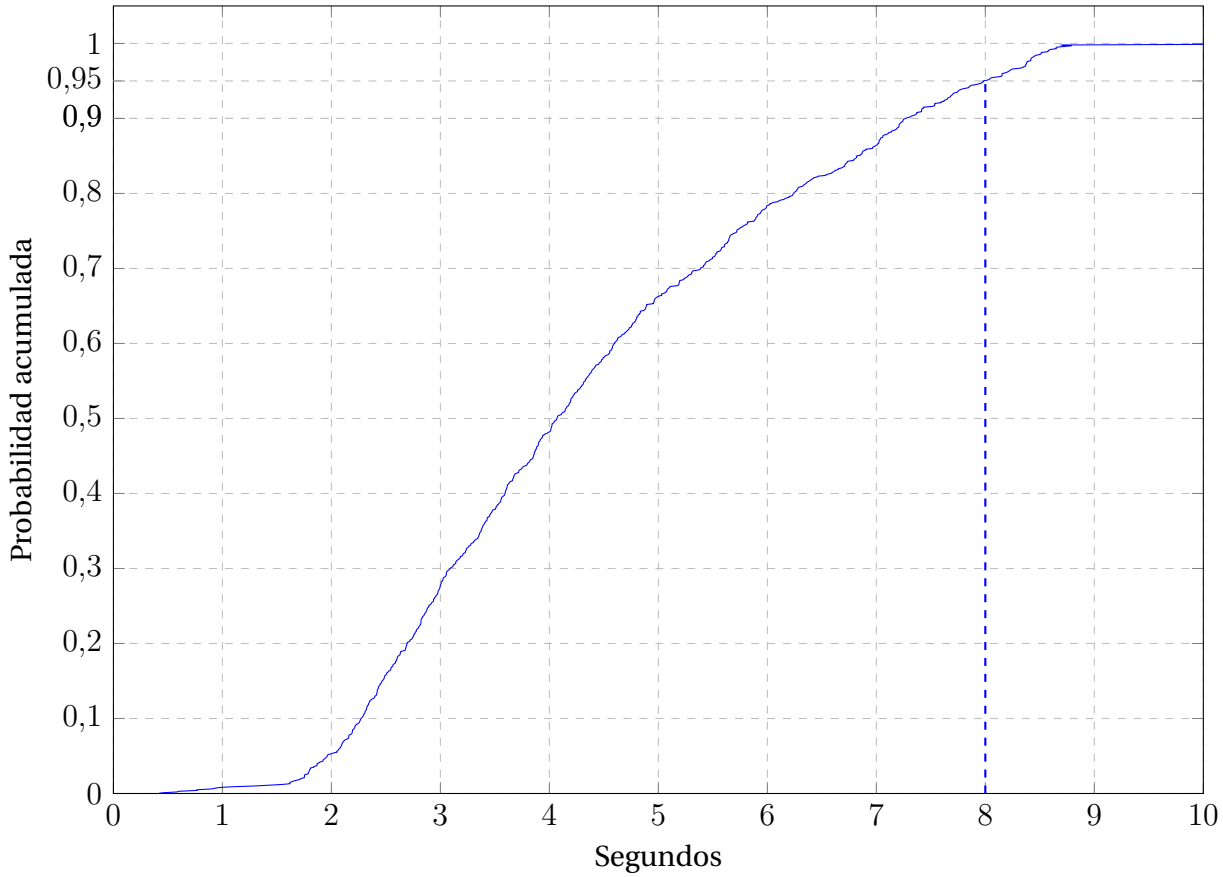

Figura 9.10: Probabilidad acumulada de solicitudes de redimensionamiento en imágenes $500Kb \leq x \leq 1Mb$ en PCM

- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño mayor a 1Mb y menor o igual a 2Mb alojadas en Amazon S3.
- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes con dimensiones aleatorias a IM-Simple.
- *Herramientas de medición:* Amazon Cloudwatch.

Configuración para la obtención de datos de rendimiento

- *Sujeto de prueba:* La función IM-XRay
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño mayor a 1Mb y menor o igual a 2mb alojadas en Amazon S3.
- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes con dimensiones aleatorias a IM-XRay.
- *Herramientas de medición:* Amazon X-Ray.

Al igual que en el experimento de la sección anterior, no se realizó ningún trabajo sobre la versión IM-KP debido a que no era necesario la extracción de un nuevo modelo, sino de, calibrar el modelo existente con los resultados de las mediciones para generar nuevas simulaciones.

Para la realización de este experimento, se obtuvieron 1000 imágenes aleatorias de tamaño mayor a 1Mb y menor o igual a 2Mb del servicio *Lorem Picsum*. Se ejecutó el *script* en `Bash` del experimento anterior para acceder a la API de *Lorem Picsum* para descargar de forma aleatoria 1000 imágenes cuyo tamaño fuera mayor a 1Mb y menor o igual a 2Mb. La distribución del tamaño de las

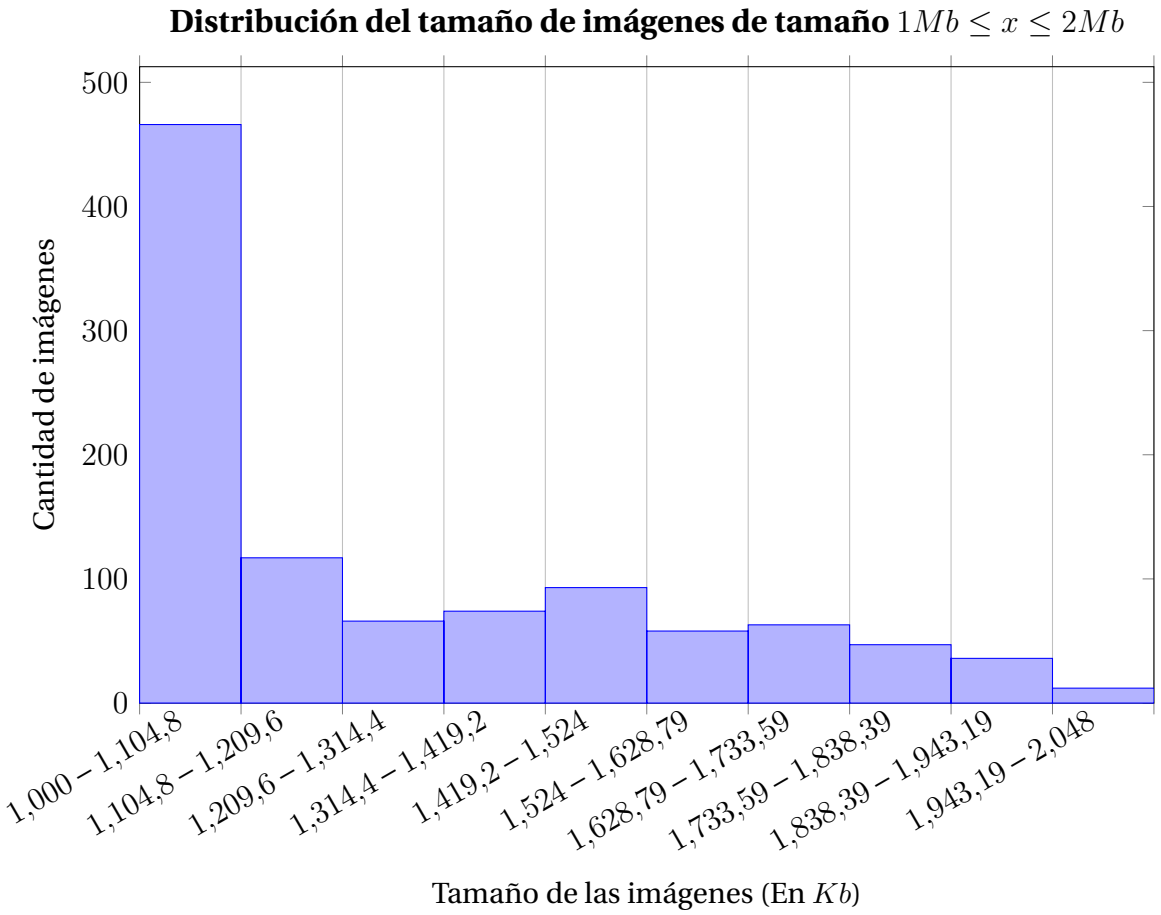

Figura 9.11: Distribución del tamaño de imágenes de tamaño $1Mb \leq x \leq 2Mb$

1000 imágenes, en Kb, se aprecia en la figura 9.11. El mismo grupo de imágenes se utilizó para realizar solicitudes de redimensionamiento sobre IM-Simple y IM-XRay.

Medición Base: 1000 invocaciones de redimensionamiento de imágenes en IM-Simple. Se ejecutó el *script* en `Bash` del experimento anterior para ejecutar 1000 invocaciones de redimensionamiento en la función IM-Simple en las imágenes de tamaño mayor a 1Mb y menor o igual a 2Mb. El *script* selecciona una imagen de forma aleatoria y luego ejecuta la solicitud de redimensionamiento utilizando dimensiones de ancho y alto de uso común para imágenes en miniatura.

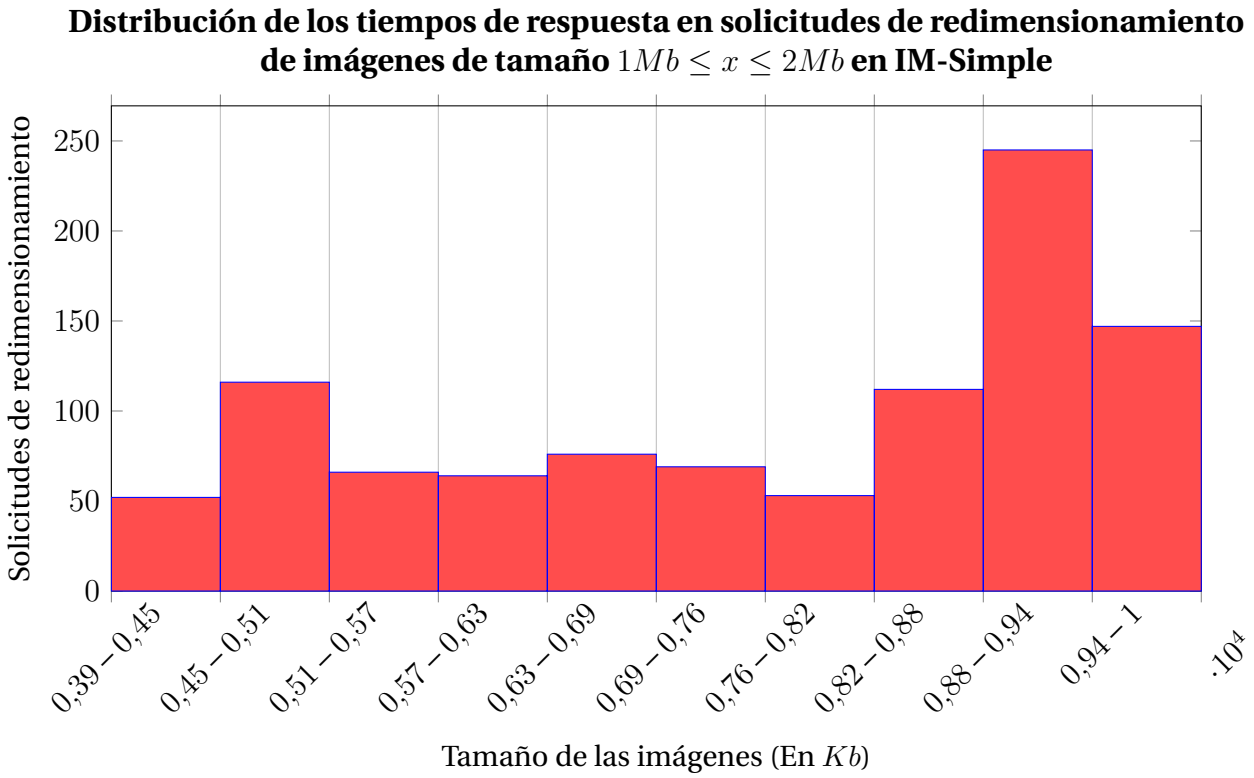

Figura 9.12: Distribución de los tiempos de respuesta en solicitudes de redimensionamiento de imágenes de tamaño $1Mb \leq x \leq 2Mb$ en IM-Simple

En la figura 9.12 se muestra la distribución de los tiempos de respuesta en las solicitudes de redimensionamiento en las imágenes de tamaño mayor a 500Kb y menor o igual a 1Mb. Más del 92,5 % de las invocaciones no superó los 9,7 segundos.

Mediciones para la calibración de modelo de rendimiento: 1000 invocaciones de redimensionamiento de imágenes en IM-XRay. Se utilizó el mismo *script* en `Bash` y la misma configuración para generar invocaciones a la función Lambda descrita en la sección anterior.

Se ejecutaron 1000 invocaciones a IM-XRay, y por medio de un *script* en `Bash`, se obtuvieron las trazas correspondientes a las 1000 invocaciones. Los nuevos datos fueron exportados a formato `.csv` e interpretados con el lenguaje `R`. En `R`, se calcularon distribuciones de frecuencia de la probabilidad en la que un componente lograba procesar una porción de la carga de trabajo total. Estos datos fueron incluidos en los *SEEFs* de cada componente del modelo. Por último se ejecutó una simulación en *Palladio Workbench* con los siguientes parámetros:

- Generación de 1000 mediciones.
- Carga de trabajo: *cerrada*. Se ejecuta una solicitud sobre el modelo hasta que la anterior termina.

En la figura 9.13, se muestra la distribución de los tiempos de respuesta en las solicitudes de redimensionamiento en las simulaciones de *Palladio Workbench* para imágenes de tamaño mayor a 500Kb y menor o igual a 1Mb. En los resultados de las simulaciones, el 95 % de las invocaciones no superó los 8 segundos en procesar la solicitud de redimensionamiento.

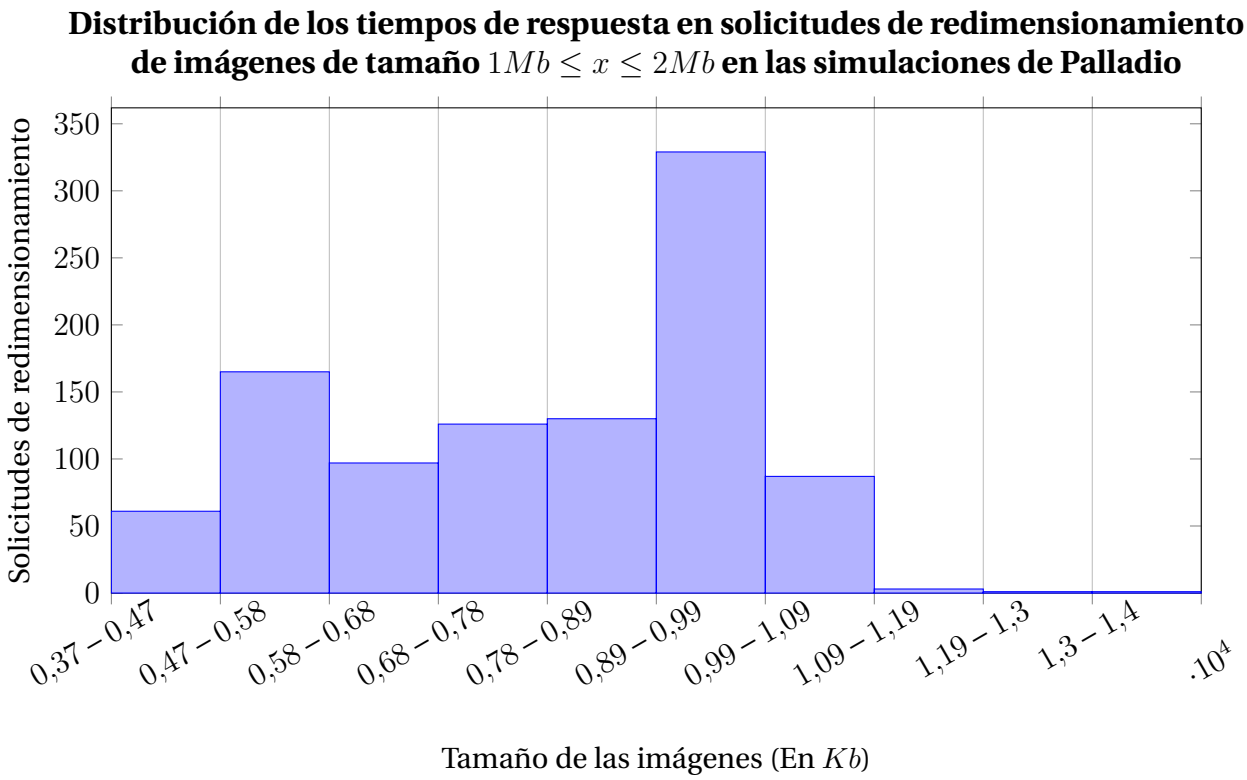

Figura 9.13: Distribución de los tiempos de respuesta en solicitudes de redimensionamiento de imágenes de tamaño $1Mb \leq x \leq 2Mb$ en las simulaciones de Palladio

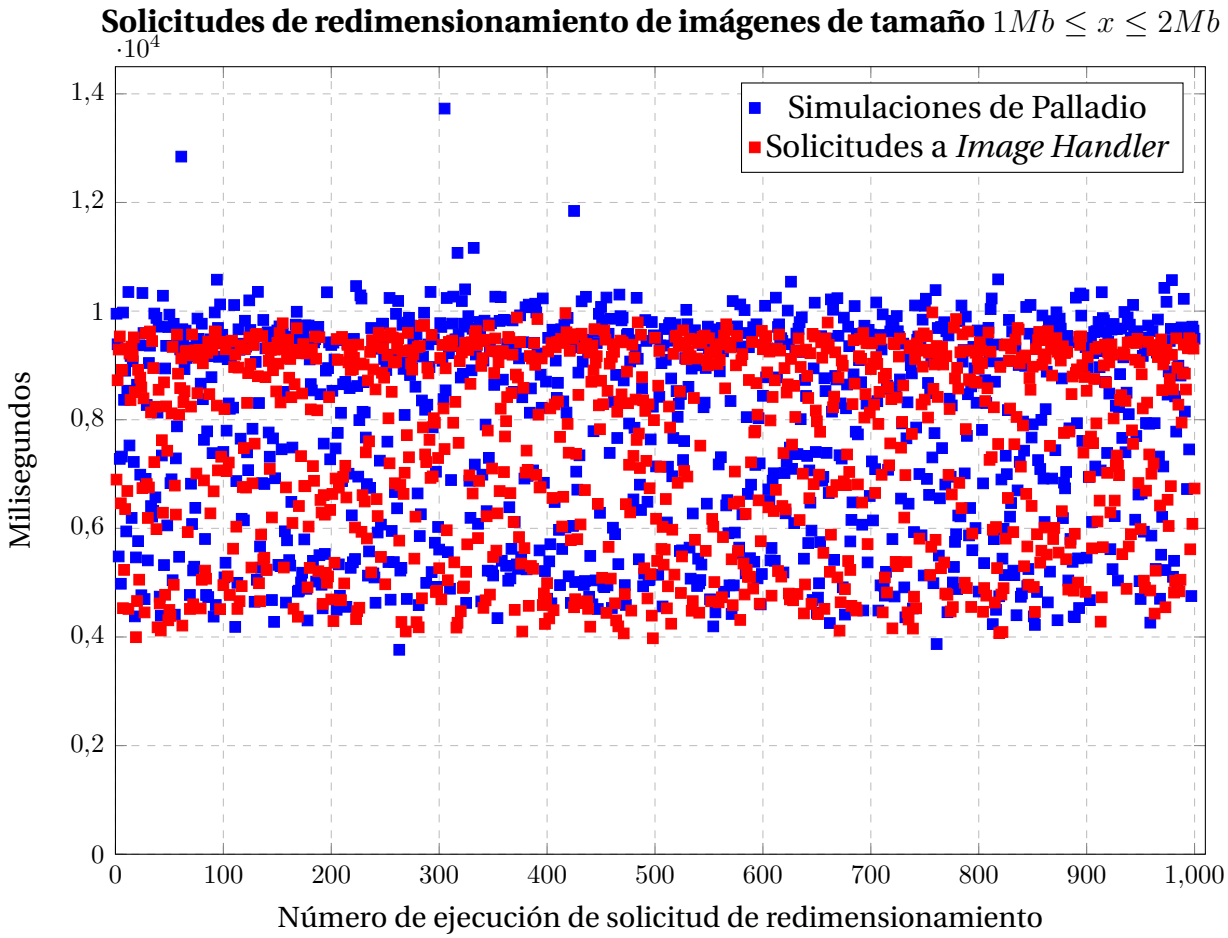

Figura 9.14: Solicitudes de redimensionamiento de imágenes de tamaño $1Mb \leq x \leq 2Mb$

En este punto, se cuenta con 1000 mediciones hechas sobre IM-Simple y un modelo al que se le simularon 1000 invocaciones. En la figura 9.14 se comparan los tiempos de respuesta obtenidos en IM-Simple y los de las simulaciones, y, en el Cuadro 9.3, un resumen de los datos estadísticos de los tiempos de respuesta en ambos sujetos de prueba.

Análisis de resultados

Bajo esta carga de trabajo es en donde se pueden apreciar mayor similitud en los tiempos de procesamiento de las solicitudes de redimensionamiento

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	–
Coeficiente de variación	0.221	0.230	–

Tabla 9.3: Resumen de datos estadísticos

entregados por IM-Simple y las simulaciones de PCM. Hay una diferencia de 257,773ms en el tiempo promedio de los tiempos de respuesta en las simulaciones PCM con respecto a los tiempos de IM-Simple. Al igual que en los experimientos anteriores, se considera que la influencia del servicio de monitoreo de AWS X-Ray es el responsable de causar la diferencia en el tiempo de procesamiento.

Los resultados muestran desviaciones estándar de 1816,152ms y 1914.258ms para IM-Simple y las simulaciones de PCM respectivamente. La desviación estándar de las simulaciones de PCM es 98.106ms mayor que la de IM-Simple. Esto indica que la agrupación de los datos con respecto a su media aritmética es muy similar entre ambas muestras.

Los coeficientes de variación de son de 0,221 para IM-Simple y de 0,230 para las simulaciones de PCM. Estos resultados son los menores de los tres experimentos y sugieren que los datos son homogéneos entre sí y que, para esta muestra se pueda considerar representativo utilizar el tiempo promedio de respuesta para caracterizar el comportamiento de la función Lambda.

Por último, de acuerdo con los resultados de las simulaciones, existe un 95 % de probabilidad de que el tiempo de procesamiento de una solicitud de redimensionamiento de una imagen de tamaño $1Mb \leq x \leq 2Mb$ tome 10,14 segundos o menos (Figura 9.15). En IM-Simple, al 100 % de las solicitudes le tomó

Probabilidad acumulada: solicitudes de redimensionamiento en imágenes $1Mb \leq x \leq 2Mb$ en PCM

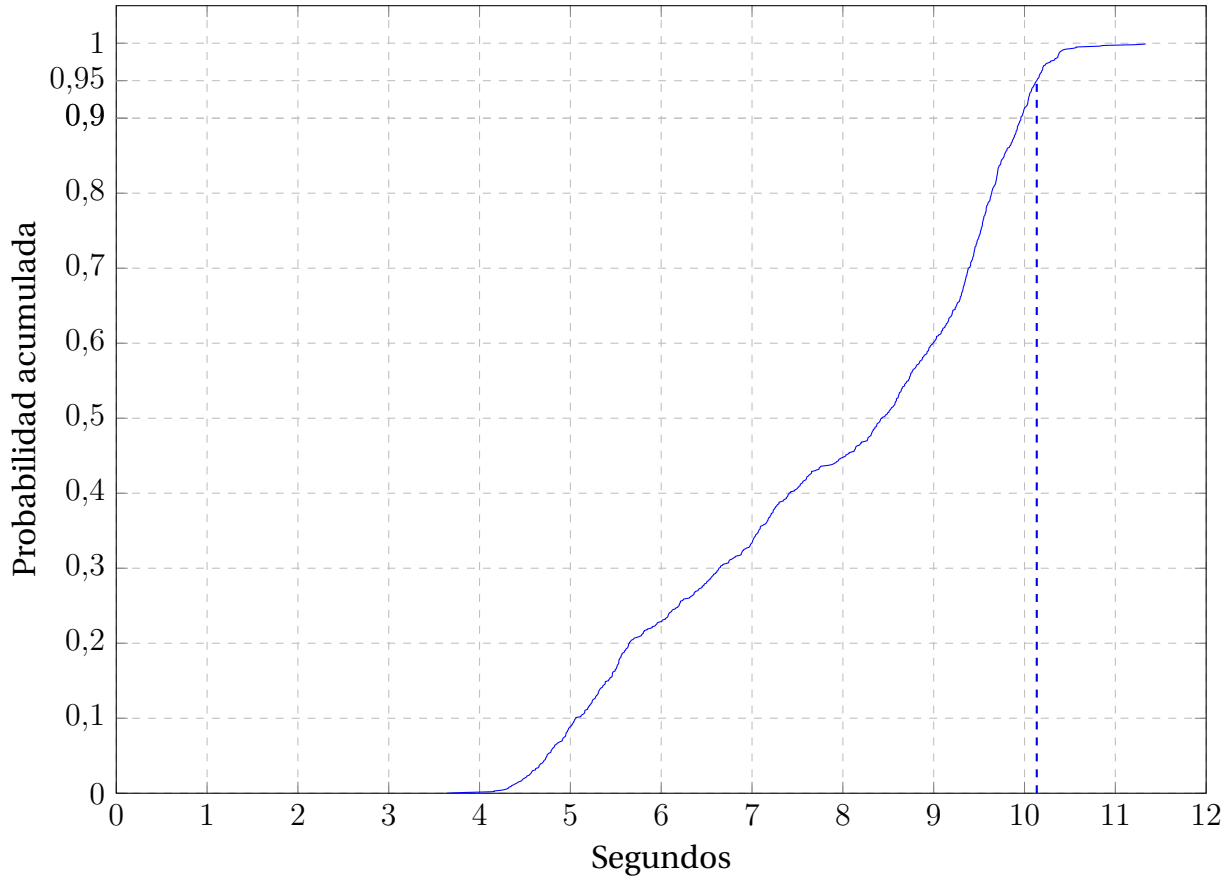

Figura 9.15: Probabilidad acumulada de solicitudes de redimensionamiento en imágenes $1Mb \leq x \leq 2Mb$ en PCM

10 segundos o menos (Figura 9.12). Para este caso, el uso del tiempo promedio de procesamiento de solicitud de redimensionamiento podría ser utilizado para caracterizar el comporamiento de la función Lambda, pero, como en los dos casos anteriores, se considera que, el uso de la probabilidad acumulada sea más adecuado para brindar una predicciones más acertada.

Resultados Generales

En los tres experimentos propuestos, los resultados provenientes de las simulaciones lograron caracterizar en gran medida lo observado en IM-Simple.

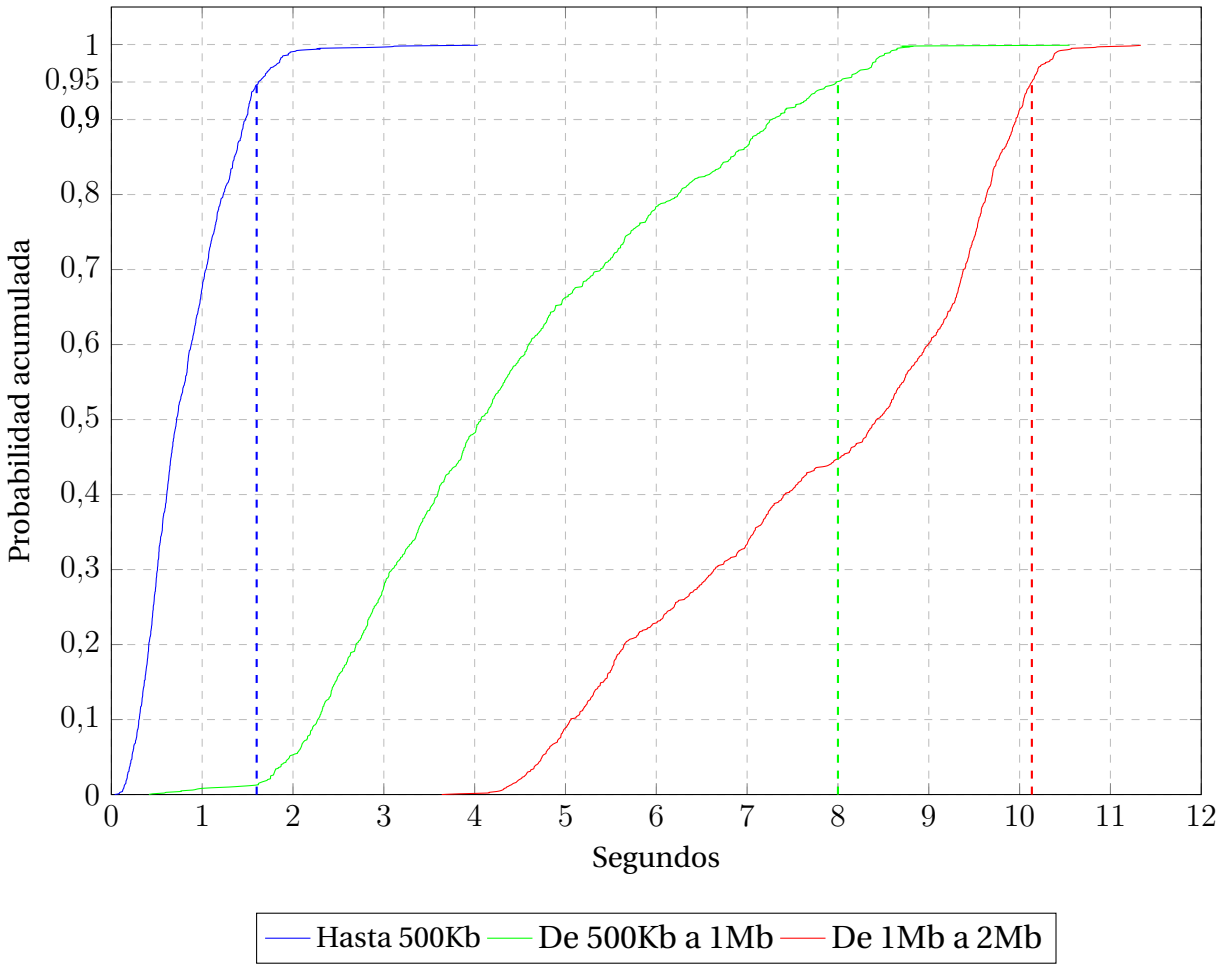

Figura 9.16: 3000 Simulaciones: Función de probabilidad acumulada para los tres escenarios de pruebas

En 9.16 se muestran las 3000 simulaciones realizadas (1000 por cada caso). Las líneas punteadas verticales señalan en dónde se concentran el 95 % de los datos de las simulaciones: 1,6 segundos para las imágnes menores a 500Kb, 8 segundos para las imágenes de entre 500Kb a 1Mb y 10,14 segundos para las imágenes entre 1Mb y 2Mb.

En el Cuadro 9.4 se muestra un resumen estadístico de los tres experimentos llevados a cabo. En este resumen se evidencia los efectos del monitoreo de Amazon X-Ray en la versión IM-XRay de *Image Handler* en el tiempo prome-

dio de procesamiento de una solicitud de redimensionamiento. Las diferencias entre el tiempo promedio de procesamiento en IM-Simple y IM-XRay fue de 209.965ms, 274.428ms y 257.773 para el experimento #1, #2 y #3 respectivamente. En promedio 247.389ms de diferencia. La desviación estándar de estos tres valores es de 33.463ms y el coeficiente de variación es de 0,13. Estos datos son muy similares entre sí, y esto hace que para el caso de las simulaciones en PCM se pueda argumentar que en promedio se puede esperar diferencia de 247.389ms en el tiempo de respuesta de las simulaciones *versus* los reportados por IM-Simple.

Cabe destacar que las diferencias observadas entre los tiempos de respuesta de las simulaciones con IM-Simple fueron hechas con base en tres valores, los cuales fueron obtenidos a partir de 3000 mediciones bajo tres grupos distintos de imágenes. Por ello los consideramos representativos.

Desde el punto de vista de la arquitectura de software, *Image Handler* muestra mejores rendimientos cuando se prueba con imágenes iguales o menores a 500Kb que con aquellas mayores a ese tamaño. Dentro de las mediciones a los componentes de *Image Handler*, el que experimentó una mayor variablidad en los tiempos de procesamiento fue el componente de redimensionamiento. Aunque es el componente más importante dentro de esta arquitectura, es también el cuello de botella. Este es el principal componente a ser modificado en aras de probar diferentes comportamientos de *Image Handler*.

Varias otras optimizaciones podrían ser tomadas en cuenta aparte de valorar cambios en la biblioteca de redimensionamiento. Aunque puedan existir bibliotecas que logren disminuir en gran medida el trabajo de redimensionamiento de una imagen, se hace útil también el considerar otras herramientas o estrategias para mejorar los tiempos de respuesta de la función, por ejemplo:

1. Limitar el tamaño de las imágenes a procesar: una vez conocido el rendimiento de la función, se podría en limitar las imágenes a procesar basado en su tamaño, por ejemplo, solamente imágenes menores a 500Kb.

2. Utilizar una memoria intermedia de acceso rápido, un *caché*, para guardar copias de imágenes redimensionadas: después de una primera solicitud de redimensionamiento, se guarda la imagen resultante en un *caché* y luego, entregar esta imagen en subsecuentes solicitudes de redimensionamiento. De esta forma las solicitudes repetitivas no llegarían a ser procesadas por la función Lambda. Esta estrategia se muestra en la Figura 8.1.

3. Preprocesar imágenes de gran tamaño fuera de línea: en el caso en donde se cuente con las imágenes originales de antemano, podría ser muy útil ejecutar procesos fuera de línea para redimensionar estas imágenes a los tamaños requeridos y hacer que *Image Handler* solamente sirva estas imágenes y no aplicar ningún tipo de lógica de redimensionamiento.

Hasta 500Kb			
Solicitud de redimensionamiento	IM-Simple	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	. –
Coeficiente 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	–
Coeficiente 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	–
Coeficiente de variación	0.221	0.230	–

Tabla 9.4: Resumen de datos estadísticos de los tres experimentos propuestos

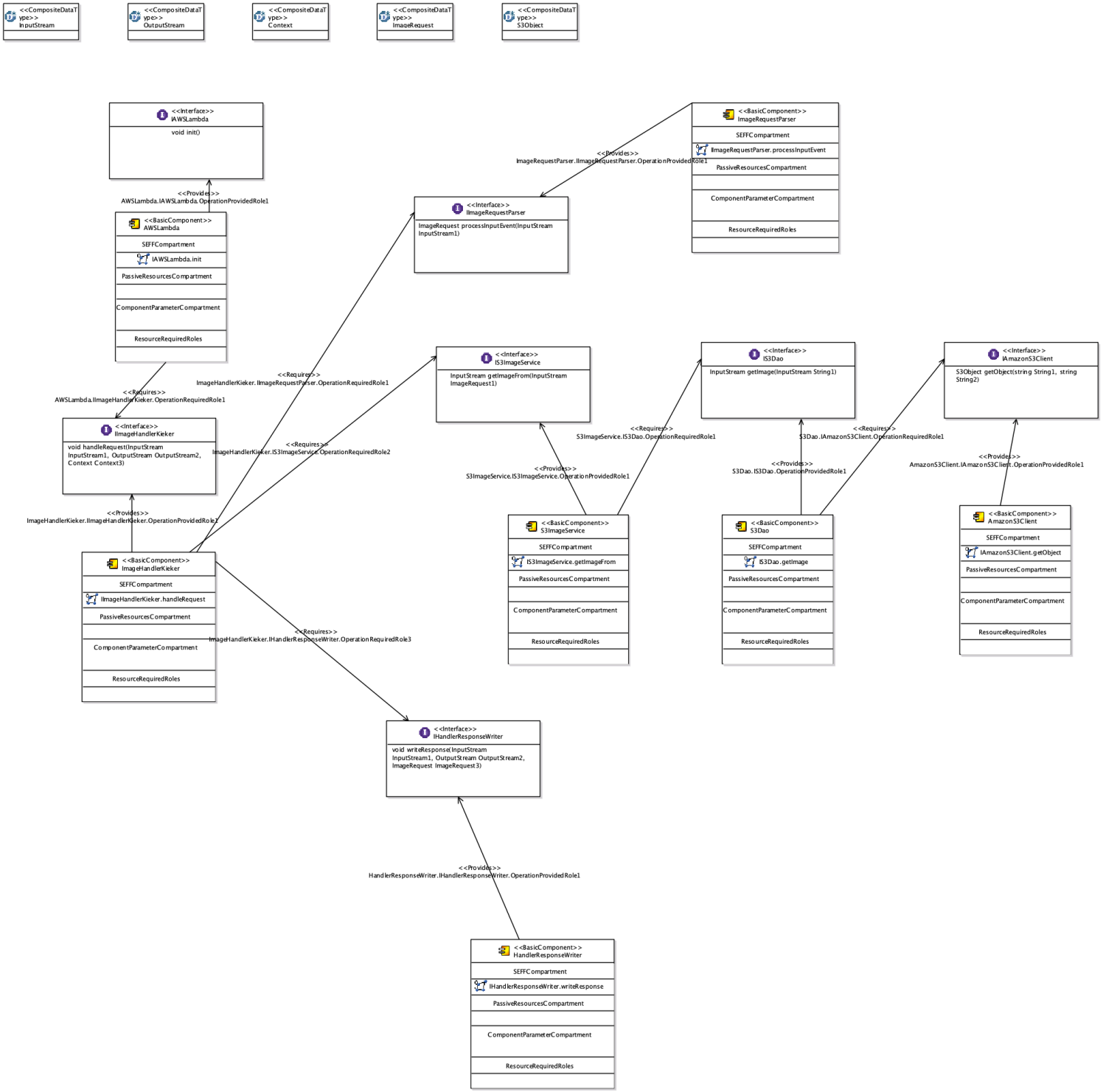

Figura 9.17: *Image-Handler* publicando eventos de rendimiento al servicio AWS X-Ray

9.0.2. Ejecución Secuencial Ininterrumpida de solicitudes de redimensionamiento

En el experimento de la Sección 9.0.1 se realizaron múltiples solicitudes de redimensionamiento con el fin de extraer y crear un modelo de rendimiento, ejecutar simulaciones sobre este modelo y comparar los resultados con lo observado en una versión de *Image Handler* real.

Una de las principales inquietudes durante la ejecución de funciones en la nube es determinar si la velocidad en la que se procesa una solicitud resulta ser baja y muestra una tendencia predecible. Los desarrolladores e implementadores necesitan evaluar esto para afinar el código fuente y la arquitectura para evitar cobros elevados por el uso de la plataforma. De acuerdo con [25], una función puede pasar por dos grandes estados: uno “fría” y una “caliente”. En el estado frío, la plataforma que soporta la función en la nube debe aprovisionar los recursos necesarios para ejecutar la función. Durante esta fase se pueden experimentar los tiempos de respuesta más prolongados. La fase caliente se da luego de que la plataforma que soporta la función reconoce que la función está siendo invocada constantemente y que, debido a este uso, necesita proporcionarle mayores recursos computacionales para brindar mejores tiempos de respuesta. Se espera que, en una función que se esté invocando constantemente, solamente un porcentaje muy bajo del tiempo se encuentre en estado frío y la mayor parte (mientras continúe siendo invocada) estará en estado caliente.

Ejecución de solicitudes de redimensionamiento simultáneas para imágenes de tamaño menor a 500Kb

Para la realización de este experimento se contó con la siguiente configuración:

- *Sujeto de prueba:* La función Lambda IM-Simple
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño menor o igual a 500Kb alojadas en Amazon S3.
- *Carga de trabajo:*
    1. 1000 invocaciones secuenciales de redimensionamiento de imágenes con dimensiones aleatorias a IM-Simple.
    2. 1000 invocaciones secuenciales de redimensionamient de una sola imagen a IM-Simple.
- *Herramientas de medición:* Amazon Cloudwatch.

En la carga de trabajo #1, utilizando imágenes aleatorias, la ejecución de las 1000 solicitudes de redimensionamiento fueron las mismas utilizadas para el experimento de la Sección 9.0.1.

La ejecución de las 1000 solicitudes de redimensionamiento tomó 2 horas y 36 minutos. Para la carga de trabajo #2, utilizando una sola imagen, las 1000 solicitudes de redimensionamiento tomaron 25 minutos. Se modificó el *script* en `Bash` del experimento de la Sección 9.0.1 para elegir aleatoriamente una imagen del *cluster* de 1000 y generar 1000 solicitudes secuenciales a partir de esta con dimensiones de ancho y alto estáticas.

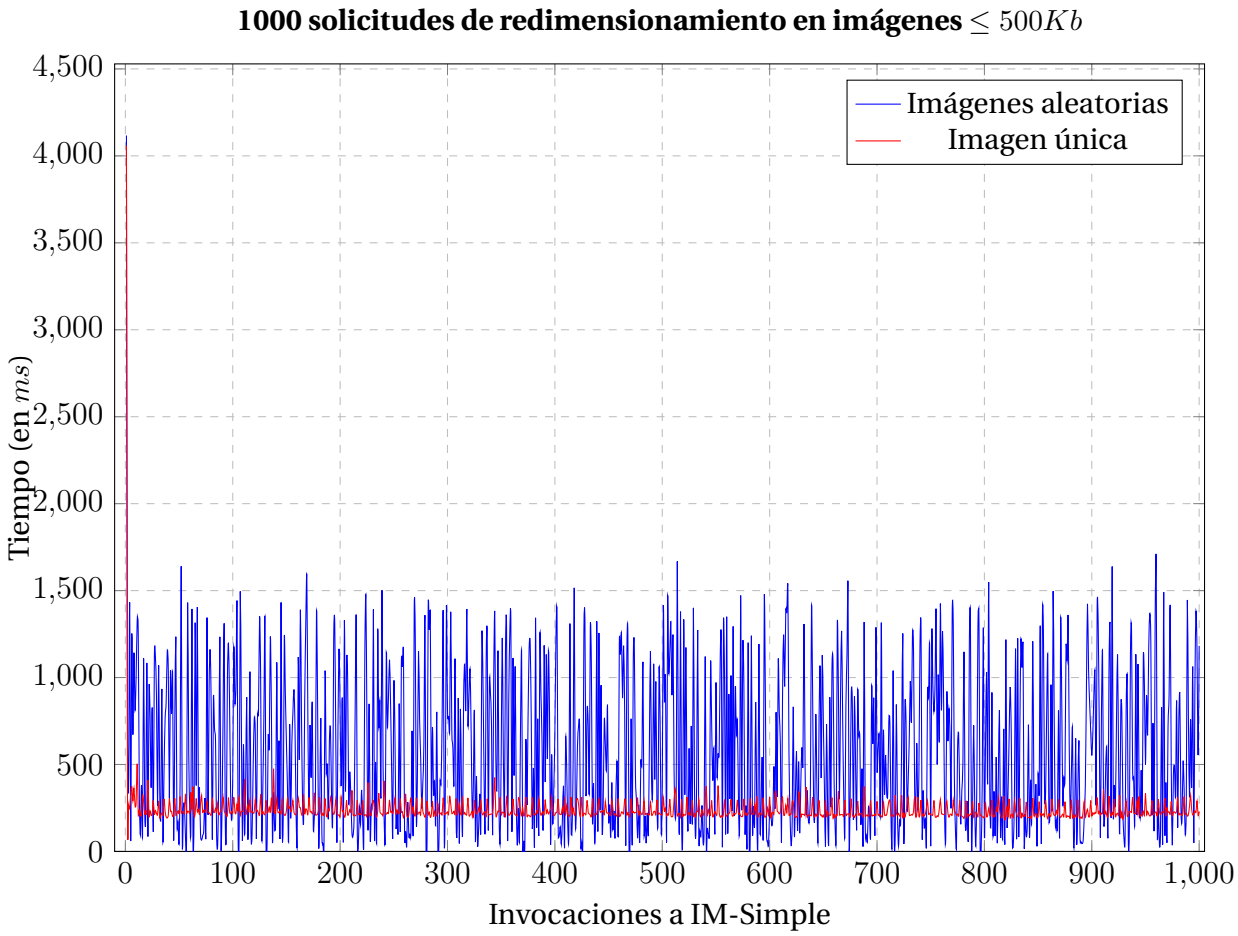

Figura 9.18: 1000 solicitudes de redimensionamiento en imágenes $\leq 500Kb$. La línea azul representa las solicitudes hechas utilizando imágenes aleatorias. La línea roja, las solicitudes utilizando una única imagen.

En la figura 9.18 se muestran 1000 ejecuciones secuenciales utilizando imágenes aleatorias (línea azul) y utilizando una sola imagen (línea roja). La imagen de la línea roja es de tamaño 40Kb. Los resultados de los tiempos de respuesta muestran una tendencia marcada: la primera solicitud tomó mucho más tiempo que el resto. Una vez realizada la primera solicitud, los tiempos de respuesta de la función mejoraron considerablemente y no se observaron en las subsecuentes invocaciones tiempos de respuesta que llegaran a ser similares al primero.

Ejecución de solicitudes de redimensionamiento simultáneas para imágenes de tamaño mayor a 500Kb y menor o igual a 1Mb

Para la realización de este experimento se contó con la siguiente configuración:

- *Sujeto de prueba:* La función Lambda IM-Simple
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño mayor a 500Kb y menor o igual a 1Mb alojadas en Amazon S3.
- *Carga de trabajo:*
    1. 1000 invocaciones secuenciales de redimensionamiento de imágenes con dimensiones aleatorias a IM-Simple.
    2. 1000 invocaciones secuenciales de redimensionamient de una sola imagen a IM-Simple.
- *Herramientas de medición:* Amazon Cloudwatch.

En la carga de trabajo #1, utilizando imágenes aleatorias, la ejecución de las 1000 solicitudes de redimensionamiento fueron las mismas utilizadas para el experimento de la Sección 9.0.1.

La ejecución de las 1000 solicitudes de redimensionamiento tomó 2 horas y 36 minutos. Para la carga de trabajo #2, utilizando una sola imagen, las 1000 solicitudes de redimensionamiento tomaron 2 horas y 30 minutos. Se modificó el *script* en `Bash` del experimento de la Sección 9.0.1 para elegir aleatoriamente una imagen del *cluster* de 1000 y generar 1000 solicitudes secuenciales a partir de esta con dimensiones de ancho y alto estáticas.

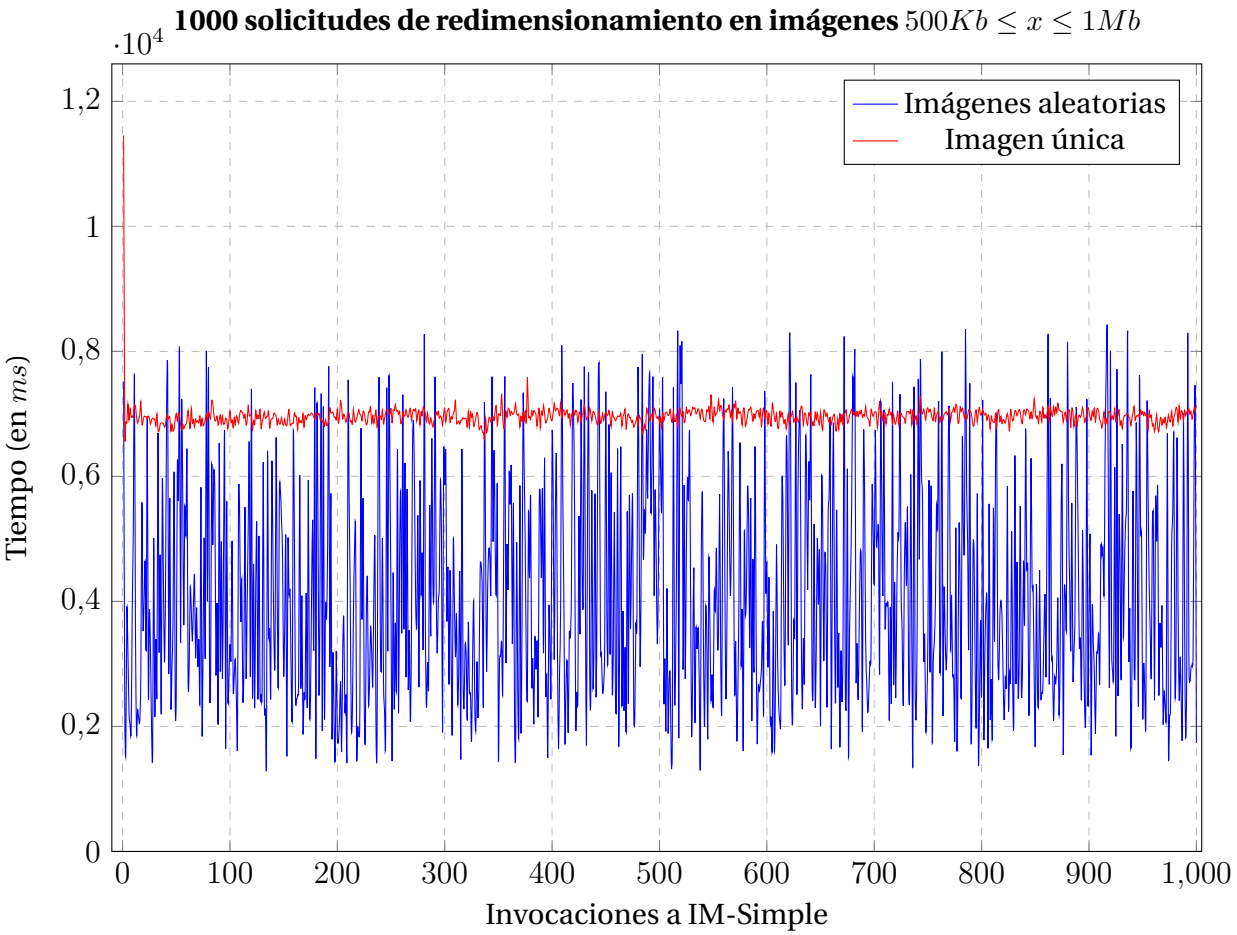

Figura 9.19: 1000 solicitudes de redimensionamiento en imágenes $500Kb \leq x \leq 1Mb$. La línea azul representa las solicitudes hechas utilizando imágenes aleatorias. La línea roja, las solicitudes utilizando una única imagen.

En la figura 9.19 se muestran 1000 ejecuciones secuenciales utilizando imágenes aleatorias (línea azul) y utilizando una sola imagen (línea roja). La imagen de la línea roja es de tamaño 750Kb. Los resultados de los tiempos de respuesta muestran una tendencia marcada: la primer solicitud tomó mucho más tiempo que el resto. Una vez realizada la primer solicitud, los tiempos de respuesta de la función mejoraron considerablemente y, en el caso de las solicitudes de redimensionamiento en imágenes aleatorias, sí se logró observar un caso en el que el tiempo de respuesta fue similar al observado en la primera solicitud. En el caso de la imagen única no se observaron tiempos de respuesta que llegaran

a ser similares al primero en posteriores invocaciones.

Ejecución de invocaciones de redimensionamiento simultáneas para imágenes de tamaño mayor a 1Mb y menor o igual a 2Mb

Para la realización de este experimento se contó con la siguiente configuración:

- *Sujeto de prueba:* La función Lambda IM-Simple
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño mayor a 1Mb y menor o igual a 2Mb alojadas en Amazon S3.
- *Carga de trabajo:*
    1. 1000 invocaciones secuenciales de redimensionamiento de imágenes con dimensiones aleatorias a IM-Simple.
    2. 1000 invocaciones secuenciales de redimensionamiento de una sola imagen a IM-Simple.
- *Herramientas de medición:* Amazon Cloudwatch.

En la carga de trabajo #1, utilizando imágenes aleatorias, la ejecución de las 1000 solicitudes de redimensionamiento fueron las mismas utilizadas para el experimento de la Sección 9.0.1.

La ejecución de las 1000 solicitudes de redimensionamiento tomó 2 horas y 36 minutos. Para la carga de trabajo #2, utilizando una sola imagen, las 1000 solicitudes de redimensionamiento tomaron 2 horas y 16 minutos. Se modificó el *script* en `Bash` del experimento de la Sección 9.0.1 para elegir aleatoriamente

una imagen del *cluster* de 1000 y generar 1000 solicitudes secuenciales a partir de esta con dimensiones de ancho y alto estáticas.

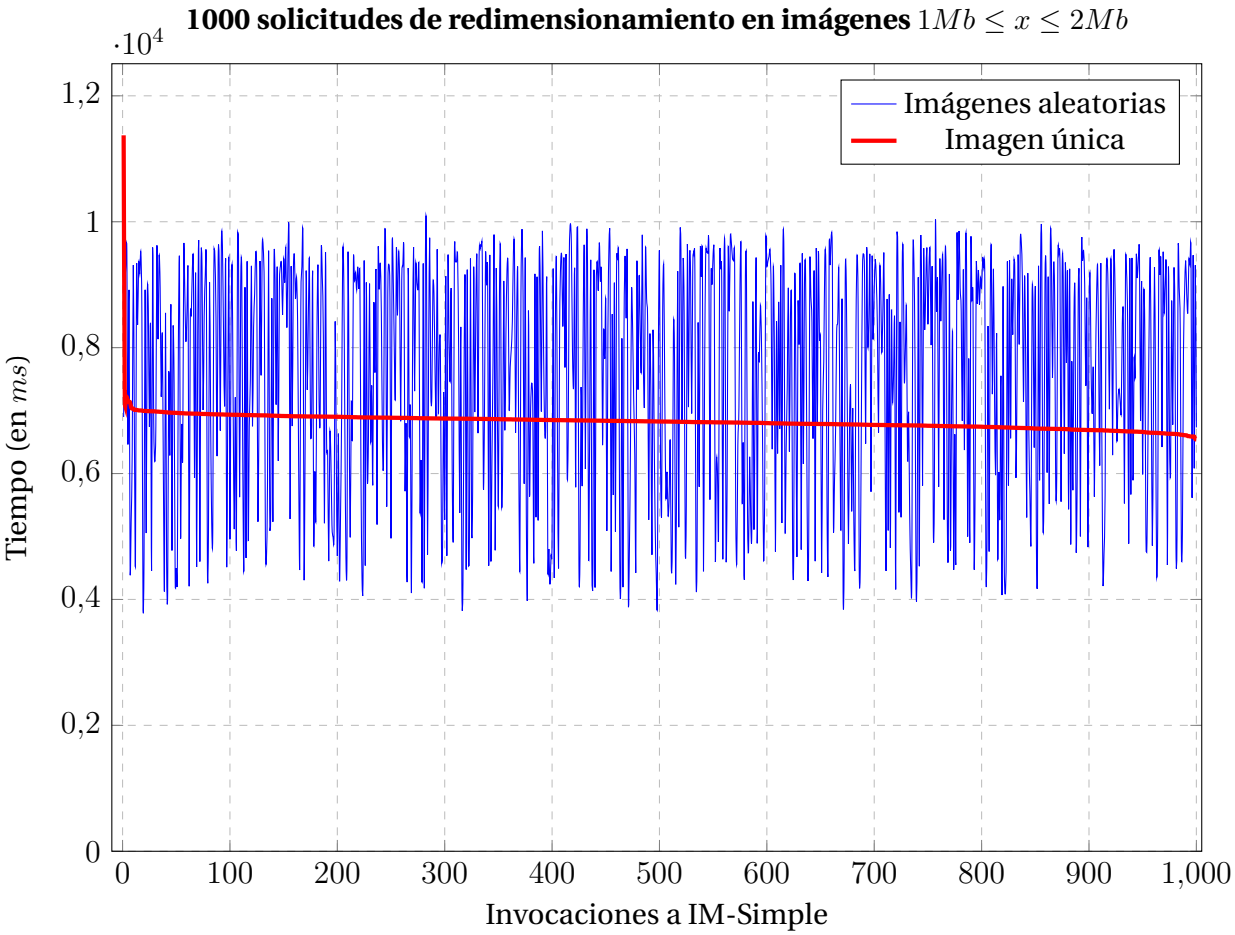

Figura 9.20: 1000 solicitudes de redimensionamiento en imágenes $1Mb \leq x \leq 2Mb$. La línea azul representa las solicitudes hechas utilizando imágenes aleatorias. La línea roja, las solicitudes utilizando una única imagen.

En la figura 9.20 se muestran 1000 ejecuciones secuenciales utilizando imágenes aleatorias (línea azul) y utilizando una sola imagen (línea roja). La imagen de la línea roja es de tamaño 1,4Mb. Los resultados de los tiempos de respuesta muestran una tendencia similar a los casos anteriores: la primer solicitud tomó mucho más tiempo que el resto. Una vez realizada la primer solicitud, los tiempos de respuesta de la función mejoraron considerablemente y, en el caso de las solicitudes de redimensionamiento en imágenes aleatorias, sí se logró observar

un caso en el que el tiempo de respuesta fue similar al observado en la primera solicitud. En el caso de la imagen única no se observaron tiempos de respuesta que llegaran a ser similares al primero en posteriores invocaciones.

Análisis de resultados

Luego de las invocaciones de redimensionamiento tanto en imágenes aleatorias como en una sola imagen, se pudo observar que, si bien en las primeras invocaciones se experimenta mayores tiempos de respuesta, los tiempos de respuesta entregados por la función Lambda lograron bajar significativamente y en los casos de las invocaciones de redimiensionamiento en una sola imagen, estos tiempos lograron mantenerse sumamente estables.

Este comportamiento sugiere que en la invocación de una función Lambda en AWS, la plataforma subyacente primero necesita aplicar tareas de aprovisionamiento para poner a disposición la función y, una vez que esto se realiza, la función paulatinamente va pasando a un estado "caliente" conforme van arribando las invocaciones de redimensionamiento. Aunque la arquitectura del servicio AWS Lambda no se encuentra disponible públicamente, en árticulos en Internet y en proyectos paralelos como SAM Cli y *AWS Lambda container image converter tool*[2] se sugiere el uso de contenedores Docker que son provisionados con el código de la función y que se encargan de procesar las invocaciones entrantes.

Relación con el modelo de rendimiento obtenido El modelo de rendimiento obtenido y las simulaciones hechas son agnósticos a los estados "frío" o "caliente" que se muestran en las observaciones. El principal objetivo de los simula-

[2] `https://github.com/awslabs/aws-lambda-container-image-converter`

dores de PCM es el de obtener múltiples combinaciones de ejecuciones de una arquitectura de software dada y, a menos que un comportamiento o característica de esta logre ser introducido en el modelo es que su impacto debería de verse reflejado en las simulaciones.

Si bien en el modelo obtenido no se introdujo esto explícitamente, las mediciones muestran que los tiempos de respuesta más prolongados se corresponden con el momento en que la función estaba en estado "frío", por ejemplo, durantes las primeras invocaciones. Si se toma como referencia el primer experimento ejecutado en la Sección 9.0.1, invocaciones de redimensionamiento en imágenes menores o iguales a 500Kb, en la figura 9.4, la primera medición a IM-Simple dió como resultado 4.1s y las subsecuentes 999 invocaciones bajaron a 1,6 segundos o menos; muchas de ellas incluso fueron menores a medio segundo. En los resultados de este experimento, existe un 95 % de probablidades que los tiempos de respuesta de las invocaciones sean de 1,6 segundos o menos, lo que, visto desde otro ángulo, quiere decir que existe un 5 % de probabilidad que los tiempos de respuesta de las invocaciones tomen entre 1,6 y 4 segundos en ser procesadas. Ahora bien, aunque 5 % parece ser una probabilidad muy alta y que no refleja lo visto en las mediciones, cuando se vuelve a los resultados de las simulaciones se observa que hay probabilidad mayor al 99 % de que una invocación de redimensionamiento tenga un tiempo de respuesta igual o menor a 2 segundos. Esto quiere decir que hay una probabilidad de menos del 1 % de que una invocación a la función Lambda tome entre 2 y 4 segundos en ser procesada. En el caso de las simulaciones de la Figura 9.4 solamente 6 casos muestran tiempos de respuesta de entre 2 y 4 segundos lo que representa un 0.006 % del total de las simulaciones realizadas.

Lo anterior señala que, para el caso de *Image Handler*, las simulaciones con

los tiempos de respuesta más prolongados representan una cantidad muy pequeña de la totalidad de los casos y que, estos casos se pueden interpretar como potenciales escenarios de la función Lambda en estado frío. Las simulaciones muestran que a pesar del impacto que tienen estos escenarios en las invocaciones, estos realmente no llegan afectar el comportamiento general de la función Lambda.

Con respecto a los resultados de este experimento y su relación con los escenarios de invocaciones de redimiensionamiento en imágenes de tamaño de 500Kb y menor o igual a 1Mb, e imágenes de tamaño de 1Mb y menor a 2Mb, expuestas en la Sección 9.0.1, se puede emplear un análisis similar: las simulaciones que reportan los tiempos de respuesta más prolongados tienen el potencial de caracterizar las ejecuciones de la función en estado “frío”.

9.0.3. Variación del intervalo de las invocaciones de redimensionamiento

Este experimento se plantea como una variante del expuesto en la Sección 9.0.2, donde se evaluaba el efecto de las invocaciones simultáneas a la función Lambda. Acá se plantea variar el intervalo entre invocaciones a la función Lambda con el fin de valorar los efectos en el comportamiento de la función y si el modelo obtenido en la Sección 9.0.1 contribuye a explicar algo de lo observado; en particular si los intervalos entre las invocaciones hacen que el rendimiento de la función se considere “fría” o “caliente” en algún punto.

Estrategia de intervalo de invocaciones a *Image Handler*

Para evaluar los efectos de los intervalos de lanzamiento en las invocaciones a *Image Handler*, se propone la ejecución de un número de invocaciones secuencial (una ráfaga), luego entrar en un tiempo de espera y luego, aplicar otra ráfaga de invocaciones.

El tiempo de espera entre ráfagas se irá incrementando al doble del tiempo de espera anterior. Por ejemplo, si el primer tiempo de espera es de 2 minutos, el siguiente será de 4 minutos, luego 8, 16, 32 minutos y así sucesivamente. Para este experimento se propone utilizar un tiempo de espera inicial de 10 minutos, para luego aumentarlo en subsecuentes ráfagas de invocaciones.

Ejecución de ráfagas de invocaciones de redimensionamiento para imágenes de tamaño menor a 500Kb

Para la realización de este experimento se utilizó la siguiente configuración base:

- *Sujeto de prueba:* La función Lambda IM-XRay.
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño menor o igual a 500Kb alojadas en Amazon S3.
- *Carga de trabajo:* 100 invocaciones secuenciales de redimensionamiento de imágenes con dimensiones aleatorias a IM-XRay seguido de un tiempo de espera de 10, 20 y 40 minutos.
- *Herramientas de medición:* Amazon Cloudwatch y Amazon X-Ray.

De acuerdo con lo observado en los experimentos anteriores, luego de la primera invocación, las subsiguientes hacen que la función entre en un estado "caliente". Por esta razón, ejercitar la función con una ráfaga de 100 invocaciones haría que se logre llegar a este estado fácilmente.

A diferencia de los experimentos anteriores, se utiliza la versión de IM-XRay en lugar de IM-Simple. La razón es que se quiere sacar provecho a las herramientas de monitoreo de AWS X-Ray para determinar si la función pasa de un estado *frío* $\rightarrow$ *caliente* $\rightarrow$ *frío* y determinar el impacto de este cambio en el tiempo de respuesta.

Variando el intervalo de las invocaciones de redimensionamiento en imágenes $\leq 500Kb$

La Figura 9.21 muestra 400 invocaciones de redimensionamiento en imágenes menores a 500Kb divididas en 100 ráfagas cada una. En la primer ráfaga la función se encuentra en estado "frío", sucede el aprovisionamiento inicial. Luego de esta primer invocación la función va entrando paulatinamente en estado "caliente" y se logra observar que para la mayoría de los casos las invocaciones de redimensionamiento no llegan a tomar más de 1,5 segundos. El tiempo promedio de ejecución de una ráfaga fue de 2 minutos y 45 segundos.

En la segunda ráfaga, luego de 10 minutos, la función no experimenta un tiempo de respuesta inicial similar al de la primer ráfaga. Esto quiere decir que pasados 10 minutos de inactividad entre la primera y segunda ráfagas, la función continúa en estado caliente.

En la tercera ráfaga, luego de 20 minutos de haberse ejecutado la última invocación de redimensionamiento de la ráfaga anterior, sí se llega observar un

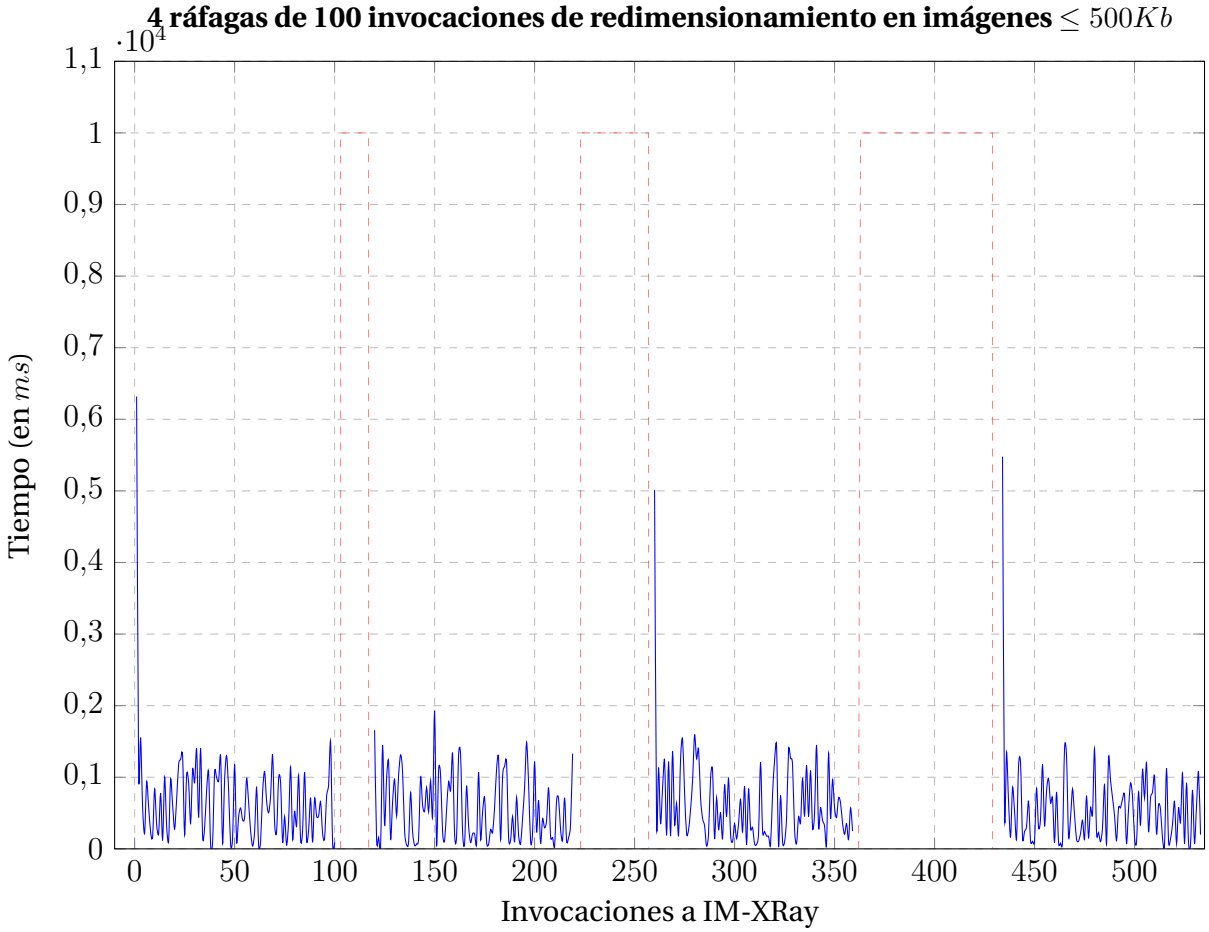

Figura 9.21: El espacio delimitado por la línea punteada roja representa 10 minutos de inactividad entre la primer ráfaga y la segunda. El segundo espacio delimitado por la línea punteada roja representa 20 minutos de inactividad entre la segunda y la tercera ráfaga.

tiempo de respuesta inicial alto. AWS X-Ray reporta que en esta invocación inicial se tuvo que re-aprovisionar la función, lo que sugiere que, durante los 20 minutos de inactividad entre una ráfaga y otra, la función fue pasando progresivamente a un estado "frío" debido a que la plataforma pudo determinar que el nivel de actividad de la función (cantidad de invocaciones) fue decreciendo hasta un punto en donde no detectó actividad alguna.

La última ráfaga también experimentó un tiempo de respuesta inicial alto. Esta vez un poco mayor al de la ráfaga anterior pero menor a la de la primer ráfaga. Hubo 40 minutos de inactividad entre la tercera ráfaga y la cuarta.

Variando el intervalo de las invocaciones de redimensionamiento en imágenes $500Kb \leq x \leq 1Mb$

Se hizo el mismo ejercicio que en la Sección 9.0.3. La Figura 9.22 muestra los resultados de la ejecución de 5 ráfagas de 100 invocaciones cada una. El tiempo promedio de ejecución de una ráfaga fue de 9 minutos y 15 segundos. En la primera invocación de la primer ráfaga se obtuvo un tiempo de respuesta alto (función en estado frío). En la segunda ráfaga no se observó tiempos de respuesta elevados debido a que la función se encontraba en estado "caliente".

En la primera invocación de la tercera y cuarta ráfaga, AWS X-Ray reportó que en ambas hubo que re-aprovisionar la función. Es decir, el tiempo de inactividad entre la segunda ráfaga (20 minutos) y entre la tercera y la cuarta (40 minutos) fue suficiente para que la plataforma volviera a marcar la función como fría de nuevo. Lo interesante de estas dos primeras invocaciones, es que, a pesar que la plataforma tuvo que volver a re-aprovisionar la función, los tiempos de respuesta reportados fueron mucho más bajos que el de la invocación inicial de

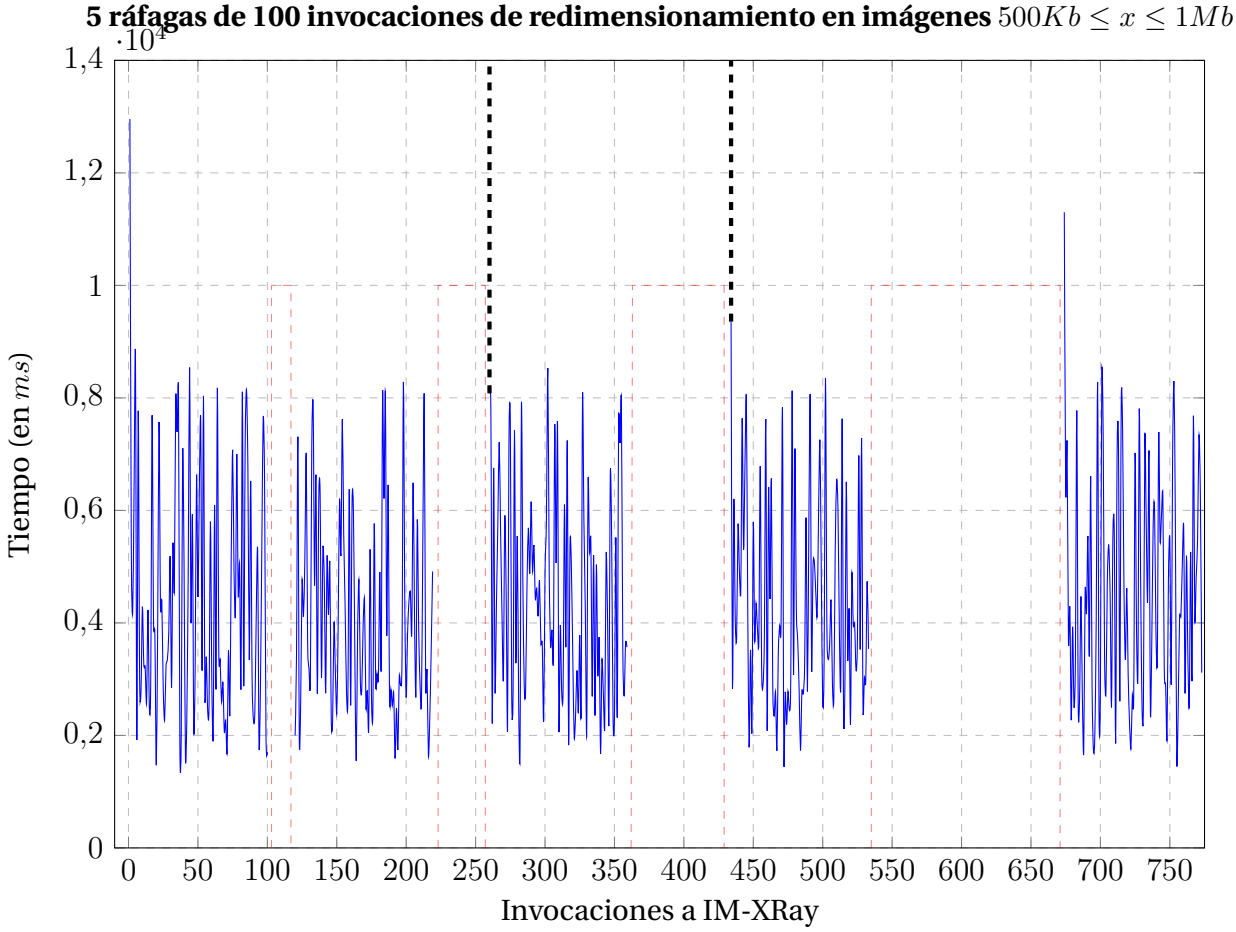

Figura 9.22: El espacio delimitado por la línea punteada roja representa 10 minutos de inactividad entre la primer ráfaga y la segunda. El espacio delimitado por la línea punteada verde representa 20 minutos de inactividad entre la segunda y la tercera ráfaga.

Entre 500Kb a 1Mb	
Ráfaga	Tiempo de 1º invocación
#1	13s
#2	N.A.
#3	8s
#4	9s
#5	11s

Tabla 9.5: Tiempo de respuesta de la primer invocación de redimensionamiento en cada ráfaga.

la primer ráfaga. En la Tabla 9.5 se pueden ver los tiempos de respuesta de las primeras invocaciones de cada ráfaga.

En la quinta ráfaga, el tiempo de respuesta de la primera invocación se elevó de nuevo. El tiempo de inactividad entre la cuarta ráfaga y la quinta fue de 80 minutos.

Debido a que los tiempos iniciales de la tercera y cuarta ráfaga fueron bajos (aunque AWS X-Ray reporta que para cada uno de ellos fue necesaria re-aprovisionar la función), fue que se la ejecución de una quinta ráfaga se hizo necesaria con el fin de validar que, conforme los tiempos de inactividad entre ráfagas aumenta, el tiempo de respuesta de la primera invocación también aumenta.

Variando el intervalo de las invocaciones de redimensionamiento en imágenes $1Mb \leq x \leq 2Mb$

Para este caso, el tiempo promedio de ejecución de una ráfaga fue de 13 minutos y 15 segundos. En la ejecución de cada ráfaga, AWS X-Ray reportó que fue necesario re-aprovisionar la función cuando se ejecutó la primera invocación de cada ráfaga. En la Figura 9.23 se muestran los resultados de la ejecución de

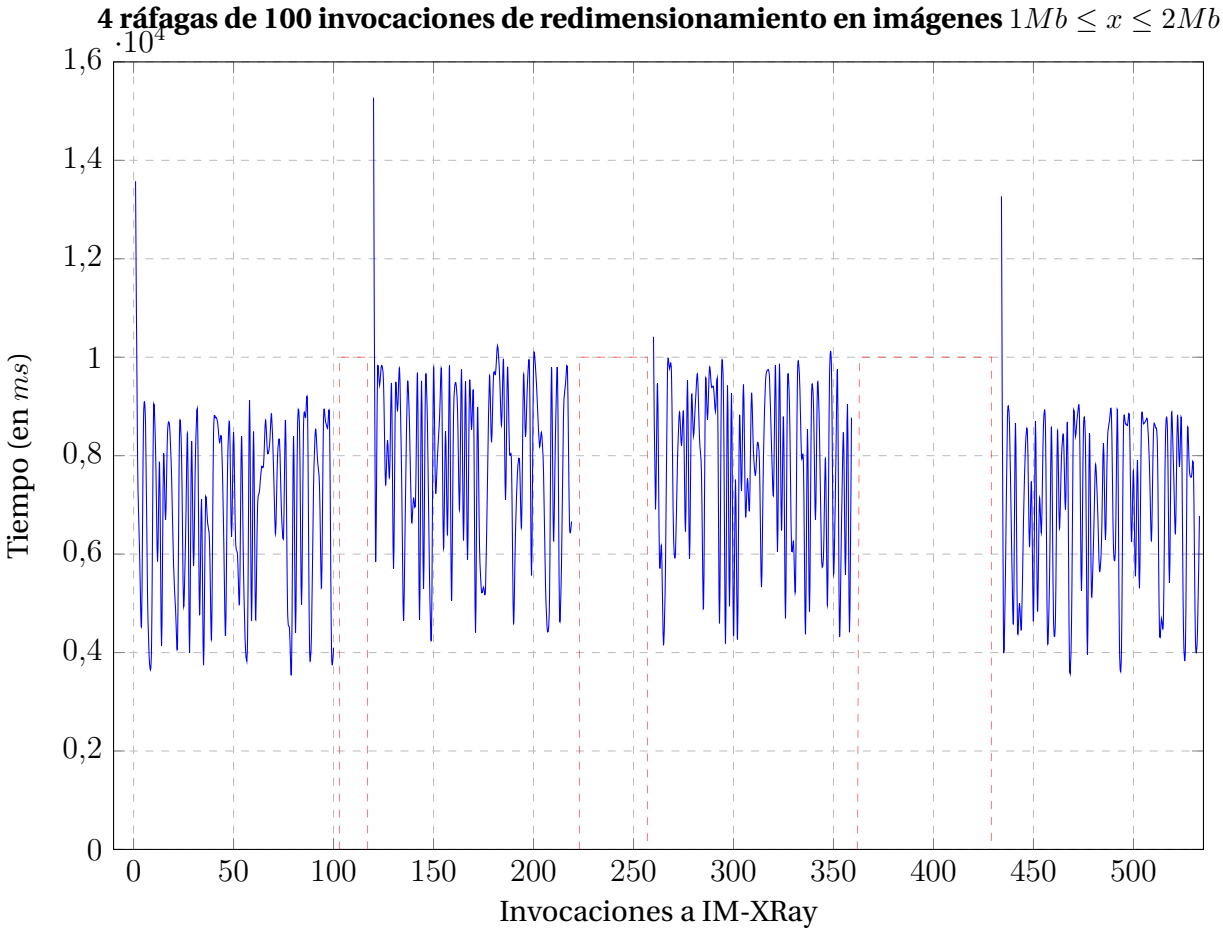

Figura 9.23: El espacio delimitado por la línea punteada roja representa 10 minutos de inactividad entre la primer ráfaga y la segunda. El espacio delimitado por la línea punteada verde representa 20 minutos de inactividad entre la segunda y la tercera ráfaga.

las cuatro ráfagas.

Los tiempos de respuesta de las primeras invocaciones de cada ráfaga resultaron ser más similares entre sí y, también se pudo observar que cuando existen tiempos de inactivad más prolongados, como el que entre la ráfaga tres y la cuando que es de 40 minutos, a la siguiente invocación le toma un mayor tiempo de respuesta que la primer invocación de la ráfaga anterior.

Análisis de resultados

La principal observación que arrojó este experimento es que conforme van aumentando los tiempos de inactividad en la función Lambda, también van aumentando los tiempos de respuesta entregados por la primera invocación luego de este tiempo de inactividad.

De acuerdo con [42], una vez que una función ha sido instalada en la plataforma, la primera invocación debe pasar por el proceso de aprovisionamiento. Luego de que esta primer invocación es procesada, la función pasa a un estado activo o "caliente" y el contenedor que la soporta se puede reutilizar para subsecuentes invocaciones. Cuando se detecta que la función se vuelve inactiva o "fría", el contenedor que la soporta se vuelve candidato a ser "reciclado" para ser utilizado por otra función que lo necesite.

Si bien no hay un tiempo límite definido para decidir cuándo el contenedor va a ser reciclado o no, los resultados de este experimento y los expuestos en [42] y [43] demuestran que entre mayor sean los tiempos de inactividad de una función, mayor será la probabilidad de que el contenedor que la soporta sea reciclado y que, cuando se vuelva a invocar a la función, se experimente un tiempo de respuesta mayor al promedio debido al proceso de aprovisionamiento.

Según las mediciones en [42], el tiempo de vida de un contenedor no parece darse en forma determinística pero se estima que está entre los 25 a 65 minutos. Una contenedor inactivo casi siempre se mantiene vivo por 25 minutos, luego de eso la probabilidad de que sea desechado crece lentamente y llega a alcanzar el 100 % luego de alrededor de 1 hora luego de la última invocación.

Relación con el modelo de rendimiento obtenido En el experimento de la Sección 9.0.1 se obtuvo un modelo de rendimiento a partir una sola ráfaga invocaciones secuenciales de redimensionamiento. No se introdujo tiempos de inactividad.

Aquí, aunque se introducen tiempos de inactividad entre ráfagas, se presenta una tendencia que se ha repetido a lo largo de los tres experimentos ejecutados hasta este momento: las primeras invocaciones reportan mayores tiempos de respuesta, especialmente la primera invocación, y conforme la función va recibiendo más y más invocaciones, su tiempo de respuesta tiende a bajar considerablemente. Esta es una característica que el modelo obtenido logró reconocer y, de igual manera como en los experimentos anteriores, las invocaciones en donde los tiempos de respuesta fueron mayores representan un porcentaje muy bajo ($\approx 1\,\%$) del total de invocaciones en una ráfaga, por lo que cuando el modelo reporte invocaciones con tiempos de respuesta alto se puede afirmar para el caso de *Image Handler* que estas invocaciones serán muy pocas y que probablemente representen momentos en donde la función tuvo que pasar forzozamente de un estado "frío" a uno "caliente".

En PCM, el comportamiento asociado a las invocaciones basadas en ráfagas y tiempos de inactividad podría llegar a ser introducido con un modelo de uso personalizado el cual pueda tomar en cuenta las particularidades del cómo es que una arquitectura de software puede ser potencialmente utilizada pero, la forma en la que PCM reporta los resultados de las simulaciones es la misma independientemente del modelo de uso (o de cualquier otro modelo) utilizado. Es por esto que, en términos de resultados, PCM siempre va a entregar resultados basados en la probabilidad de que una determinada simulación se ejecute en una arquitectura dada. Lo anterior se usa para aclarar que mientras se utilice

PCM para ejecutar simulaciones mediante el esquema de ráfaga/tiempo de espera, los resultados obtenidos no vendrán dados de forma específica para cada ráfaga ejecutada, sino más bien serán presentados de forma general.

9.0.4. Comparación de SAM CLI con observaciones reales de AWS Lambda

Serverless Application Model o SAM CLI es la primera herramienta desarrollada por Amazon para probar funciones Lambda localmente. SAM CLI permite desarrollar funciones Lambda en cualquiera de los lenguajes de programación soportados por el servicio AWS Lambda y probarlos sin necesidad de instalarlos directamente en el servicio. Para emular el servicio, SAM CLI, hace uso de Docker para instalar y ejecutar el código de una función en un contenedor especializado para tal fin.

SAM CLI no es la primera herramienta desarrollada para probar, instalar y emular funciones Lambda en un entorno local, pero si es la primera desarrollada por Amazon por lo que se espera que mediante su uso se pueda tener una aproximación muy cercana del comportamiento que va a tener una función Lambda cuando esté en producción.

En este experimento se propone explorar el comportamiento de *Image Handler* cuando se ejecutan invocaciones de redimensionamiento en SAM CLI y, comparar lo obtenido con invocaciones de redimensionamiento efectuadas en el servicio de AWS Lambda en producción. Los resultados de este experimento permitirán determinar si las invocaciones hechas en SAM CLI entregan tiempos de respuesta similares a los del servicio AWS Lambda para que, de esta forma se pueda considerar SAM CLI como una herramienta confiable para simulación de

invocaciones de rendimiento.

Estrategia de comparación de invocaciones de redimensionamiento en `SAM CLI` y AWS Lambda

Similar a los experimentos anteriores, se propone ejecutar 1000 ejecuciones de redimensionamiento a *Image Handler*. Para valorar la similitud entre los resultados, el conjuntos de invocaciones utilizadas serán las mismas para `SAM CLI` y AWS Lambda: la $k-$ésima invocación a *Image Handler* realizada por medio de `SAM CLI` será igual a la $k-$ésima invocación realizada en AWS Lambda.

Ejecución de 1000 invocaciones de redimensionamiento para imágenes de tamaño menor a 500Kb

Para la realización de este experimento se utilizó la siguiente configuración base:

- *Sujeto de prueba:* La función lambda IM-Simple.
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño menor o igual a 500Kb.
- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes en IM-Simple.
- *Herramientas de medición:* Amazon Cloudwatch.

Configuración para la realización del experiento en `SAM CLI`:

- *Sujeto de prueba:* La función lambda IM-Simple.

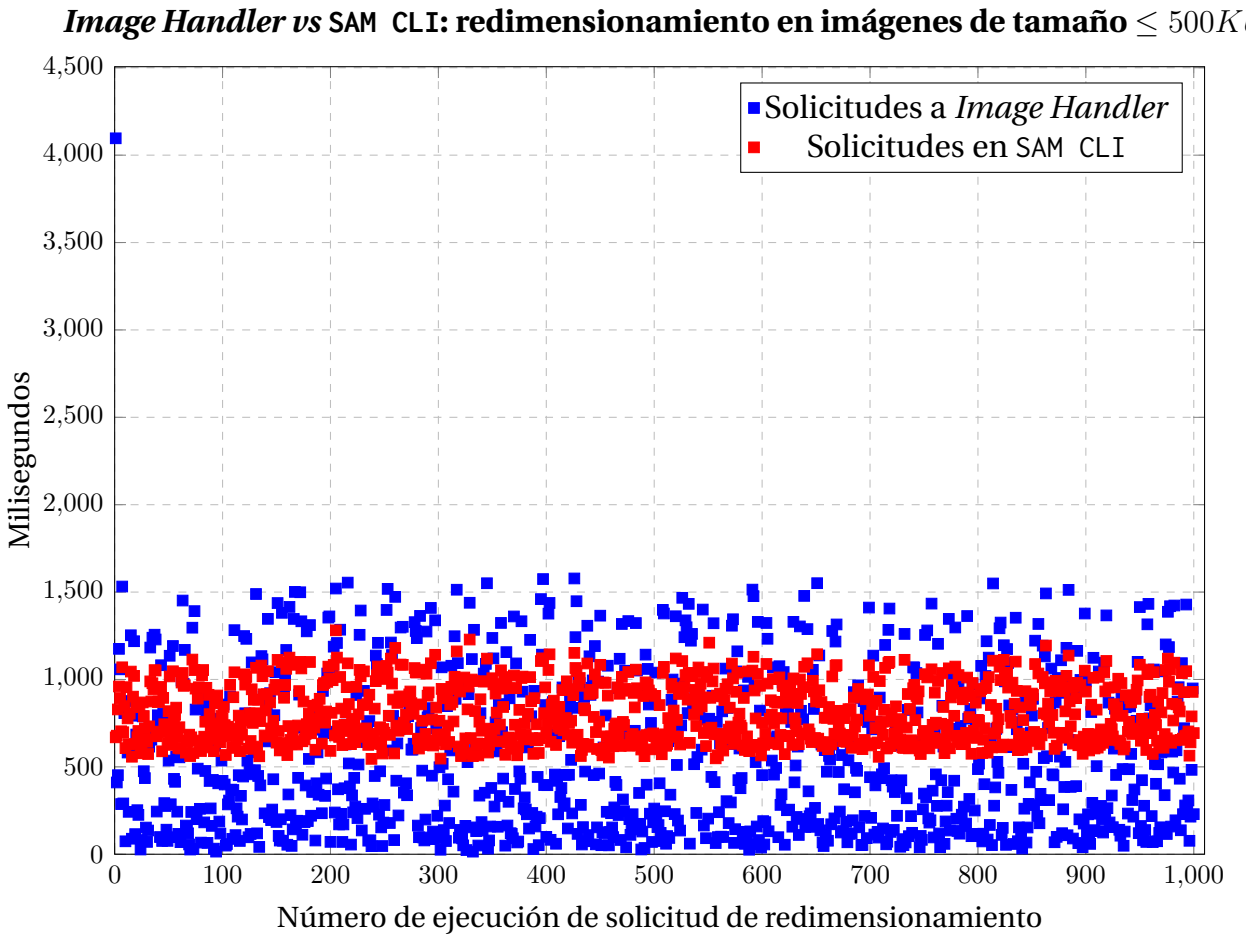

Figura 9.24: Solicitudes de redimensionamiento de imágenes de tamaño $1Mb \leq x \leq 2Mb$

- *Ambiente:* `SAM CLI` instalado en máquina virtual alojada en el servicio AWS EC2[3], de tipo `t2.large`[4].
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño menor o igual a 500Kb.
- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes en IM-Simple.
- *Herramientas de medición:* Bitácoras de `SAM CLI`.

[3] `https://aws.amazon.com/ec2/`
[4] `https://aws.amazon.com/ec2/instance-types/t2/`

AWS Lambda vs SAM CLI: Hasta 500Kb			
	AWS Lambda	SAM CLI	Diferencia
Duración	15min	90min	–
Tiempo promedio	571.15ms	804.35ms	233.20ms
Desviación estándar	449.76ms	162.22ms	287.54ms
Varianza	202281.41	26314.94	–
Mediana	698.28ms	771.52ms	–
Coeficiente de variación	0.64	0.21	–

Tabla 9.6: Resumen de datos estadísticos

Resultados: La Figura 9.24, muestra la comparación de los tiempos de respuesta de las 1000 invocaciones realizadas y en la tabla 9.6 un resumen de los datos estadísticos obtenidos. A primera vista se puede ver que el conjunto de tiempos de respuesta de las invocaciones cuando se utilizó SAM CLI fue más homogéneo que las invocaciones en AWS Lambda. Esto también se refleja en los coeficientes de variación: 0.64 y 0.21 para las invocaciones en AWS Lambda y SAM CLI respectivamente, lo que claramente sugiere que el conjunto de tiempos de respuesta obtenido de las invocaciones de SAM CLI es más homogéneo que el de AWS Lambda. Esto también lo confirma una menor desviación estándar en el conjunto de tiempos de respuesta de las invocaciones hechas mediantes SAM CLI.

En la tabla 9.6, el tiempo promedio de las invocaciones realizadas en AWS Lambda fue más bajo que el de SAM CLI. En SAM CLI, el menor tiempo de respuesta fue de 545.74ms mientras que en los resultados obtenidos de AWS Lambda aproximadamente el 56 % de los resultados estuvo por debajo de los 545.74ms. Esto indica que a pesar que el conjunto de los tiempos de respuesta de SAM CLI es más homogéneo, los tiempos obtenidos de AWS Lambda muestran un mejor rendimiento.

Ejecución de 1000 invocaciones de redimensionamiento para imágenes de tamaño $500Kb \leq x \leq 1Mb$

Para la realización de este experimento se utilizó la siguiente configuración base:

- *Sujeto de prueba:* La función lambda IM-Simple.
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño mayor a 500Kb y menor o igual a 1Mb.
- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes en IM-Simple.
- *Herramientas de medición:* Amazon Cloudwatch.

Configuración para la realización del experiento en `SAM CLI`:

- *Sujeto de prueba:* La función lambda IM-Simple.
- *Ambiente:* `SAM CLI` instalado en máquina virtual alojada en el servicio AWS EC2, de tipo `t2.large`.
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño mayor o igual a 500Kb y menor a 1Mb.
- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes en IM-Simple.
- *Herramientas de medición:* Bitácoras de `SAM CLI`.

***Image Handler vs* `SAM CLI`: redimensionamiento en imágenes de tamaño** $500Kb \leq x \leq 1Mb$

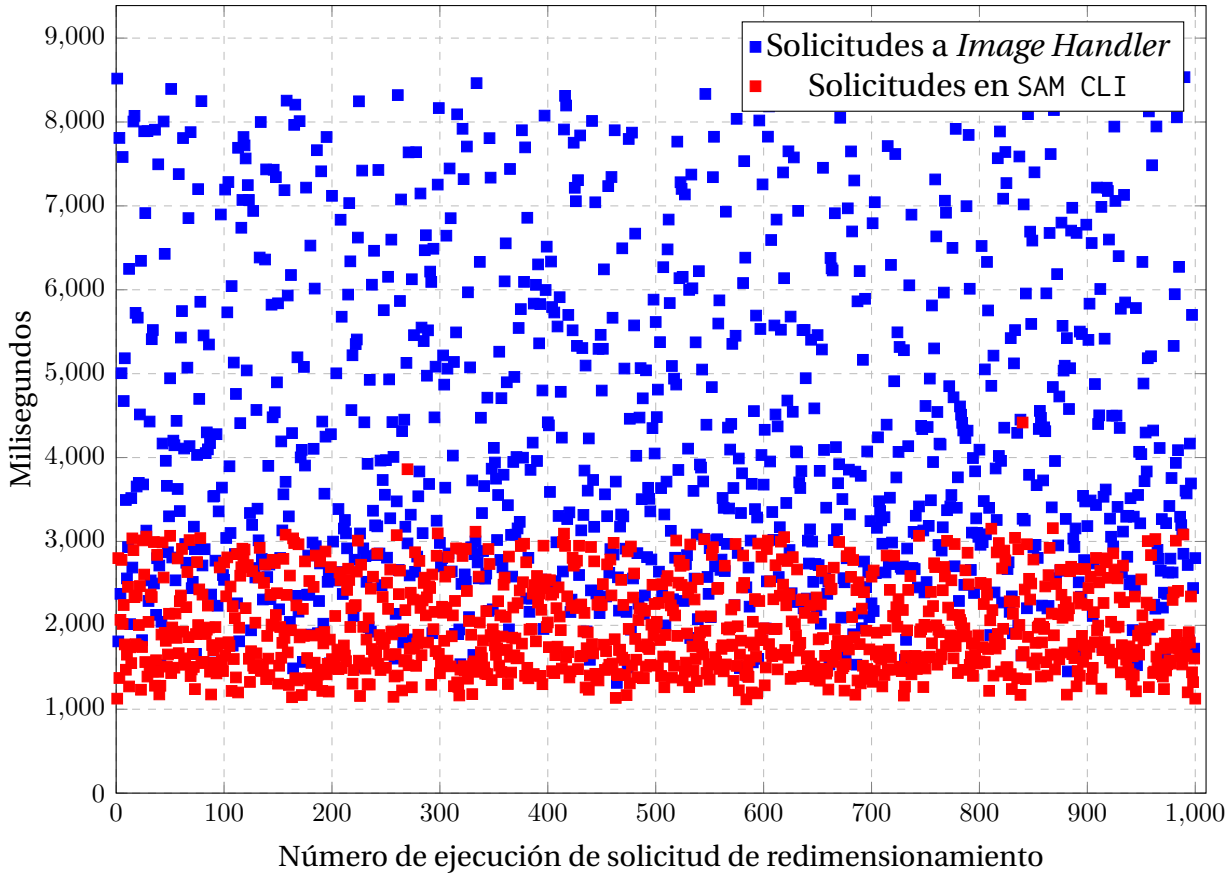

Figura 9.25: Solicitudes de redimensionamiento de imágenes de tamaño $500kb \leq x \leq 1Mb$

AWS Lambda vs SAM CLI: Hasta 1Mb			
	AWS Lambda	`SAM CLI`	Diferencia
Duración	72min	218min	–
Tiempo promedio	4345.80ms	1969.49ms	2376.31ms
Desviación estándar	1867.64ms	524.95ms	1342.69ms
Varianza	3488063	275569.63	–
Mediana	3898.82ms	1851.99ms	–
Coeficiente de variación	0.48	0.28	–

Tabla 9.7: Resumen de datos estadísticos

Resultados: El 99.8 % del conjunto de los tiempos de respuesta en SAM CLI se agruparon entre los 1.1 y 3.1 segundos mientras que el 66.8 % del conjunto de los tiempos de respuesta en AWS Lambda estuvieron por encima de los 3.1 segundos. Los tiempos de respuesta de las mediciones hechas en SAM CLI se siguen mostrando mucho más homogéneos que los de AWS Lambda pero, al igual que en el experimento anterior, esto no es una indicación de similitud entre los dos conjuntos de datos.

En general, los resultados obtenidos de SAM CLI muestran mejor rendimiento que los de AWS Lambda pero no logran caracterizar lo visto en las observaciones en el ambiente de producción. Los tiempos promedios de 4345.80ms y 1969.49ms en AWS Lambda y SAM CLI respectivamente aunado con los valores de las desviaciones estándar de 1867.64ms y 524ms para el mismo caso, sugiere que estos conjuntos de datos tienen comportamientos diferentes entre sí.

Por último, a SAM CLI le tomó 218 minutos en procesar las 1000 invocaciones de redimensionamiento, mientras que a AWS Lambda le tomó 72 minutos. Individualmente, las invocaciones de redimensionamiento de SAM CLI muestran mejores tiempos, pero el proceso en general toma más tiempo. A SAM CLI le toma más tiempo aprovisionar sus recursos antes de cada invocación, pero en la bitácora solamente se reporta el tiempo de procesamiento de la invocación como tal. En las condiciones actuales no es posible observar el tiempo de aprovisionamiento en SAM CLI.

Ejecución de 1000 invocaciones de redimensionamiento para imágenes de tamaño $1Mb \leq x \leq 2Mb$

Para la realización de este experimento se utilizó la siguiente configuración base:

- *Sujeto de prueba:* La función lambda IM-Simple.
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño mayor a 1Mb y menor o igual a 2Mb.
- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes en IM-Simple.
- *Herramientas de medición:* Amazon Cloudwatch.

Configuración para la realización del experiento en `SAM CLI`:

- *Sujeto de prueba:* La función lambda IM-Simple.
- *Ambiente:* `SAM CLI` instalado en máquina virtual alojada en el servicio AWS EC2, de tipo `t2.large`.
- *Repositorio de imágenes: Cluster* de 1000 imágenes de tamaño mayor o igual a 1Mb y menor a 2Mb.
- *Carga de trabajo:* 1000 invocaciones secuenciales de redimensionamiento de imágenes en IM-Simple.
- *Herramientas de medición:* Bitácoras de `SAM CLI`.

***Image Handler vs* SAM CLI: redimensionamiento en imágenes de tamaño** $1Mb \leq x \leq 2Mb$

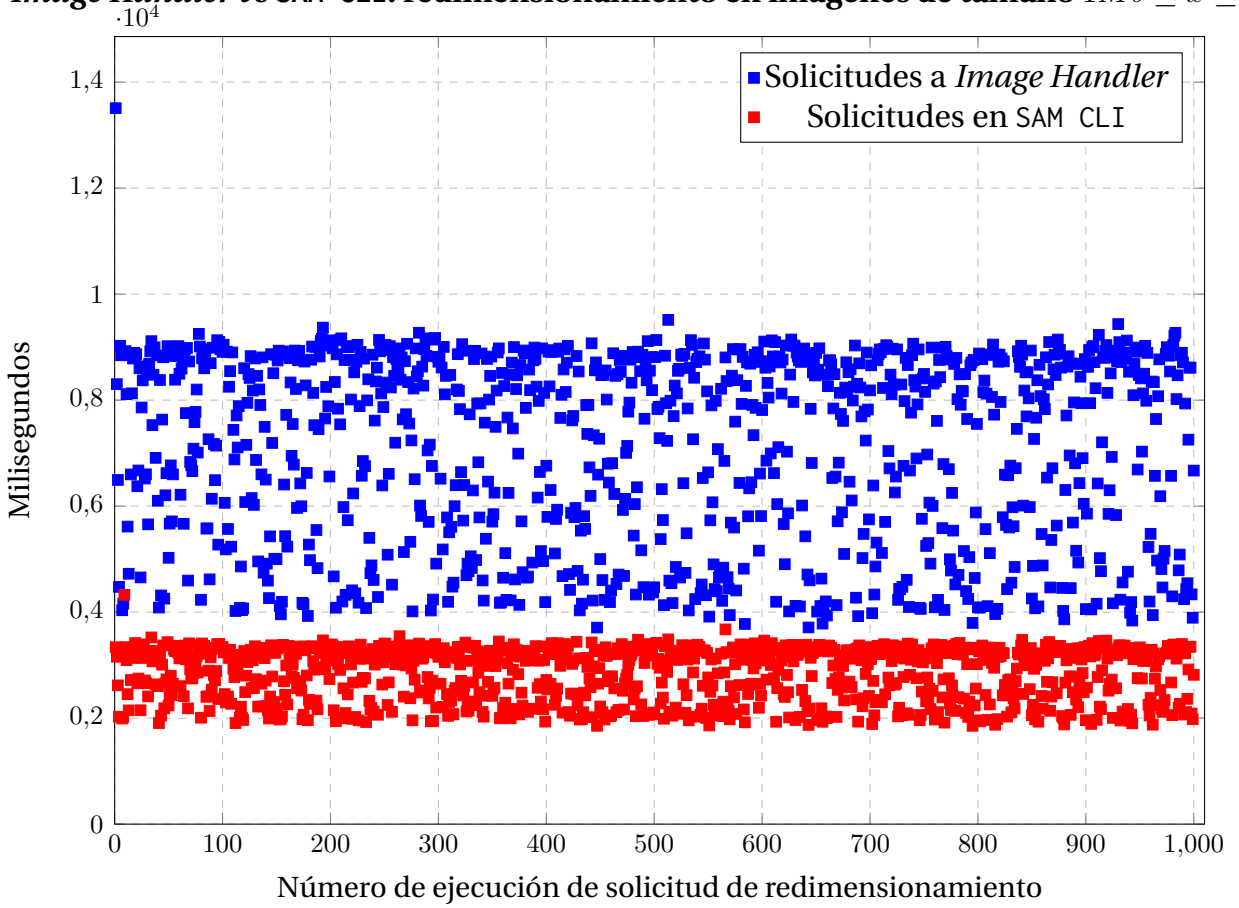

Figura 9.26: Solicitudes de redimensionamiento de imágenes de tamaño $1Mb \leq x \leq 2Mb$

AWS Lambda vs SAM CLI: Hasta 2Mb			
	AWS Lambda	SAM CLI	Diferencia
Duración	120min	225min	–
Tiempo promedio	7113.377ms	2840.91ms	4272.464ms
Desviación estándar	1768.520ms	494ms	1268.523ms
Varianza	3106479.237	244032.83	–
Mediana	7785.15ms	3053.72ms	–
Coeficiente de variación	0.226	0.16	–

Tabla 9.8: Resumen de datos estadísticos

Resultados En este experimento, los dos conjuntos obtenidos resultaron a ser los más disímiles entre sí. Solamente el 0,2 % de las invocaciones realizadas por medio de `SAM CLI` se encontraron en el dentro del rango de tiempo de las invocaciones directas en AWS Lambda. Los datos de la tabla 9.8 confirman que estos dos conjuntos de datos se comportan de manera muy diferente.

La duración total de todas las invocaciones de redimensionamiento fue de 120 minutos cuando se utilizó la función Lambda de forma directa y de 225 minutos con `SAM CLI`, similar a los tiempos del experimento anterior. Los tiempos de respuesta obtenidos cuando se usó `SAM CLI` fueron mejores que las invocaciones directas pero estas no pueden ser utilizadas como una referencia para predecir el comportamiento de la función Lambda cuando se redimensionen imágenes de tamaño a 1Mb y menor o igual 2Mb.

`SAM CLI` como herramienta para la simulación en *Image Handler*

Los resultados de los experimentos realizados no sugieren que el uso de `SAM CLI` sea adecuado como herramienta para tener algún grado de predicción del cómo se van a comportar las invocaciones de redimensionamiento a *Image Handler* cuando esté ejecutándose en AWS Lambda.

El escenario que presentó mayor similitud fue con las invocaciones de redimensionamiento en imágenes menores a los 500Kb. En este caso, al menos el 100 % de las invocaciones realizadas por medio de `SAM CLI` se ubicaron dentro del rango de los tiempos de respuesta cuando se usó AWS Lambda. Conforme se fue incrementando el tamaño de las imágenes a redimensionar, las diferencias en los tiempos de respuesta de un conjunto con respecto al otro se hicieron más notables. Esto hace que para el caso de *Image Handler*, `SAM CLI` no se conside-

re confiable para caracterizar el comportamiento de la función en un ambiente en producción. Pese a esto, no se pone en duda la utilidad de `SAM CLI` para la implementación, pruebas e instalación de funciones Lambda en AWS.

Some UNADECA, Greatest Common Divisor GCD

9.0.5. Resumen de resultados de los cuatros principales experimentos propuestos

Experimento #1 El primer experimento de la Sección 9.0.1 estuvo dirigido a generar un modelo de rendimiento a partir de las bitácoras de la función *Image Handler*. Para esto, se implementó una versión de la función que estuvo instrumentalizada con la biblioteca de Kieker para generar una bitácora en dónde registrar los datos del rendimiento de los eventos asociados a la entrada, obtención de la imagen, redimensionamiento y salida de la función. Luego, se tomó una bitácora y se le pasó a PMX para extraer un modelo de rendimiento en formato PCM.

Una vez obtenido este modelo de rendimiento, se procedió a crear tres experimentos en donde se realizaron mil invocaciones de redimensionamiento sobre tres grupos de imágenes y, al mismo tiempo, se realizaron refinamientos en los componentes del modelo de rendimiento para reflejar los cambios en la demanda de la función para ejecutar mil simulaciones en *Palladio Workbench*.

Revisar esta redaccion con ITZ Por último se compararon los resultados de las invocaciones reales de redimensionamiento con los de los de las simulaciones las cuales pudieron caracterizar el comporamiento de la función Lambda con una probabilidad del 95 %.

Experimento #2 El experimento de la Sección 9.0.2 se planteó para explorar el comportamiento de la función Lambda cuando esta era ejercitada durante un periodo continuo y, al mismo tiempo, evaluar si el modelo de rendimiento obtenido en el experimento #1 podría explicar algo del comportamiento observado aquí.

Se ejecutaron ráfagas de mil invocaciones de redimensionamiento secuenciales utilizando imágenes aleatorias y otras ráfagas de mil invocaciones utilizando la misma imagen. Las invocaciones se realizaron sobre los tres mismos grupos de imágenes definidos en el experimento #1.

Como resultado de la ejecución de las mil ráfagas se pudo notar con mayor detalle cómo la función Lambda arranca inicialmente en un estado "frio" y conforme va recibiendo invocaciones, va pasando paulatinamente a un estado "caliente", tal y como lo señalaba la literatura consultada. Con respecto a los resultados de este experimento y el modelo de rendimiento obtenido del experimento #1, se observó que hay una relación tanto entre las simulaciones que reportaron los tiempos de respuesta más prolongados con los tiempos de respuesta de la función cuando se encontraba en estado "frío", así como los tiempos de respuesta promedio con los tiempos de respuesta de la función en estado "caliente".

Experimento #3 Descrito en la Sección 9.0.3 y que es una variante del experimento #2. Se introdujeron tiempos de inactividad entre ráfagas de invocaciones de redimensionamiento y, como en el experimento #2, se buscó evaluar el comportamiento de la función ante estas ráfagas y la posible relación de este comportamiento con el modelo de rendimiento.

Se ejecutaron ráfagas de cien invocaciones de redimensionamiento y se introdujo tiempos de inactividad de 10, 20, 40 y 80 minutos entre cada ráfaga. Se utilizaron los mismos tres grupos de imágenes del experimento #1 en las invocaciones de redimensionamiento.

Conforme se fue incrementando el tiempo de inactividad entre ráfagas, observamos un incremento también en la posibilidad de que la función cayera en un estado "frío" y aunque este fue un comportamiento que no fue introducido en el modelo de rendimiento explícitamente, sí se pudo notar una correspondencia entre los tiempos de respuesta de la función Lambda en estado "frío" y "caliente" en ráfagas individuales con los resultados de las simulaciones.

Experimento #4 En este experimento se puso a prueba si el uso de la herramienta `SAM CLI` podía considerarse confiable para simular y tener algún grado de predicción del comportamiento de una función Lambda en producción.

Para probar esto, se realizaron mil invocaciones de redimensionamiento en cada una de las dos plataformas, de manera que las secuencia de imágenes fuera idéntica en ambos casos. Los resultados de las invocaciones demostraron que el comportamiento en `SAM CLI` resultó ser muy diferente al de AWS Lambda y, al menos para el caso de *Image Handler* no resultó ser de utilidad para obtener una estimación de la ejecución de la función en un ambiente de producción.

Capítulo 10

Guía Metodológica

En este capítulo se introducen los actividades que se llevaron a cabo para la extracción de un modelo de rendimiento a partir de una función Lambda, tal y como se hizo en la Sección 9.0.1.

10.1. Selección del caso de uso

```
1  /* Usuarios expertos podrían saltarse el proceso de
      instrumentalización y extracción                          */
2  si se cuenta con un modelo PCM existente entonces
3  |  utilizar este modelo para simulaciones y pruebas;
4  en otro caso
5  |  si lenguaje de la función está escrita en Java entonces
6  |  |  se puede utilizar la biblioteca(SDK) de Kieker;
7  |  en otro caso
8  |  |  implementar integración con Kieker explícitamente;
9  |  fin
10 |  si plataforma FaaS es AWS Lambda entonces
11 |  |  la integración con Kieker se da por medio de JMS;
12 |  |  se pueden utilizar las herramientas de monitoreo de AWS;
13 |  en otro caso
14 |  |  /* La integración con Kieker podría ser similar a la
   |  |     propuesta o no                                      */
15 |  |  implementar integración con Kieker explícitamente;
16 |  |  hacer uso de herramientas de medición alternas;
17 |  fin
18 fin
```

Algoritmo 1: How to write algorithms

Hoy en día se cuenta con una amplia variedad de opciones tanto en plataformas *FaaS* como en lenguajes de programación que pueden ser utilizados para el desarrollo de funciones Lambda. En la Sección 2.1.3 se dan a conocer varias de estas opciones.

La selección del lenguaje de programación se hace especialmente importante si es que se desea seguir el enfoque de extracción y obtención de modelos de rendimiento expuesto en este trabajo. Al momento de escribir este documento, el único lenguaje soportado por Kieker, la herramienta que gestiona la bitácora con los eventos producidos por la función Lambda, es Java.

Esto lo que quiere decir es que actualmente Kieker cuenta con bibliotecas

escritas en Java que pueden ser fácilmente incluidas en el proyecto que se está desarrollando y de esta forma lograr una integración con algún servicio de bitácoras de eventos de Kieker. Si bien se puede lograr la integración entre Kieker y otro lenguaje de programación aparte de Java, esta integración deberá ser proporcionada por el implementador de forma explícita.

La plataforma utilizada para la prueba e instalación de la función Lambda fue AWS Lambda. Tal y como se describió en el Capítulo 8 y en la Sección 9.0.1, se hizo uso de herramientas de gestión de bitáboras (CloudWatch) y de monitoreo (AWS X-Ray) para obtener mediciones alternativas de la función que fueron de gran ayuda a la hora de refinar el modelo de rendimiento. En el caso que se desee implementar un caso de uno similar a *Image Handler* en algún otro proveedor de servicios *FaaS* (Sección 2.1.3), se deberá de tomar en consideración las herramientas y particularidades de dicha plataforma para la obtención de mediciones de rendimiento.

El enfoque de trabajo expuesto en este documento sigue una serie de actividades que se consideran útiles para la extracción y obtención de modelos de rendimiento de una función Lambda. Estas actividades fueron adoptadas luego de estudiar la literatura recomendada y, debido a que en este trabajo se está haciendo una exploración inicial a este método de trabajo, se decidió seguir estas actividades de acuerdo al orden recomendado.

Implementadores con mayor experiencia en el modelado y simulación basado en componentes podrían prescindir o modificar partes de las actividades expuestas con el fin de obtener resultados más directos de los aquí expuestos. En particular se podría:

- Proporcionar un modelo en PCM *a priori* de una función Lambda y de esta

forma saltarse el proceso de instrumentalización y extracción.

- Utilizar herramientas de medición alternas para la refinamiento del modelo.

10.2. Instrumentalización con Kieker

Se tiene que descargar Kieker de `http://kieker-monitoring.net/`, instalarlo y ejecutarlo en una computadora en donde se pueda tener acceso y control para manipular las bitáboras obtenidas. Se recomienda la lectura del manual de usuario de Kieker, disponible en [41], para obtener mayores detalles en su instalación y configuración.

Kieker permite varias formas de configuración y ejecución. Debido a que las funciones Lambda se ejecutan en ambientes que son inaccesibles por los implementadores, se optó por configurar Kieker para que escuche los eventos de la función por medio de una cola JMS. La cola JMS y el servicio de Kieker fueron instalados y configurados en una máquina virtual independiente.

En la Sección 8.2.2 se introduce la programación requerida para generar entradas en una bitádora de Kieker. Se generaron eventos de tipo `OperationExecutionRecord` que son los tipos de regitros más básicos que se pueden producir en Kieker y, además, se configuró la biblioteca de Kieker para que publicara los eventos de rendimiento a una cola JMS. El servicio de Kieker consumía los eventos de la cola y los registraba en una bitácora local.

$$\text{Función Lambda} \rightarrow \text{JMS} \rightarrow \text{Kieker} \rightarrow \text{Bitácora}$$

Se recomienda la lectura del manual de usuario y del tutorial de Kieker dis-

ponibles en [41] y en [44] respectivamente para tener mayores detalles sobre otros tipos de eventos que pueden ser generados por la biblioteca de Kieker.

10.3. Extracción del modelo con PMX

Capítulo 11

Conclusiones

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. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering (FOSE '07)*, pp. 171–187, May 2007.

[4] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Perform. Eval.*, vol. 67, pp. 634–658, Aug. 2010.

[5] T. de Gooijer, "Performance modeling of `ASP.Net` web service applications : an industrial case study," 2011.

[6] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann, *Modeling and Simulating Software Architectures: The Palladio Approach*. The MIT Press, 2016.

[7] Y. Jin, A. Tang, J. Han, and Y. Liu, "Performance evaluation and prediction for legacy information systems," in *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, (Washington, DC, USA), pp. 540–549, IEEE Computer Society, 2007.

[8] O.-Q. Noorshams, *Modeling and Prediction of I/O Performance in Virtualized Environments*. PhD thesis, 2015. Disponible en: `http://dx.doi.org/10.5445/KSP/1000046300`.

[9] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2002.

[10] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey," *IEEE Transactions on Software Engineering*, vol. 30, pp. 295–310, May 2004.

[11] A. Koziolek, H. Koziolek, and R. Reussner, "Peropteryx: Automated application of tactics in multi-objective software architecture optimization," in *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, QoSA-ISARCS '11, (New York, NY, USA), pp. 33–42, ACM, 2011.

[12] S. Kounev, F. Brosig, and N. Huber, "The Descartes Modeling Language," tech. rep., Department of Computer Science, University of Wuerzburg, October 2014.

[13] G. Brataas, E. Stav, S. Lehrig, S. Becker, G. Kopčak, and D. Huljenic, "Cloudscale: Scalability management for cloud systems," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, (New York, NY, USA), pp. 335–338, ACM, 2013.

[14] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, pp. 3–22, Jan. 2009.

[15] P. M. Mell and T. Grance, "Sp 800-145. the nist definition of cloud computing," tech. rep., Gaithersburg, MD, United States, 2011.

[16] K. Craig-Wood, "IaaS vs. PaaS vs. SaaS definition," May 2010. https://www.katescomment.com/iaas-paas-saas-definition/, obtenido el 27 de Octubre del 2018.

[17] "Serverless computing - Amazon Web Services," 2008. Obtenido el 26 Setiembre del 2018 de https://aws.amazon.com/serverless.

[18] M. Roberts, "Serverless architectures," 2018. Obtenido el 25 Setiembre del 2018 de https://martinfowler.com/articles/serverless.html.

[19] N. Novkovic, "Top function as a service (FaaS) providers," May 2018. https://dashbird.io/blog/top-function-as-a-service-faas-providers/, obtenido el 22 de Noviembre del 2018.

[20] M. Cavallari and F. Tornieri, "Information systems architecture and organization in the era of microservices," in *Network, Smart and Open* (R. Lamboglia, A. Cardoni, R. P. Dameri, and D. Mancini, eds.), (Cham), pp. 165–177, Springer International Publishing, 2018.

[21] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *2017 IEEE/ACM 1st*

International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE), pp. 8–13, May 2017.

[22] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. R. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kroß, S. Spinner, C. Vögele, J. Walter, and A. Wert, "Performance-oriented devops: A research agenda," *CoRR*, vol. abs/1508.04752, 2015.

[23] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 21–30, April 2017.

[24] M. Crane and J. Lin, "An exploration of serverless architectures for information retrieval," in *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval*, ICTIR '17, (New York, NY, USA), pp. 241–244, ACM, 2017.

[25] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 159–169, April 2018.

[26] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 179–182, May 2016.

[27] E. F. Boza, C. L. Abad, M. Villavicencio, S. Quimba, and J. A. Plaza, "Reserved, on demand or serverless: Model-based simulations for cloud budget planning," in *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pp. 1–6, Oct 2017.

[28] E. van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A spec rg cloud group's vision on the performance challenges of faas cloud architectures," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, (New York, NY, USA), pp. 21–24, ACM, 2018.

[29] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 405–410, June 2017.

[30] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," *Elastic*, vol. 60, p. 80, 2016.

[31] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, MOTA '16, (New York, NY, USA), pp. 5:1–5:4, ACM, 2016.

[32] J. Spillner, "Practical tooling for serverless computing," in *Proceedings of the10th International Conference on Utility and Cloud Computing*, UCC '17, (New York, NY, USA), pp. 185–186, ACM, 2017.

[33] DZone, "The 2018 dzone guide to cloud: serverless, functions, and multicloud," 2018. https://dzone.com/guides/

cloud-serverless-functions-and-multi-cloud, obtenido el 17 de octubre del 2018.

[34] RightScale, "State of the cloud report," 2018. https://www.rightscale.com/lp/state-of-the-cloud, obtenido el 17 de octubre del 2018.

[35] D. Ocean, "Currents: A quarterly report on developer trends in the cloud," 2018. https://www.digitalocean.com/currents/june-2018/, obtenido el 17 de octubre del 2018.

[36] C. F. Foundation, "Where PaaS, containers and serverless stand in a multi-platform world," June 2018. https://www.cloudfoundry.org/multi-platform-trend-report-2018/.

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

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

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

[40] 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.

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

[42] M. Shilkov, “Cold starts in aws lambda,” Jun 2019. https://mikhail.io/serverless/coldstarts/aws/, obtenido el 10 de Julio del 2019.

[43] M. Shilkov, “When does cold start happen on aws lambda?,” Jun 2019. https://mikhail.io/serverless/coldstarts/aws/intervals/, obtenido el 10 de Julio del 2019.

[44] N. E. Andre van Hoorn, “201403-icpe-dublin,” Jun 2014. https://kieker-monitoring.atlassian.net/wiki/spaces/DOC/pages/24215578/201403-ICPE-Dublin, obtenido el 7 de Agosto del 2019.