Mejora del rendimiento de aplicaciones serverless basadas en la nube con GraalVM y Quarkus

Ron Mosquera, Bryan Alexander y Villegas Armijos, Carlos David

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Sistemas e Informática

Trabajo de titulación, previo a la obtención del título en Ingeniero en Sistemas e Informática

MSc. Campaña Ortega, Eduardo Mauricio

Sangolquí, 10 de agosto de 2022

COPYLEAKS

TESIS-QUARKUS-RON-VILLEGAS - FINAL - SIN INDICE.docx

Scanned on: 13:38 August 2, 2022 UTC

Firmado electrónicamente por:
EDUARDO MAURICIO CAMPAÑA ORTEGA

Identical Words	62
Words with Minor Changes	5
Paraphrased Words	35
Omitted Words	136

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Sistemas e Informática

Certificación

Certifico que el trabajo de titulación: "**Mejora del rendimiento de aplicaciones serverless basadas en la nube con GraalVM y Quarkus**" fue realizado por los señores **Ron Mosquera, Bryan Alexander** y **Villegas Armijos, Carlos David;** el mismo que cumple con los requisitos legales, teóricos, científicos, técnicos y metodológicos establecidos por la Universidad de las Fuerzas Armadas ESPE, además fue revisado y analizado en su totalidad por la herramienta de prevención y/o verificación de similitud de contenidos; razón por la cual me permito acreditar y autorizar para que se lo sustente públicamente.

Sangolquí, 02 de agosto del 2022

Firmado electrónicamente por:
EDUARDO MAURICIO
CAMPAÑA ORTEGA

.....

Campaña Ortega, Eduardo Mauricio

C. C. 1708856701

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Sistemas e Informática

Responsabilidad de Autoría

Nosotros, **Ron Mosquera, Bryan Alexander y Villegas Armijos, Carlos David**, con cédulas de ciudadanía n° 1725158453 y n° 1725473498, declaramos que el contenido, ideas y criterios del trabajo de titulación**: Mejora del rendimiento de aplicaciones serverless basadas en la nube con GraalVM y Quarkus** es de nuestra autoría y responsabilidad, cumpliendo con los requisitos legales, teóricos, científicos, técnicos, y metodológicos establecidos por la Universidad de las Fuerzas Armadas ESPE, respetando los derechos intelectuales de terceros y referenciando las citas bibliográficas.

Sangolquí, 02 de agosto del 2022

Ron Mosquera, Bryan Alexander
C.C. 1725158453

Villegas Armijos, Carlos David
C.C. 1725473498

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Sistemas e Informática

Autorización de Publicación

Nosotros, **Ron Mosquera, Bryan Alexander y Villegas Armijos, Carlos David**, con cédulas de ciudadanía n° 1725158453 y n° 1725473498, autorizamos a la Universidad de las Fuerzas Armadas ESPE publicar el trabajo de titulación: **Mejora del rendimiento de aplicaciones serverless basadas en la nube con GraalVM y Quarkus** en el Repositorio Institucional, cuyo contenido, ideas y criterios son de nuestra responsabilidad.

Sangolquí, 02 de agosto del 2022

Ron Mosquera, Bryan Alexander
C.C. 1725158453

Villegas Armijos, Carlos David
C.C. 1725473498

Dedicatoria

El presente trabajo lo dedico a mis padres, que me han apoyado dándome ánimos y fuerzas para seguir en todo el transcurso de la carrera, a mi tía Mery que ha sido mi segunda mamá, a mi pareja que me ha ayudado en impulsado a ser más responsable, a mis abuelitos que ya no se encuentran conmigo, pero sé que están orgullosos de que culmine una meta más, y a mi tío Juan que ha sido un gran apoyo en el transcurso de la carrera .

Dedico este trabajo a los amigos que que hice dentro de la carrera, ya que hemos estado apoyándonos los unos con los otros, también dedico este trabajo a mi mejor amigo Chiste, que ha sido un gran apoyo desde el bachillerato, y mis amigos James, Tami y Nina.

Bryan Alexander Ron Mosquera

Esta tesis es dedicada a:

A mis padres Marlon y Emérita quienes con su amor, paciencia y esfuerzo me han permitido llegar a cumplir hoy un sueño más, gracias por inculcar en mi el ejemplo de esfuerzo y valentía, de no temer las adversidades porque Dios está conmigo siempre.

A mis hermanos Josue y Carla por su cariño y apoyo incondicional, durante todo este proceso, por estar conmigo en todo momento gracias. A toda mi familia porque con sus oraciones, consejos y palabras de aliento hicieron de mi una mejor persona y de una u otra forma me acompañan en todos mis sueños y metas.

Finalmente quiero dedicar esta tesis a todas mis amigos, por apoyarme cuando más las necesito, por extender su mano en momentos difíciles y por el amor brindado cada día, de verdad mil gracias hermanitas, siempre las llevo en mi corazón.

Carlos David Villegas Armijos

Agradecimiento

Agradezco a mis padres, por apoyarme en mi carrera universitaria, por darme mi primera computadora para poder emprender el camino en esta hermosa carrera.

Agradezco a mi tía Mery, por aconsejarme y estar pendiente de mí como una segunda mamá.

Agradezco a mi pareja Priscila, por estar conmigo en el transcurso de la carrera, siendo mi apoyo incondicional.

Agradezco a mi compañero de tesis, pues hemos construido este trabajo juntos y nos hemos esforzado mucho en él.

Agradezco al Ing. Mauricio Campaña por guiarnos en el proceso de elaboración de nuestro trabajo de titulación.

Bryan Alexander Ron Mosquera

Quiero expresar mi gratitud a Dios, quien con su bendición llena siempre mi vida y a toda mi familia por estar siempre presentes.

A mis padres, ustedes han sido siempre el motor que impulsa mis sueños y esperanzas, quienes estuvieron siempre a mi lado en los días y noches más difíciles durante mis horas de estudio. Siempre han sido mis mejores guías de vida. Hoy cuando concluyo mis estudios, les dedico a ustedes este logro amados padres, como una meta más conquistada. Orgullosa de haberlos elegido como mis padres y que estén a mi lado en este momento tan importante.

Gracias por ser quienes son y por creer en mí.

Finalmente quiero expresar mi sincero agradecimiento al Ingeniero. Mauricio Campaña, principal colaborador durante todo este proceso, quien con su dirección, conocimiento, enseñanza y colaboración permitió el desarrollo de este trabajo.

Carlos David Villegas Armijos

Índice de Contenidos

Índice de Tablas

Índice de Figuras

Resumen

Existe una gran cantidad de proyectos realizados sobre una arquitectura monolítica, generando dificultades al momento de escalar o realizar actualizaciones, situación que no ocurre con la arquitectura de microservicios, ya que cada parte es un componente pequeño que se puede escalar de manera independiente.

Paralelamente, con el objetivo de mejorar el tiempo de desarrollo y escalabilidad de cada componente, surgen frameworks que permiten obtener un mejor rendimiento, como Quarkus, que es optimizado para contenedores e implementado en aplicaciones serverless.

En esta investigación se realiza la comparación del framework Quarkus con Spring Boot, este último es el más popular en el ecosistema Java, para la comparación se construyó una aplicación sobre la JVM y sobre GraalVM, para determinar qué framework brinda un mejor rendimiento al momento de recibir y atender una alta demanda. Dichas solicitudes fueron enviadas por Artillery, una biblioteca de JavaScript que permite realizar pruebas de carga y analizar los logs en AWS CloudWatch.

Al finalizar con los análisis de las métricas de AWS CloudWatch, los resultados indicaron que el framework Quarkus sobre GraalVM tiene un mejor rendimiento, ya que atiende 153526 peticiones de la prueba de carga, con una latencia de 10.4778ms, mientras que Spring sobre GraalVM atiende 78621 peticiones con una latencia de 35.7914ms.

Palabras Claves: Quarkus, Serverless, Lambdas, GraalVM, Imagen Nativa

Abstract

There are many projects based on a monolithic architecture, generating difficulties at the time of scaling or upgrades, a situation that does not occur with microservices architecture, since each part is a small component that can be scaled independently.

At the same time, to improve the development time and scalability of each component, frameworks have emerged that allow better performance, such as Quarkus, which is optimized for containers and implemented in serverless applications.

In this research a comparison of the Quarkus framework with Spring Boot, the latter is the most popular in the Java ecosystem, for the comparison an application was built on the JVM and on GraalVM, to determine which framework provides better performance when receiving and serving a high demand. These requests will be sent by Artillery, a JavaScript library that allows load testing and log analysis in AWS CloudWatch.

At the end of the AWS CloudWatch metrics analysis, the results indicated that the Quarkus framework on GraalVM has better performance, serving 153526 load test requests, with a latency of 10.4778ms, while Spring on GraalVM serves 78621 requests with a latency of 35.7914ms.

Keywords: Quarkus, Serverless, Lambdas, GraalVM, Native Image

Capítulo I

Introducción

En el capítulo I del presente trabajo de titulación se abordan los antecedentes, la problemática, justificación, objetivos (general y específicos), alcance, hipótesis y la estructura del documento.

Antecedentes

Una arquitectura monolítica es aquella en la que todos los módulos se encuentran encapsulados en un mismo paquete, debido a esto todas las funcionalidades deben de desplegarse juntas (Newman, 2019). La arquitectura monolítica es ideal para proyectos pequeños, ya que la capa de lógica del negocio y la capa de acceso a los datos se encuentran en un mismo paquete, gracias a esto se puede evitar el rediseño de la aplicación.

Uno de los grandes problemas que tiene esta arquitectura aparece cuando las aplicaciones se vuelven obsoletas y necesitan ser migradas a nuevas tecnologías que permitan su escalabilidad, el proceso de migración se vuelve complejo ya que hay que realizar la extracción de la lógica de negocio embebida, además de la dificultad de la migración de las bases de datos relacionales que afectan a las transacciones de la aplicación (Velepucha, Flores, & Torres, 2019).

Por mucho tiempo se mantuvo un estilo de arquitectura monolítica que en su momento satisfacía todos los requerimientos, los principales puntos débiles de este estilo arquitectónico es el escalado de la aplicación, los cambios constantes en el aplicativo hacen que el despliegue a producción sea ineficiente, por lo que debía pasar todo el monolítico pruebas unitarias y de integración. Para dar solución a estos inconvenientes apareció un nuevo estilo de arquitectura llamado microservicios.

La arquitectura de microservicios es un estilo arquitectónico relativamente nuevo que se está volviendo cada vez más popular en la industria. Un microservicio es un componente

pequeño que se puede desarrollar e implementar de forma independiente, es fácil de escalar y tiene una sola responsabilidad (Dragoni, y otros, 2017). Estas características hacen que los microservicios sean particularmente convenientes para la entrega continua (Thönes, 2015). Los microservicios se construyen en torno a las capacidades comerciales y proporcionan un estilo arquitectónico capaz de organizar equipos multifuncionales en torno a los servicios (Dragoni, y otros, 2017). Se supone que los microservicios ayudan a las empresas a ofrecer valor a sus clientes de forma rápida y continua.

En las últimas tres décadas (Dragoni, y otros, 2017) muchas aplicaciones se han reducido en tamaño y han explotado en número. Siguiendo una tendencia que ya se daba en la computación en red, la tecnología de computación en la nube y la práctica de desarrollo de software han permitido cargas de trabajo más diversas y tareas más focalizadas. Recientemente, los avances en la tecnología de contenedores (como Docker) han aprovechado esta tendencia de reducción para disminuir el tiempo de construcción y la sobrecarga de despliegue del software, permitiendo procesos mucho más granulares en comparación con el aprovisionamiento tradicional basado en servidores e incluso con las máquinas virtuales (VM).

Paralelamente, con el objetivo de mejorar el tiempo de desarrollo y la resistencia y escalabilidad del sistema, las prácticas de desarrollo han promovido nuevas arquitecturas de microservicios de grano fino en lugar de las tradicionales arquitecturas monolíticas y orientadas a servicios. La tecnología emergente de la computación sin servidor combina estos avances, dando un paso más hacia la computación como una utilidad basada en Internet. Pero ¿cuál es el estado de la tecnología sin servidor y hacia dónde se dirige? La computación sin servidor es un conjunto de tecnologías de computación (en la nube) que se adhieren a tres principios: (1) la lógica operativa se abstrae de los usuarios; (2) los usuarios sólo pagan por los recursos que necesitan, con una granularidad fina; (3) el modelo de computación se basa en eventos y las operaciones se escalan de forma elástica.

RM Seguridad Industrial es una empresa que se dedica a la comercialización de productos de seguridad industrial, ya sea calzado, cascos o indumentaria para la seguridad de los empleados. Ya que es una empresa pequeña, el control de inventarios se lo realiza de manera manual y con la herramienta de ofimática Excel, lo que la organización desea es una aplicación web que permita controlar la entrada y salida de productos, además que desea implementarlo en la nube, ya que no cuenta con infraestructura para poseer sus propios servidores.

Planteamiento del problema

Java es un lenguaje de programación orientado a objetos, en cada versión se agregan mejoras y nuevas características. Java destaca entre los demás lenguajes gracias a la portabilidad que ofrece añadiendo un gran aislamiento en la JVM que protege el código de todas las peculiaridades de los distintos sistemas operativos y plataformas.

Hemos sido testigos de cómo los contenedores han solucionado la sobrecarga de las máquinas virtuales, con su llegada ya no es necesario la abstracción que ofrece la JVM para ejecutar aplicaciones en diversos sistemas operativos.

Los contenedores deben ser ligeros, el tiempo de inicio de la aplicación debe ser corto, para poder ampliar o reducir el número de contenedores que permitan satisfacer las necesidades de los usuarios, la huella de memoria debe ser pequeña para reducir costos al ejecutar la aplicación en la nube. Todas esas características deseadas no se las puede implementar con Java, para dar solución a esos inconvenientes nació Quarkus, un framework de Java de alto rendimiento.

Para poder hacer el despliegue de una aplicación desarrollada en Java se necesita de una máquina virtual, específicamente de la Java Virtual Machine, en la JVM se puede compilar código de lenguajes basados en Java, como Kotlin o Scala, el problema surge cuando se quiere compilar código realizado en Python o JavaScript, ya que la JVM no permite compilar código que

no sea Java. Para solucionar este problema aparece GraalVM, que es una máquina virtual que permite compilar código escrito en diferentes lenguajes de programación como JavaScript, C++, Python, entre otros.

Quarkus con su proceso de compilación Ahead-of-Time (AOT) a través de GraalVM elimina el paso de la interpretación de JVM, de esa manera se puede ejecutar imágenes nativas de nuestra aplicación, Quarkus permite que Java sea un candidato con un excelente rendimiento como lenguaje para el desarrollo de aplicaciones con una arquitectura de microservicios.

Por otro lado, los nuevos modelos de entrega de la computación en la nube como FaaS (Función como Servicio) en donde las funcionalidades aumentan en granularidad, Java no es de las mejores opciones para desplegar funciones como un servicio en proveedores de nube como lo es Amazon Web Services, en este contexto, Quarkus podría hacer viable el desarrollo aplicaciones Java para la nube, ya que la implementación de funciones serverless es una gran opción junto a Spring.

Justificación

En los últimos años, la computación en la nube ha permitido un crecimiento continuo de muchas empresas, esto se debe a que la nube se adapta a los requerimientos cambiantes de cualquier tipo de proyecto.

Java no ha sido de las opciones más populares para implementar serverless, pero con la llegada de Quarkus, las limitaciones ocasionadas con JVM como la utilización de gran cantidad de memoria, el inicio lento de la aplicación, disminuyen drásticamente. La comparación entre Quarkus y Spring permitirá a empresas o interesados en desarrollar aplicaciones serverless elegir entre ambos frameworks.

Las aplicaciones serverless utilizan un proveedor de nube para su hosting, obtener el mejor desempeño a un mínimo costo es una prioridad para empresas interesadas en este tipo

de arquitectura. Por lo que nuestra investigación permitirá elegir entre ambos frameworks el que nos dé un mejor desempeño a un menor costo.

Gracias a que Quarkus utiliza menos memoria y tiene un tiempo de inicio corto al momento de hacer un despliegue de la aplicación, se obtendría un mínimo costo, y sumado el desempeño de las aplicaciones serverless que se desarrollen con Quarkus, se podría maximizar el ahorro de recursos y minimizar costos.

FaaS ofrece buenas razones para existir dentro del ya rico panorama de los servicios en la nube. Proporciona abstracciones de alto nivel de los elementos informáticos distribuidos, reduciendo la necesidad de que los usuarios sean expertos en sistemas distribuidos o gestionen ellos mismos complejas arquitecturas basadas en microservicios. Permite a los usuarios delegar las cuestiones operativas, lo que permite a las organizaciones centrarse en las cuestiones de negocio.

Aprovechar las características y niveles de abstracción ofrecidos por el modelo de entrega FaaS para diseñar soluciones eficientes y de fácil escalamiento utilizando el lenguaje de programación Java, este modelo nos permite centrarnos en la lógica del negocio y evitar el desperdicio de recursos de administrar la infraestructura subyacente.

Objetivos

Objetivo General

Desarrollar una aplicación que permita demostrar la mejora de rendimiento de las aplicaciones serverless basadas en la nube con GraalVM y Quarkus.

Objetivos Específicos

- Desarrollar una aplicación de control de entrada y salida de productos para la empresa RM Seguridad Industrial.
- Desplegar la infraestructura de la aplicación en el proveedor de nube AWS.

- Realizar un análisis comparativo del rendimiento entre una aplicación desarrollada en Spring y Quarkus.

Alcance

El alcance del proyecto está definido por el objetivo general, en donde se va a desarrollar una aplicación serverless para demostrar la mejora de rendimiento de una aplicación Java realizada en Quarkus, a comparación de la aplicación desarrollada en Spring, para demostrar la hipótesis se van a emplear pruebas de carga, arranque en frío, tiempo de ejecución. La aplicación se realizó para la empresa RM Seguridad Industrial con RUC 1712489259001, permite realizar el ingreso de productos nuevos a la bodega de la empresa

Hipótesis de Trabajo

Una aplicación serverless implementada con GraalVM y Quarkus tiene mejor rendimiento que una aplicación serverless implementada con Spring Boot.

Estructura del trabajo

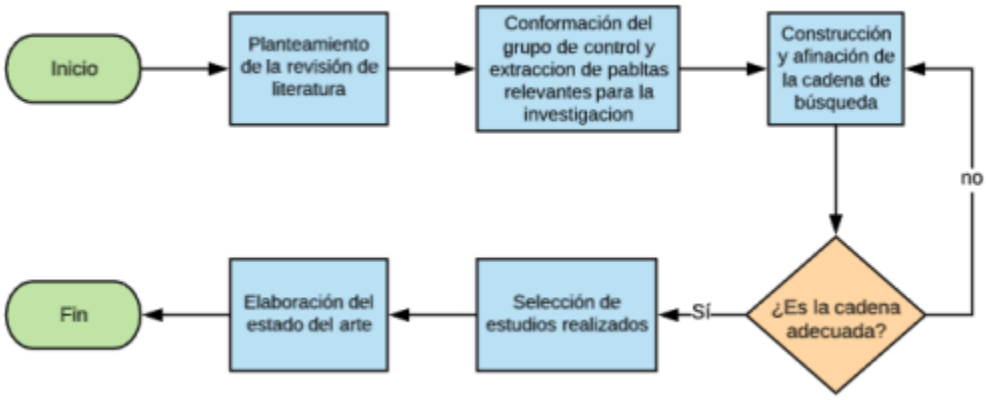
El presente trabajo de titulación está estructurado por cinco capítulos los cuales abordan las siguientes temáticas: capítulo ll es el estado de la cuestión en el que se realiza una revisión de la literatura con sus diferentes fases para llevar a cabo el estado del arte, el capítulo lll se propone una arquitectura con diferentes herramientas y el diseño de la aplicación las cuales sirven de apoyo para el desarrollo de la aplicación, el capítulo IV es la evaluación en donde se comparan los tiempos de respuesta al momento de realizar peticiones al servidor alojado en AWS, se ejecutan la pruebas y se analizan los resultados. Finalmente, el capítulo V son las conclusiones, recomendaciones y los trabajos futuros que se presentan a partir del presente trabajo.

Capítulo II

Para analizar el estado del arte acerca del uso de la investigación reproducible en la ingeniería de software se realizó una revisión de literatura preliminar basado en las guías de revisión sistemática de literatura propuesta por (Pearl Brereton, 2007). Las actividades consideradas para este proceso se muestran en la Figura 1 y se describen a continuación.

Figura 1

Revisión sistemática de literatura



Planteamiento de la revisión de literatura

En base al problema de investigación planteado fue posible: tener un contexto para realizar la búsqueda de estudios científicos, definir un objetivo de búsqueda y plantear preguntas de investigación.

El objetivo de investigación es: Analizar la factibilidad de propuestas que se encuentren enfocadas en la implementación de aplicaciones serverless en la nube, comparación de proveedores de cloud con serverless, a través de una revisión de literatura preliminar.

Conformación del grupo de control y extracción de palabras claves relevantes para la investigación

Para la revisión de literatura se han definido aquellos artículos considerados relevantes para la investigación, son aquellos que toman aspectos generales acerca de mejoras de rendimiento con GraalVM, Quarkus y eficiencia de costos en proveedores de nube para el despliegue de Funciones como Servicio.

El grupo de control seleccionado que se indica en la Tabla 2 se obtuvo luego de una serie de validaciones sobre los estudios propuestos.

Tabla 1

Grupo de Control

Título	Cita	Palabras Clave
Uma Análise Comparativa De Desempenho Entre Diferentes Tecnologias De Execução De Aplicações Web Do Lado Do Servidor	(Almeida, 2020)	GraalVM. Java. Docker. Tempo de resposta. Spring Boot. Quarkus.
Beginning Quarkus Framework	(Koleoso, 2020)	Quarkus, Beginning,Java EE, Jakarta EE, practical, projects, code, source, pragmatic, in-depth, tutorial, learn, Java, Kubernetes, IBM, Red Hat, JBoss, Spring, cloud, native
Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures	(Villamizar, y otros, 2017)	Cloud computing Microservices Service-oriented architectures Scalable applications Software engineering Software architecture Microservice architecture Serverless architectures AWS Lambda Amazon Web Services
The SPEC-RG reference	(van Eyk, y otros, 2019)	Reference Architecture,

architecture for faas: From microservices and containers to serverless platforms		Serverless Computing, Function-as-a-Service, FaaS, Microservices, Containers
Cost efficiency under mixed serverless and serverful deployments	(Reuter, Back, & Andrikopoulos, 2020)	Cost efficiency, mixed serverless deployments, serverful deployments, serverless computing paradigm, pay-per-use billing model, application stack, regular service selection problem, nonFaaS deployments, hybrid deployment model, simultaneous FaaS deployment

Construcción de la cadena de búsqueda

Con las palabras clave que fueron obtenidas de los artículos científicos del grupo de control, además de varias pruebas con varias combinaciones de las mismas palabras, se logró conformar las siguientes cadenas de búsqueda:

((QUARKUS) AND (CLOUD COMPUTING)) AND (SERVERLESS APPLICATION).

La primera enfocada al framework de desarrollo Quarkus y a la nube, y la siguiente enfocada en aplicaciones serverless.

Selección de estudios

Las cadenas de búsqueda construidas fueron aplicadas en la base digital Science Direct e IEEE Xplore, donde se obtuvieron alrededor de 250 resultados relacionados, los cuales fueron sometidos a filtros para obtener una cantidad manejable de documentos. Los filtros que se aplicaron fueron los siguientes:

- Estudios del tipo: technical report, conference paper y journal paper.

- Vigencia: Estudios realizados a partir del año 2014. Año elegido debido al rápido avance de la tecnología, siendo necesaria la actualidad de los estudios.
- Fuente de revisión de literatura: IEEE Xplore, Science Direct, Springer.

Luego de aplicar estos filtros, la base digital nos dejó alrededor de 28 artículos científicos clasificados como relevantes, los mismos que fueron revisados por los investigadores, para así elegir 4 estudios primarios que conforman la base para realizar el estudio del estado del arte, estos estudios se muestran en la Tabla 2.

Tabla 2

Estudios primarios seleccionados

Código	Título	Cita
EP1	A. Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus	(Šipek, Muharemagić, Mihaljević, & Radovan, 2020)
EP2	Uma análise comparativa de desempenho entre diferentes tecnologias de execução de aplicações web do lado do servidor.	(Almeida, 2020)
EP3	Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM.	(Šipek, Mihaljević, & Radovan, 2019)
EP4	Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures.	(Villamizar, y otros, 2017)

Elaboración del estado del arte

EP1 (Sipek et al. 2020): Šipek, M., Muharemagić, D., Mihaljević, B., & Radovan, A. Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus. In 2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO) (pp. 1746-1751). IEEE.

Este estudio menciona que los requisitos de las soluciones se han vuelto más complejos, para mitigar esta complejidad las soluciones se han migrado a la nube, las soluciones monolíticas

se ejecutan en costosos servidores físicos los cuales son difíciles gestionar sus recursos, distribución de la potencia de cálculo y la estabilidad.

Las aplicaciones en la nube se despliegan en un cluster Docker o en contenedores colocados en Kubernetes, además se analiza el rendimiento de imágenes nativas creadas con GraalVM para el framework nativo de Java para Kubernetes, también se realizan diferentes pruebas en varios escenarios.

EP2 (Almeida 2020): Uma análise comparativa de desempenho entre diferentes tecnologias de execução de aplicações web do lado do servidor.

Este trabajo el rendimiento de las aplicaciones nativas utilizando Spring Boot y Quarkus, las imágenes nativas fueron creada con GraalVM, se hizo una comparativa entre Quarkus y Spring donde se concluye que la aplicación construida con Quarkus mejora su rendimiento mientras que con Spring no hubo cambio en su rendimiento.

EP3 (Sipek et al. 2019): Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM. In 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) (pp. 1671-1676). IEEE.

En este artículo se describen las características de la máquina virtual políglota GraalVM, además describe cómo manejar la interoperabilidad entre diferentes lenguajes dentro de la máquina virtual GraalVM sin afectar el rendimiento. Este artículo también compara GraalVM con algunos de los competidores más especializados, y presenta resultados probados en un entorno académico.

EP4 (Villamizar et al. 2017): Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. Service Oriented Computing and Applications, 11(2), 233-247.

En este artículo se destaca los costes generados al adoptar la arquitectura de microservicios además realizan la comparación de costes de una aplicación web desarrollada y

desplegada utilizando los mismos escenarios de escabilidad con tres enfoques diferentes: arquitectura monolítica, una arquitectura de microservicios operada por el cliente de la nube, y una arquitectura de microservicios operada por el proveedor de la nube.

Es muy importante destacar que al adoptar AWS Lambda los costos de infraestructura bajan en un 70% o más de acuerdo con los resultados mostrados en la tabla de resultados, también describe los retos a los que nos enfrentamos durante la implementación y el despliegue de las aplicaciones de microservicios.

Discusión General del estado del Arte

En EP1 y EP2 hace el análisis de la imagen nativa en un entorno de Kubernetes y diferentes escenarios, pero no evalúa el rendimiento en el proveedor de nube AWS, EP3 analiza el rendimiento de la máquina virtual GraalVM este articulo no desarrolla un caso de uso en AWS Lambda.

Finalmente, EP4 muestra el rendimiento en diferentes escenarios con algunos tipos de arquitectura, las tecnologías utilizadas son: Scala, NodeJs y Angular; este estudio no realiza pruebas entre los frameworks Quarkus y Spring Boot.

Características del estado del arte

Dentro de la revisión de literatura preliminar realizada, la mayoría de los autores proponen soluciones para minimizar costos en varios proveedores en la nube, además se menciona el incremento de rendimiento al crear una imagen nativa con GraalVM.

Por otro lado, se muestran estudios comparativos que muestran métricas en diferentes escenarios que permite observar mejoras relevantes entre las diferentes tecnologías y proveedores de nube.

Capítulo III

Desarrollo

En la presente sección se detalla la arquitectura propuesta, las herramientas propuestas en la arquitectura y el propósito de estas dentro de la aplicación, las herramientas empleadas para la interacción con el usuario y las herramientas que se emplearán para realizar las pruebas.

Arquitectura

Se propone una arquitectura serverless en la nube como se muestra en la figura 1, la aplicación está desarrollada empleando el conjunto de librerías de React, que es una librería enfocada al desarrollo web, esto permite que la aplicación se pueda utilizar en varios dispositivos que tengan conexión a internet.

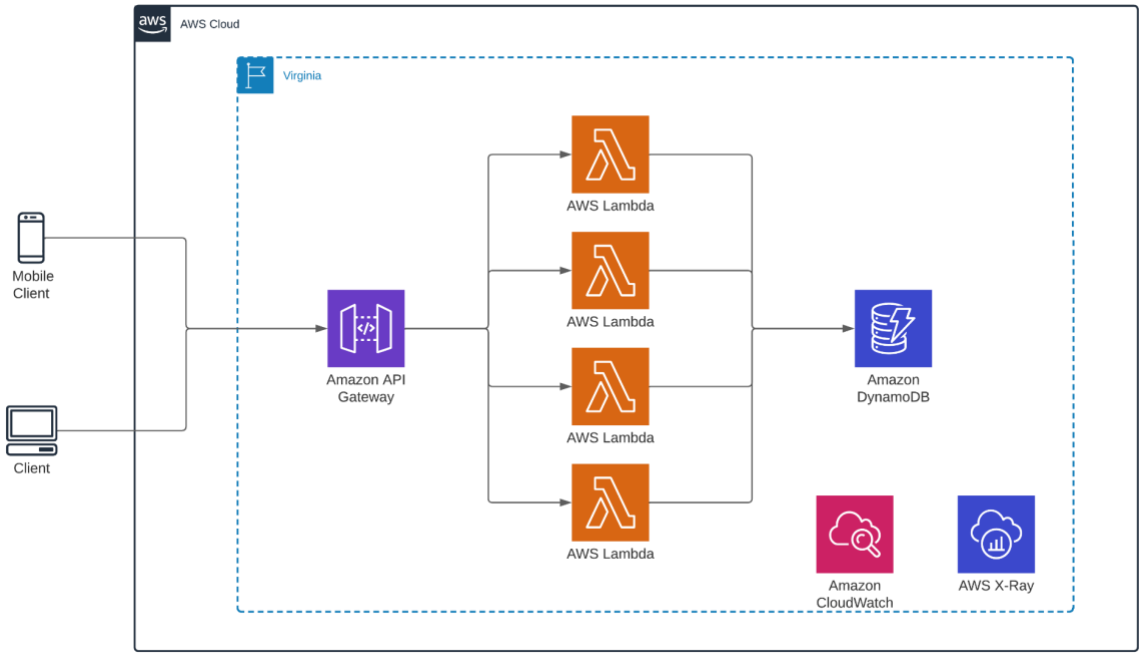
El desarrollo frontend se realiza con NextJS, Material UI y Styled Components, con la finalidad de ofrecer una interfaz rápida y fácil de usar para el usuario, además de poder reutilizar componentes en un futuro para escalar la aplicación.

El desarrollo backend se realiza con dos frameworks de Java: Quarkus y Spring Boot, con dichos frameworks se realizarán funciones lambdas que se conectan con DynamoDB y que entregan un endpoint para poder consumir el servicio REST mediante un API.

Las pruebas de carga y de rendimientos de las lambdas se realizan con AWS CloudWatch y con Artillery, se envían peticiones durante 10 minutos y se evalúan los resultados obtenidos en AWS CloudWatch.

Figura 2

Arquitectura de la aplicación



Microservicios

Los microservicios son una forma de diseñar software, en la cual se utilizan un conjunto de pequeños servicios independientes que se integran en un sistema o aplicación mediante protocolos livianos de comunicación, estos servicios se construyen y muestran de forma individual (Molina, Loja, Zea, & Loaiza, 2016).

Las características principales de los microservicios comprenden:

- Se consideran altamente mantenibles y testeables
- Son altamente desacoplados
- Son muy independientes
- Se mantienen organizados alrededor de las capacidades de negocio

Patrones de diseño de los Microservicios

De acuerdo con el autor (Barrios Contreras, 2018), los microservicios presentan los siguientes patrones de diseño:

1) Descomposición

- Descomposición por capacidades de negocio: En este se define los servicios correspondientes a las capacidades de negocio.
- Descomposición por subdominio: En este se definen los servicios correspondientes a subdominios del Domain-Driven Design.
- Servicio autocontenido: En este se diseñan los servicios con el objetivo de manejar peticiones síncronas, es decir, sin esperar a que otro servicio responda.
- Servicio por equipo: En este, se diseña un único servicio por equipo, con el objetivo de facilitar la autonomía entre equipos.

2) Refactorización en microservicios (Refactoring to microservices)

   En este patrón se destaca:

- La aplicación estranguladora (Strangler Application), la cual se encarga del desarrollo incremental en el que las funcionalidades de la aplicación se convierten en servicios desacoplados, con lo cual, se reduce el tamaño de la aplicación con el tiempo.
- La capa anticorrupción, la cual define la función de traductor entre dos modelos de dominio.

3) Gestión de datos (Data management)

   La gestión de los datos se realiza por alguno(s) de los siguientes tipos:

- Base de datos por servicio: En este tipo de gestión de datos, cada servicio tiene su propia base de datos.
- Base de datos compartida: En este tipo de gestión de datos, los servicios comparten una misma base de datos.

- Saga: En este tipo, se realiza una secuencia de las transacciones locales, con el objetivo de mantener la consistencia de los datos a través de los servicios.
- Composición de una API: En este tipo, se implementan las consultas de invocación de los servicios que poseen los datos y se realiza una combinación en la memoria.
- CQRS (Command-Query Responsibility Segregation): En este tipo, se realiza la separación de la lógica de comandos (lectura y esritura) para realizar consultas a la base de datos de manera eficiente.
- Evento de dominio: En este tipo, se publica un evento cuando un dato cambie.
- Event Sourcing: Se produce cuando persisten los datos agregados como una secuencia de eventos.
- Monitorización: En este tipo, se emplean herramientas o librerías con el objetivo de mantener un sistema de seguimiento de los datos que están involucrados en los distintos servicios.

4) Mensajería transaccional (Transactional messaging)

- Bandeja de salida transaccional (transactional outbox): los mensajes y/o eventos que son generados por la base de datos se almacenan en una bandeja de salida (outbox), la cual, al ser leída, almacena los mensajes y/o eventos que son enviados al broker.
- Seguimiento de registro de transacciones (transaction log tailing): Es el que permite enviar al broker los mensajes y/o eventos almacenados en la bandeja de salida (outbox).
- Editor de encuestas (polling publisher): Es el que permite almacenar los mensajes y/o eventos que se generan por la base de datos en la bandeja de salida (outbox).

5) Pruebas (Testing)

- Prueba impulsada por el consumidor: Este conjunto de pruebas se realiza por los desarrolladores desde un servicio que provee a otro que lo consume.

- Prueba del lado del consumidor: Es un conjunto de pruebas para un consumidor de un servicio, en la cual se verifica que la recepción del servicio se realiza de manera satisfactoria.
- Prueba de componentes de servicio: Es un conjunto de pruebas que verifican un servicio de forma individual, utilizando cuentas o casos de prueba para cualquier servicio que invoque.

6) Patrones de despliegue (Deployment patterns)

Este patrón de diseño está orientado al desarrollo del servicio, por lo cual se incluyen los siguientes:

- Se debe considerar múltiples instancias de un servicio por host.
- Cada servicio debe constar con una instancia por host.
- Se debe contar con una instancia de servicio por cada VM (Máquina Virtual).
- Se debe considerar una instancia de servicio por cada contenedor.
- Despliegue sin servidor: En caso de desplegar un servicio usando una plataforma de despliegue sin servidores.
- Plataforma de despliegue de servicios: En caso de desplegar los servicios utilizando una plataforma de despliegue que sea altamente automatizada que proporcione una abstracción de servicios.

7) Patrones de comunicación (Communication patterns)

- Procedimiento de invocación remota: Se debe utilizar un protocolo basado en el RPI (Remote Procedure Invocation) para realizar la comunicación entre servicios.
- Mensajería: Se utilizan mensajes, generalmente asíncronos, para lograr la comunicación entre servicios.
- Protocolo específico de dominio.

- Consumidor idempotente: Es cuando se garantiza que los consumidores de mensajes puedan ser invocados o llamados varias veces mediante un mismo mensaje.

8) API externa (External API)
    - API gateway: Es un servicio que proporciona a cada cliente una interfaz integrada de servicios.
    - Backend a Frontend: Se trata de una pasarela API separada dependiendo del tipo de cliente.
9) Fiabilidad (Reliability)
    - Circuit breaker: Es cuando se invoca un servicio remoto a través de un proxy, el cual falla inmediatamente cuando se supera el máximo de llamadas incorrectas.
10) Seguridad (Security)
    - Token de acceso: Es un token que almacena de forma segura la información sobre el usuario, la cual se encuentra involucrada entre los servicios.

Observabilidad (Observability)

- Agregación de logs.
- Métricas de la aplicación: Es el proceso que monitoriza el código de un servicio, con el objetivo de reunir estadísticas sobre las operaciones del servicio requerido.
- Registro de auditoría: Es el proceso que se encarga de registrar la actividad de los usuarios en una base de datos
- Rastreo distribuido: Es el proceso que se encarga de monitorizar la petición de un servicio y asigna a cada solicitud un identificador único que se comunica y transfiere entre los servicios.
- Seguimiento de excepciones: Se realiza cuando se informa de todas las excepciones y/o incidencias a un servicio, y por consiguiente, se notifica a quien corresponda.

- Registro de cambios y despliegues.

Arquitectura de los Microservicios

La arquitectura de microservicios tiene un impacto relevante entre la aplicación del servicio y la base de datos; debido a que cada servicio tiene un esquema de datos propio, con lo cual, se puede elegir el modelo más conveniente acorde con la necesidad del caso (Cita 3 - Arquitectura de microservicios).

Entre las principales características se encuentran:

- Enfocarse en una cosa a la vez.
- Se organiza en torno a la lógica del negocio.
- Tiene una alta autonomía entre cada servicio.
- Presenta productos, no proyectos.
- Puede contener una gran diversidad de tecnologías.

Serverless

Serverless es un término que significa sin servidor, y hace referencia a una solución que permite desarrollar y ejecutar aplicaciones de una manera rápida y con un costo menor, debido a que no es necesario proveer una infraestructura, sin embargo, evidentemente, detrás de todo esto existen servidores que permiten correr las aplicaciones y/o sus correspondientes datos/servicios, y dejando la administración y demás recursos necesarios al servidor al proveedor de la nube.

Entre las ventajas de usar serverless, se puede enfatizar en las siguientes:

- No se requiere administrar la infraestructura: No se necesita proveer ni administrar servidores, debido que esto se convierte en la responsabilidad del proveedor de nube.
- Escalabilidad: Los servicios en la nube generalmente tienen la característica de ser flexibles, los cuales permiten escalar de acuerdo a demanda; en lo que respecta a

Serverless, la aplicación escala de forma automática, a la vez que mantiene un equilibrio en cuanto a recursos, adaptándose rápidamente a las necesidades que se requieran.

- Ahorro de costos: Esta parte se justifica debido a que se paga por el tiempo que dura la ejecución, en lugar de una instancia.
- Alta disponibilidad y tolerancia a fallas: Los servicios sin servidor ofrecen alta disponibilidad y tolerancia a fallas de forma predeterminada, lo cual hace posible la replicación de datos sin realizar ningún proceso.

Framework

Un framework puede definirse como un armazón, el cual funciona como una estructura que contiene técnicas y/o recursos requeridos para determinados proyectos.

En lo concerniente a web, se puede considerar como una aplicación genérica incompleta, es decir, como un esqueleto, el cual tiene la ventaja de ser configurable, teniendo en algunos casos directrices a nivel de arquitectura, lo cual ofrece al desarrollador un conjunto de herramientas que permiten agilitar el proceso de desarrollar una aplicación web concreta, permitiendo reutilizar componentes.

API Rest

API REST es un tipo de API (Interfaz de programación de aplicaciones) que se basa en REST, la cual no es un estándar, pero se basa en algunos, tales como HTTP (Protocolo de transferencia de hipertexto) y URL (Uniform Resource Locator), en asociación con esto, se encuentran tecnologías como JSON (JavaScript Object Notation), el cual es un formato ligero de intercambio de datos de cliente a servidor y viceversa.

Herramientas

Plataformas de desarrollo

WebStorm

WebStorm es un entorno de desarrollo integrado para JavaScript y tecnologías relacionadas al lenguaje. Permite la integración de plugins que facilitan el desarrollo de aplicaciones, además de contar con una terminal integrada (JetBrains, 2022).

IntelliJ IDEA

IntelliJ IDEA es un entorno de desarrollo integrado para Java y tecnologías relacionadas al lenguaje, como Maven o Scala. Cuenta con herramientas que se comunican con SDKs y JDKs facilitando el desarrollo de aplicaciones (JetBrains, 2022).

AWS CLI

La interfaz de línea de comandos de AWS (AWS CLI) es una herramienta de código abierto que le permite interactuar con los servicios de AWS mediante comandos de shell de línea de comandos. Con una configuración mínima, utilizando la CLI de AWS, los comandos (shell de Linux, comandos de la línea de comandos de Windows, control remoto (AWS) que implementan una funcionalidad equivalente a la proporcionada por la consola de administración de AWS basada en navegador desde un símbolo del sistema del programa de terminal (Amazon Web Services, 2022).

Herramientas para el Frontend de la aplicación

Javascript

JavaScript (JS) es un lenguaje de programación ligero, interpretado o compilado justo a tiempo con funciones de primera clase. Aunque es más conocido como lenguaje de scripting

para páginas web, muchos entornos distintos al de los navegadores también lo utilizan, como Node.js, Apache CouchDB y Adobe Acrobat. JavaScript es un lenguaje dinámico basado en prototipos, multiparadigma y de un solo hilo, que admite estilos orientados a objetos, imperativos y declarativos (por ejemplo, la programación funcional) (Mozilla Developer Network, 2022).

React

React, es una biblioteca de JavaScript gratuita y de código abierto. Sirve para construir interfaces de usuario combinando secciones de código (componentes) en sitios web completos. Originalmente construida por Facebook, Meta y la comunidad de código abierto la mantienen ahora (Facebook Open Source, 2022).

React.js está construido usando JSX - Una combinación de JavaScript y XML. Los elementos se crean con JSX y luego se utiliza JavaScript para representarlos en el sitio. Aunque React tiene una curva de aprendizaje pronunciada para un desarrollador junior, se está convirtiendo rápidamente en una de las bibliotecas de JavaScript más populares y demandadas.

NextJS

Next.js es un framework de JavaScript que permite construir sitios web estáticos rápidos y extremadamente fáciles de usar, así como aplicaciones web utilizando React, ya que construye aplicaciones híbridas que contienen tanto páginas renderizadas del lado del servidor como páginas generadas estáticamente (NextJS, 2022).

Material UI

Material-UI es un framework CSS que proporciona componentes de React y sigue el Material Design de Google lanzado en 2014. Material-UI permite utilizar diferentes componentes para crear una interfaz de usuario. Google utiliza Material Design para garantizar que,

independientemente de cómo interactúen los usuarios con los productos que utilizan, tendrán una experiencia consistente. Material Design incluye directrices para la tipografía, la cuadrícula, el espacio, la escala, el color, las imágenes, entre otros elementos, además permite a los diseñadores construir diseños deliberados con jerarquía, significado y un enfoque en los resultados (Material UI, 2022).

Styled Components

Styled-components es una biblioteca construida para desarrolladores de React y React Native. Permite utilizar estilos a nivel de componente en las aplicaciones, combinando JavaScript y CSS utilizando una técnica llamada CSS-in-JS. Styled components se basan en literales de plantillas etiquetadas, lo que significa que el código CSS real se escribe entre los signos de puntuación cuando se aplica el estilo a los componentes. Esto ofrece a los desarrolladores la flexibilidad de reutilizar el código CSS de un proyecto a otro, dando la facilidad de no asignar los componentes creados a estilos CSS externos (Styled Components, 2022).

Herramientas para el Backend de la aplicación

Java

Java es un lenguaje de programación de propósito general, orientado a objetos y de alto nivel que fue desarrollado originalmente por James Gosling, un informático canadiense, en lo que entonces era Sun Microsystems, en el estado de California, EE. UU, en 1991.

Las ventajas de Java son el costo gratuito, la simplicidad, la independencia de la plataforma, la orientación a objetos, la seguridad y los subprocesos múltiples, la creación de redes, desarrollo móvil y desarrollo empresarial (Xiao, 2019).

Spring

Spring fue creado en 2003 por Rod Johnson, autor de J2EE Development without EJB (Wrox Publishing, 2004). Spring Framework fue la respuesta a toda la complejidad que tenían las especificaciones de J2EE en ese momento. Hoy en día, ha mejorado, pero es necesario contar con toda una infraestructura para ejecutar ciertos aspectos del ecosistema (Gutierrez, 2018).

Podemos decir que Spring es una tecnología complementaria a Java EE. El Spring Framework integra varias tecnologías, como la API de Servlet, la API de WebSocket, las utilidades de concurrencia, la API de JSON Binding, la validación de beans, JPA, JMS y JTA/JCA.

Quarkus

Quarkus es un framework basado en Java y nativo de Kubernetes hecho para máquinas virtuales Java y compilación nativa. Optimiza Java específicamente para los contenedores y permite que se convierta en una plataforma eficaz para la nube con servicios serverless y Kubernetes. Está diseñado para trabajar con estándares, frameworks y bibliotecas populares de Java.

Quarkus incluye un marco de extensión necesario para configurar, arrancar e integrar un marco en su aplicación ofreciendo herramientas para añadir extensiones. Además, Quarkus también proporciona la información correcta a GraalVM para la compilación nativa de la aplicación (Quarkus, 2022).

Native Image es una tecnología que permite compilar por adelantado el código Java en un ejecutable independiente, denominado imagen nativa. Este ejecutable incluye las clases de la aplicación, las clases de sus dependencias, las clases de la biblioteca de tiempo de ejecución y el código nativo enlazado estáticamente del JDK. No se ejecuta en la VM de Java, sino que incluye los componentes necesarios, como la gestión de la memoria, la programación de hilos, etc., de un sistema de ejecución diferente, llamado "Substrate VM", que son los componentes de

tiempo de ejecución. El programa resultante tiene un tiempo de inicio más rápido y una menor sobrecarga de memoria en tiempo de ejecución en comparación con una JVM (Quarkus, 2022).

GraalVM

GraalVM es una máquina virtual Java (JVM) y un kit de desarrollo Java (JDK) creados por Oracle, brindando un tiempo de ejecución de alto rendimiento que proporciona mejoras en el rendimiento y la eficiencia de las aplicaciones.

Los objetivos de GraalVM son: escribir un compilador más rápido y fácil de mantener, mejorar el rendimiento de los lenguajes que se ejecutan en la JVM, reducir los tiempos de inicio de las aplicaciones, integrar el soporte multilingüe en el ecosistema Java, así como proporcionar un conjunto de herramientas de programación para ello (GraalVM, 2022).

GraalVM añade un compilador optimizado al JDK, que proporciona optimizaciones de rendimiento para lenguajes individuales e interoperabilidad para aplicaciones políglotas. Además de soportar código Java, GraalVM también soporta lenguajes de programación adicionales, incluyendo Scala, Kotlin, Groovy, Clojure, R, Python, JavaScript, Ruby.

DynamoDB

DynamoDB es una base de datos NoSQL que ofrece Amazon Web Services (AWS), brinda un rendimiento fiable y escalable, ya que permite el acceso mediante una API, brindando patrones de consulta avanzados. Se recomienda usar DynamoDB en aplicaciones con grandes cantidades de datos y requisitos estrictos de latencia, evitando que las consultas sean lentas debido a la cantidad de operaciones SQL. Se puede emplear DynamoDB en aplicaciones sin servidor con AWS Lambda ya que este servicio proporciona una computación autoescalable. Se puede acceder a DynamoDB a través de una API HTTP y realiza la autenticación y la autorización

a través de los roles, lo que la convierte en una opción perfecta para crear aplicaciones serverless (Amazon Web Services, 2022).

AWS Lambda

AWS Lambda es un recurso de gestión de programación de Amazon Web Services que permite a los usuarios ejecutar código sin necesidad de aprovisionar las aplicaciones. AWS brinda una solución escalable y factura en base a la demanda de recursos.

AWS Lambda garantiza alta disponibilidad, antes de la aparición de servicios ágiles en la nube, las empresas debían tener la infraestructura necesaria para cada ejecución de una aplicación o programa. Tenían que asegurarse de que las demandas de entrada y memoria no desbordaran lo que sus sistemas internos podían manejar, pero gracias a AWS Lambda y las herramientas relacionadas, la propia plataforma se expande y contrae para manejar la demanda en tiempo real de las aplicaciones y los programas.

AWS Lambda permite ejecutar programas en muchos lenguajes diferentes, como Java, C# y Python, así como muchos tipos diferentes de aplicaciones o servicios de back-end. Es un excelente ejemplo de cómo los sistemas tecnológicos actuales quitan la responsabilidad del aprovisionamiento a un administrador de programas (Amazon Web Services, 2022).

AWS CloudWatch

Amazon CloudWatch es un servicio de Amazon Web Services que proporciona monitorización para los recursos de AWS y las aplicaciones de los clientes que se ejecutan en la infraestructura de Amazon. CloudWatch permite monitorizar en tiempo real los recursos de AWS. El servicio recopila y proporciona automáticamente métricas de utilización de la CPU, latencia y recuento de solicitudes. Los usuarios también pueden estipular las métricas adicionales que

deben supervisarse, como el uso de la memoria, los volúmenes de transacciones o las tasas de error (Amazon Web Services, 2022).

Para realizar búsquedas dentro de los logs de Amazon CloudWatch se emplea CloudWatch Logs Insights, con esta herramienta se pueden obtener datos que ayuden a responder de forma eficaz y eficiente los problemas operativos, debido a que CloudWatch Logs Insights proporciona un lenguaje de consulta especialmente diseñado con comandos, similar a SQL.

CRITERIOS DE SELECCIÓN

Los criterios utilizados para seleccionar las herramientas descritas anteriormente son: popularidad, rendimiento, servicios de AWS orientados a serverless.

Se utilizó AWS porque es líder en el cuadrante mágico de Garden, DynamoDB porque es una base de datos que se paga por el uso bajo demanda y altamente escalable, AWS API Gateway porque se integra con AWS Lambda, AWS CloudWatch debido a que centraliza los registros de los servicios de AWS.

Capítulo IV

Evaluación y Resultados

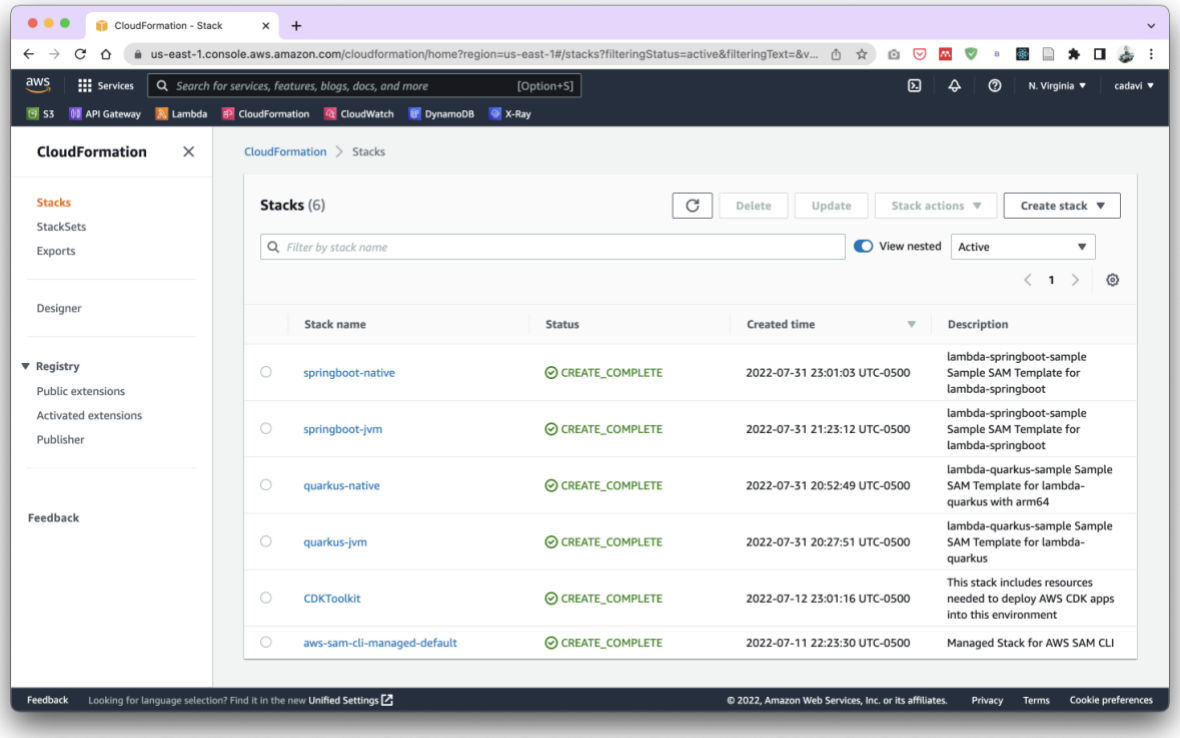
En este capítulo se define la metodología que permite determinar cuál framework proporciona un mejor rendimiento sobre el proveedor de nube AWS.

Existen varios métodos para comparar frameworks, uno de ellos son las pruebas de carga que permiten cuantificar el rendimiento de la aplicación en un entorno de alta demanda. Cada framework es sometido a las mismas condiciones a través del moderno kit de herramientas Artillery.

Artillery va realizar la prueba de carga en las aplicaciones listadas en la figura 3, se tiene 4 aplicaciones, Quarkus y Spring Boot con JVM e imagen nativa.

Figura 3

Lista de Aplicaciones en Cloud Formation



Prueba de carga

Artillery permite realizar pruebas de carga HTTP y HTTPS de forma declarativa por medio de un archivo *.yml*, este archivo permite definir escenarios con sus respectivos flujos, además a cada escenario se le puede asignar una probabilidad de ocurrencia.

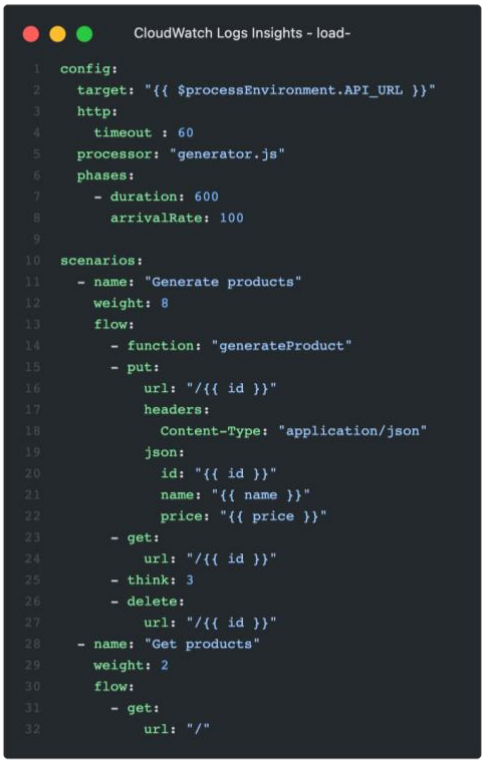
Por otro lado, permite consumir funciones generadoras para la carga dinámica de datos, crear usuarios virtuales y la duración de la prueba de carga.

La cuota de rendimiento predeterminada por tabla es de 40000 unidades de solicitud de lectura y 40000 unidades de solicitud de escritura para DynamoDB bajo demanda según la documentación de AWS, además se consumen 20000 unidades adicionales con el fin de aumentar el rendimiento y probar la hipótesis planteada.

Para el análisis comparativo entre ambos frameworks se realizaron 100 peticiones/segundo durante 10 minutos a nuestros puntos finales de la API para alcanzar las 60000 unidades de lectura y escritura en DynamoDB. Las peticiones son consumidas por las lambdas a través de la API Gateway de AWS.

Figura 4

Archivo .yml para prueba de carga



AWS CloudWatch Log Insights

Los logs generados por las peticiones HTTP son almacenados en el servicio de AWS CloudWatch, estos logs son procesados con CloudWatch Logs Insights para obtener información sobre el consumo de CPU, RAM, latencia, costo por petición e inicio en frío.

La latencia es una métrica que nos permite cuantificar la capacidad de respuesta de nuestra aplicación a la demanda generada por los usuarios. Se utilizan percentiles para evitar que datos atípicos distorsionen los resultados de la latencia como es en el caso de la latencia promedio.

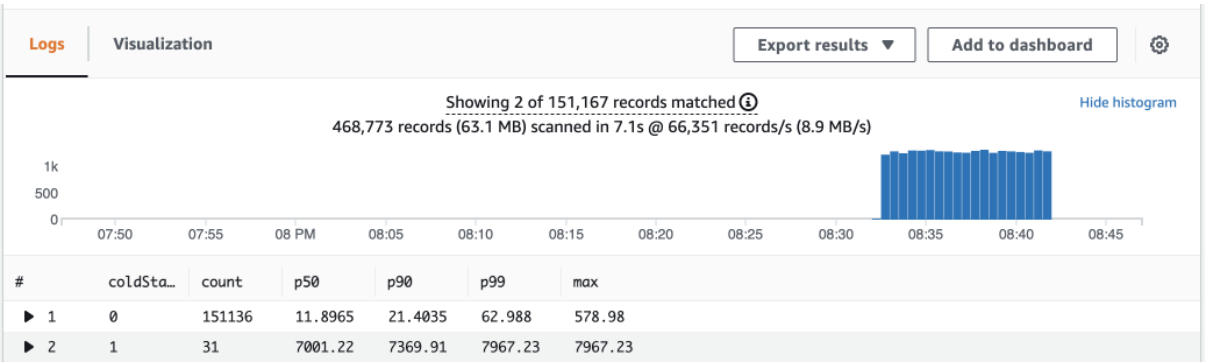
Para obtener los datos para el análisis, se ejecutó la siguiente consulta en AWS CloudWatch, la consulta permite obtener el tiempo de inicio de la aplicación, la cantidad de solicitudes atendidas, la latencia por percentiles y el tiempo máximo de cuando existe inicio en frio.

```
filter @type="REPORT
| fields greatest(@initDuration, 0) + @duration as duration, ispresent(@initDuration) as coldStart
| stats count(*) as count, pct(duration, 50) as p50, pct(duration, 90) as p90, pct(duration, 99) as
p99, max(duration) as max by coldStart
```

En la figura 5 muestra la ejecución de la consulta anterior para obtener la latencia por percentiles sobre las lambdas de la aplicación *quarkus-jvm*, el resultado muestra los inicios en frío, percentil 50, 90, 99 y la latencia máxima, la búsqueda agrupa en percentiles la latencia de las solicitudes hacia la aplicación *quarkus-jvm*, el percentil p50 indica que el 50% de las solicitudes tienen una latencia menor o igual a 7 segundos cuando la lambda tiene un arranque en frío, el percentil p90 indica que el 90% de las solicitudes tienen una latencia de 7.369 segundos cuando la lambda tiene arranque en frío, por último el percentil p99 indica que el 99% de las solicitudes tienen una latencia de 7.967 segundos cuando la lambda tiene un arranque en frío.

Figura 5

Resultado de consulta de la aplicación quarkus-jvm en AWS CloudWatch



El arranque en frío es el tiempo utilizado para la importación de bibliotecas y dependencias fuera del controlador de la lambda, establecer configuraciones e inicializar conexiones a otros servicios. Por lo tanto, cuando una lambda tiene arranque en frío la latencia de las solicitudes aumenta.

Cuando la lambda ya se encuentra inicializada la latencia disminuye, en consecuencia, el rendimiento aumenta. Por esta razón, es importante disminuir los tiempos generados por los arranques en frío.

En la figura 5, se observa que cuando no existe arranque en frío, la latencia disminuye de forma considerada.

Figura 6

Resultado de query de la aplicación spring-jvm en AWS CloudWatch

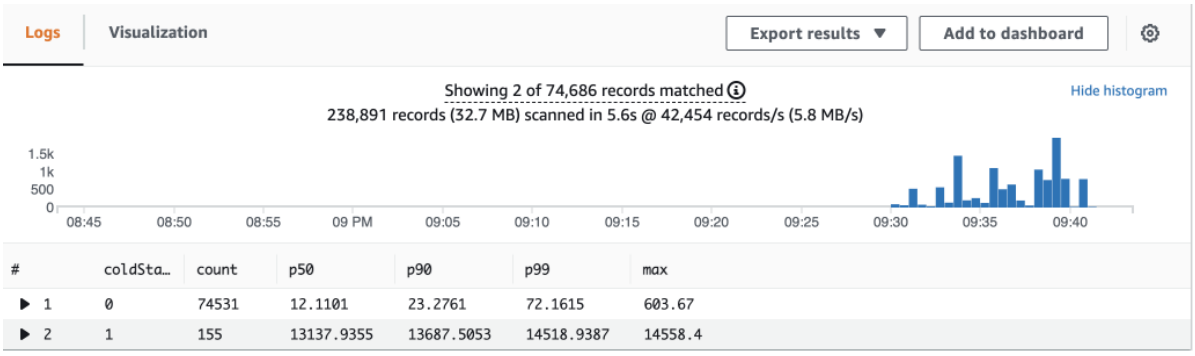


Figura 7

Resultado de query de la aplicación quarkus-graalvm en AWS CloudWatch

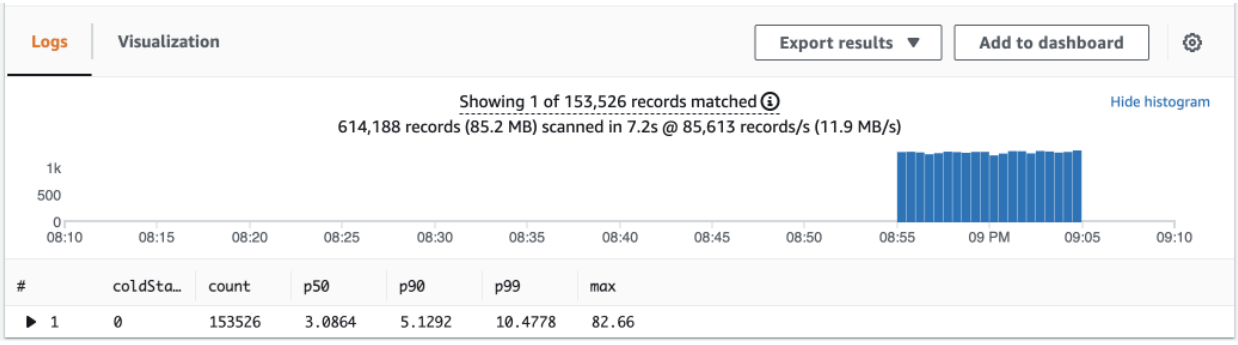
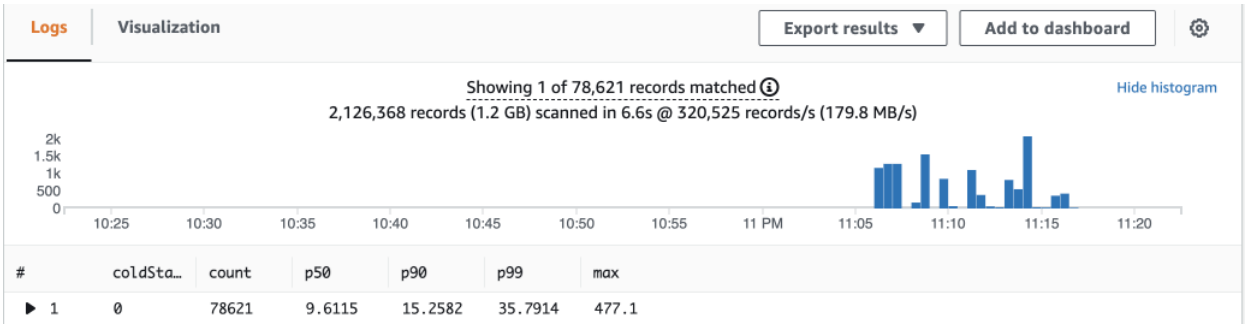


Figura 8

Resultado de query de la aplicación spring-graalvm en AWS CloudWatch



En las figuras 6, 7, 8 el análisis y la interpretación de los datos es similar al de la figura 5, lo que sobresale, son tiempos de latencia inferiores de Quarkus sobre Spring Boot.

Resultados

En las tablas 3, 4 se resume toda la información de los percentiles para las categorías arranque en frío y arranque caliente para ambos frameworks: Quarkus y Spring Boot.

En la tabla 5, se toma el percentil 99 para nuestro análisis, en el cual se puede destacar al framework Quarkus con GraalVM, debido a que, en un intervalo de tiempo de 10 minutos, atendió 153526 solicitudes con una latencia 10.4778 ms cuando existe arranque en caliente, superando al framework Spring Boot con GraalVM, que atendió 78621 solicitudes con una latencia de 35.7914 ms en la misma categoría de arranque.

De acuerdo con los resultados se puede comprobar que Quarkus tiene un mejor rendimiento que Spring Boot, en consecuencia, Quarkus mejora el rendimiento de las aplicaciones serverless.

Tabla 3

Resultados de latencia con JVM

	Cold Start (ms)	**Warm Start (ms)**

	p50	p90	p99	max	p50	p90	p99	max
Quarkus	7001.22	7369.91	7967.23	7967.23	11.8965	21.4035	62.988	578.98
Spring Boot	13137.9355	13687.5053	14518.9381	14558.4	12.1101	23.2761	72.1615	603.67

Tabla 4

Resultados de latencia con GraalVM

	Cold Start (ms)				Warm Start (ms)			
	p50	**p90**	**p99**	**max**	**p50**	**p90**	**p99**	**max**
Quarkus	-	-	-	-	3.0864	5.1292	10.4778	82.66
Spring Boot	-	-	-	-	9.6115	15.2582	35.7914	477.1

Tabla 5

Resultados de latencia y solicitudes atendidas

Framework	**Cantidad de Solicitudes atendidas**	**P99 (Latencia Arranque en Frío)**	**P99 (Latencia Arranque en caliente)**
Quarkus - JVM	151167	7967.23 ms	62.988 ms
Quarkus - GraalVM	153526	0.0000001 ms	10.4778 ms
Spring Boot - JVM	74686	14518.9381ms	72.1615 ms
Spring Boot - GraalVM	78621	0.0000001 ms	35.7914 ms

DISCUSIÓN GENERAL DE LOS RESULTADOS

Los datos de la tabla 5 evidencian que el rendimiento de Spring Boot es superado por Quarkus demostrando que es capaz de procesar el doble de solicitudes en el mismo rango de tiempo, debido Quarkus con GraalVM atiende 153526 solicitudes. Por otro lado, Spring Boot con GraalVM atiende 78621 solicitudes, además se puede evidenciar que hay menos latencia

cuando se utiliza Quarkus con GraalVM, mostrando que el 99% de las solicitudes tienen una latencia menor o igual a 10.4778 ms.

Con base a lo mencionado anteriormente, respecto a la interpretación de los resultados, se demuestra que la hipótesis que se planteó es verdadera, por ende, la aplicación serverless implementada con GraalVM y Quarkus tiene mejor rendimiento que una aplicación serverless implementada con Spring Boot.

Después de haber realizado la presente investigación se recomienda que para estudios futuros se considere desarrollar una comparativa entre los lenguajes más populares como son: TypeScript, Python, Java, C#, Golang y Rust; en diferentes proveedores de nube y cargas de trabajo utilizando programación asíncrona, esto permitirá ampliar la presente investigación tomando como guía este trabajo.

Capítulo V

Conclusiones y recomendaciones

Conclusiones

- La imagen nativa disminuye el tiempo de arranque en frío, en consecuencia, la latencia es mínima cuando se utiliza Quarkus con GraalVM, mejorando el rendimiento de la aplicación serverless, evidenciando que Quarkus procesa aproximadamente un 48,79% más de solicitudes que Spring Boot en el mismo rango de tiempo.
- El marco de trabajo Serverless Application Model (SAM) permitió desplegar la infraestructura necesaria para la aplicación y transformación digital de la empresa, haciendo uso del proveedor de nube AWS para enfocarse en la innovación de sus productos.
- La aplicación desarrollada agilizó las operaciones de la empresa RM Seguridad Industrial eliminando el trabajo manual en gestión de productos, minimizando los tiempos de respuestas y maximizando su rendimiento.

Recomendaciones

- Mantener actualizadas las dependencias de Quarkus y Spring en el proyecto, ya que los frameworks se encuentran en constante actualización, dichas actualizaciones pueden contener mejoras en el funcionamiento del framework,dando como resultado un aumento de rendimiento del proyecto.
- Para mejorar el rendimiento aún más se puede utilizar programación asincrónica como: Munity para Quarkus y el proyecto React para Spring Boot.
- Utilizar la suite de utilidades de PowerTools de Java para implementar las mejores prácticas dentro de aplicaciones serverless.
- Utilizar una arquitectura hexagonal para desarrollar una arquitectura evolutiva.

Bibliografía

Barrios Contreras, D. A. (2018). *Arquitectura de Microservicios. Tecnología Investigación y Academia.* Obtenido de https://revistas.udistrital.edu.co/index.php/tia/article/view/9687

Amazon Web Services. (2022). *¿Qué es Amazon DynamoDB?* Obtenido de https://docs.aws.amazon.com/es_es/amazondynamodb/latest/developerguide/Introduction.html

Amazon Web Services. (2022). *¿Qué es AWS Lambda?* Obtenido de Documentación: https://docs.aws.amazon.com/es_es/lambda/latest/dg/welcome.html

Amazon Web Services. (2022). *Guía del usuario de la versión 2.* Obtenido de Documentación: https://docs.aws.amazon.com/es_es/cli/latest/userguide/cli-chap-welcome.html

Pearl Brereton, B. A. (2007). *Lessons from applying the systematic literature review process within the software engineering domain.* Obtenido de ScienceDirect: https://www.sciencedirect.com/science/article/pii/S016412120600197X

Dragoni, N., Giallorenzo, S., Lluch Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). *Microservices: Yesterday, Today, and Tomorrow.* Obtenido de SpringerLink: https://link.springer.com/chapter/10.1007/978-3-319-67425-4_12

van Eyk, E., Grohmann, J., Eismann, S., Bauer, A., Versluis, L., Toader, L., . . . Iosup, A. (2019). *The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms.* Obtenido de IEEE Xplore: https://ieeexplore.ieee.org/document/8894540

GraalVM. (2022). *Introduction to GraalVM*. Obtenido de Documentation: https://www.graalvm.org/22.2/docs/introduction/

Gutierrez, F. (2018). *Spring Framework 5.* Berkeley, CA: Apress.

JetBrains. (2022). *Essential tools for software developers and teams*. Obtenido de JetBrains: https://www.jetbrains.com/

Mozilla Developer Network. (2022). *JavaScript*. Obtenido de https://developer.mozilla.org/en-US/docs/Web/JavaScript

Material UI. (2022). *Material UI - Overview*. Obtenido de MUI Core: https://mui.com/material-ui/getting-started/overview/

Newman, S. (2019). *Monolith to Microservices.* O'Reilly Media, Inc.

Quarkus. (2022). *What is Quarkus?* . Obtenido de Quarkus: https://quarkus.io/about/

Facebook Open Source. (2022). *React. A JavaScript library for building user interfaces*. Obtenido de React: https://reactjs.org/

Molina, J., Loja, N., Zea, M., & Loaiza, E. (2016). *Evaluación de los Frameworks en el Desarrollo de Aplicaciones Web con Python.* Obtenido de Revista Latinoamericana de Ingeniería de Software: http://revistas.unla.edu.ar/software/article/view/1149

Šipek, M., Mihaljević, B., & Radovan, A. (2019). Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM. En *42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (págs. 1671-1676). Obtenido de IEEE Xplore: https://ieeexplore.ieee.org/document/8756917

Šipek, M., Muharemagić, D., Mihaljević, B., & Radovan, A. (2020). Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus. En *43rd International Convention on Information, Communication and Electronic Technology (MIPRO).*

Styled Components. (2022). *Documentation*. Obtenido de Styled Components: https://styled-components.com/docs

Thönes, J. (2015). Microservices. *IEEE Software*, 116-116. Obtenido de IEEE Xplore: https://ieeexplore.ieee.org/abstract/document/7030212

NextJS. (2022). *Introduction*. Obtenido de NextJS: https://nextjs.org/learn/foundations/about-nextjs

Xiao, P. (2019). Introduction to Java . En *Practical Java Programming for IoT, AI, and Blockchain* (págs. 1-12). Wiley Data and Cybersecurity.

Almeida, M. S. (2020). *Uma análise comparativa de desempenho entre diferentes tecnologias de execução de aplicações web do lado do servidor.* Obtenido de Repositório Institucional da UFSCar: https://repositorio.ufscar.br/handle/ufscar/13637?show=full

Velepucha, V., Flores, P., & Torres, J. (2019). Migration of Monolithic Applications Towards Microservices Under the Vision of the Information Hiding Principle: A Systematic Mapping Study. En *Advances in Emerging Trends and Technologies* (págs. 90–100). Springer, Cham.

Koleoso, T. (2020). *Beginning Quarkus Framework.* Apress Berkeley, CA.

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., . . . Lang, M. (2017). Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. En *Service Oriented Computing and Applications* (págs. 233–247).

Reuter, A., Back, T., & Andrikopoulos, V. (2020). Cost efficiency under mixed serverless and serverful deployments. En *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (págs. 242-245). IEEE.

Amazon Web Services. (2022). *¿Qué es Amazon CloudWatch?* Obtenido de Documentación: https://aws.amazon.com/cloudwatch/

Apéndice