PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Escuela de Posgrado

Propuesta de método de evaluación de tecnologías serverless tipo FaaS y CaaS para el despliegue de software de aplicaciones transaccionales

Tesis para obtener el grado académico de Maestro en Informática con mención en Ingeniería de Software que presenta:

Guillermo Dante Matos Cuba

Asesor:

Mg. Dennis Stephen Cohn Muroy

Lima, 2023

Informe de Similitud

Yo, **Dennis Stephen COHN MUROY**, docente de la Escuela de Posgrado de la Pontificia Universidad Católica del Perú, asesor de la tesis titulada “Propuesta de método de evaluación de tecnologías serverless tipo FaaS y CaaS para el despliegue de software de aplicaciones transaccionales”, del autor **Guillermo Dante MATOS CUBA**, dejo constancia de lo siguiente:

- El mencionado documento tiene un índice de puntuación de similitud de **08%**. Así lo consigna el reporte de similitud emitido por el software Turnitin el **19/01/2024**.
- He revisado con detalle dicho reporte y la tesis y no se advierte indicios de plagio.
- Las citas a otros autores y sus respectivas referencias cumplen con las pautas académicas.

Lugar y fecha:

Lima, 19 de enero del 2024.

Apellidos y nombres del asesor: **COHN MUROY, Dennis Stephen**	
DNI: **43513429**	Firma 
ORCID: **0000-0003-4820-0178**	

Dedicatoria

A mis padres Agripina y Guillermo.
A mis hermanos Renzo y Cesar.

Agradecimientos

El desarrollo de esta tesis implicó un gran reto en el plano personal, profesional y familiar. El trabajo realizado ha sido posible gracias al apoyo de un grupo de personas a las cuales presento mi más profundo agradecimiento en estas líneas.

A mis padres, Agripina y Guillermo, por su comprensión y apoyo constante.

A mi asesor Dennis Cohn por sus enseñanzas en arquitectura de software, sus recomendaciones para elaborar esta tesis y su paciencia para apoyarme en conseguir los objetivos desafiantes de esta investigación.

A la profesora Loretta Gasco por sus enseñanzas de estadística y énfasis en la práctica con el lenguaje de programación R lo cual me permitió aplicar diversas herramientas estadísticas durante la evaluación de resultados de los experimentos.

Resumen

Los proveedores de servicios de computación en nube ofrecen una amplia variedad de servicios de despliegue de *software* que están en constante actualización, lo que implica diversos desafíos para arquitectos y desarrolladores cuando tiene que decidir cuál es la opción más adecuada para cumplir con los requerimientos de *performance*, generándose entonces la necesidad de validar con precisión los parámetros de configuración de los servicios de despliegue ofrecidos por estos proveedores.

El estudio realizado propone un método de evaluación de los servicios de despliegue de software que presentan mayores avances en la actualidad, denominados *serverless*, considerando a FaaS y las nuevas versiones de CaaS como las tecnologías que representan sus beneficios.

El método se ha elaborado en base a buenas prácticas de pruebas de *performance* e investigación experimental.

Para validar la efectividad del método se han implementado experimentos en la plataforma de AWS usando una aplicación de *benchmark* desarrollado exclusivamente para este estudio, durante la experimentación se observaron oportunidades para optimizar costos en el diseño y selección de servicios de los componentes de una arquitectura CaaS.

Estos experimentos han sido realizados usando el *framework* de desarrollo de software de AWS CDK que permite el despliegue y eliminación de arquitecturas en segundos, lo cual permite optimizar tiempos y costos durante el ciclo de evaluación de *performance*.

Abstract

Cloud computing service providers offer a wide variety of software deployment services that are constantly updated, which implies various challenges for architects and developers when they need to decide which is the most appropriate option to meet performance requirements. Thus, generating the need to precisely validate the configuration parameters of the deployment services offered by these providers.

The study carried out proposes a method for evaluating the software deployment services that currently present the greatest advances, called serverless, considering FaaS and the new versions of CaaS as the technologies that represent their benefits.

The method has been developed based on good practices of performance testing and experimental research.

To validate the effectiveness of the method, experiments have been implemented on the AWS platform using a benchmark application developed exclusively for this study. During the experimentation, opportunities were observed to optimize costs in the design and selection of services of the components of a CaaS architecture.

These experiments have been carried out using the AWS CDK software development framework that allows the deployment and elimination of architectures in seconds, which allows optimizing time and costs during the performance evaluation cycle.

ÍNDICE

Índice de tablas

Índice de figuras

1 INTRODUCCIÓN

La tecnología de computación *serverless* ha ganado la atención de muchas empresas tanto grandes como pequeñas [1]. La computación *serverless* es un modelo de ejecución general para aplicaciones distribuidas en la nube, en este modelo, la responsabilidad para gestionar los recursos corresponde exclusivamente a los proveedores de nube, en consecuencia, esto permite que los desarrolladores se enfoquen en el desarrollo centrado en el usuario final [2] .

Además, desde la perspectiva de los clientes de los proveedores de nube se promete más ahorros con el modelo de pago por uso, en el cual solo la invocación de funciones activas es cobrada mientras que el pago de máquinas virtuales IaaS estándar ocasiona costos incluso cuando no están en uso [3].

Sin embargo, muchos autores plantean varios problemas importantes, por ejemplo, la dependencia exclusiva de un proveedor de plataforma comercial, falta de herramientas especializadas para desarrollo y *debugging*, gastos generales impredecibles debido a la alta latencia de los *cold starts*

y sorprendentemente altos costos para código de computación intensiva [3].

Entonces, en los despliegues en nube, es crítico que los usuarios tengan un conocimiento preciso

del *performance* de sus aplicaciones para que puedan elegir las configuraciones adecuadas del servicio de nube y de esta forma satisfacer los objetivos de *performance* y costos [4].

Una forma de hacer pruebas de performance es desplegando sistemas de *software* propios y aplicaciones en la infraestructura *cloud* y realizar pruebas rigurosas con ellos para evaluar su *performance* bajo diferentes condiciones, sin embargo, este enfoque puede ser complejo, prolongado y caro, y un pequeño o mediano negocio podría no poseer el tiempo, recursos y especialización propia

para hacer una evaluación exhaustiva y proactiva de esta forma [5].

Una alternativa más práctica es realizar un *benchmark* de aplicaciones representativas contra cargas de trabajo representativas para estimar el *performance* de diferentes tipos de aplicaciones cuando están desplegados en la infraestructura *cloud*. Luego, los resultados del *benchmarking* pueden ser usados para estimar el *performance* de diferentes ofertas de infraestructura y obtener observaciones valiosas de las diferencias de *performance* entre proveedores [5].

Esta investigación presenta un método práctico y eficiente acorde con las buenas prácticas de investigación experimental que permita la evaluación de performance de aplicaciones desplegadas en los servicios *serverless* de proveedores de nube.

1.1 Planteamiento del problema

Existe una intensa actualización de tecnologías *serverless* en los proveedores de nube que no está siendo evaluada adecuadamente [3], como las tecnologías basadas en contenedores (CaaS), la cual tienen diversas propuestas de optimización para competir con tecnologías FaaS [2]. Además, las empresas proveedoras de nube ofrecen sus propias versiones de tecnología *serverless* con diferentes funcionalidades y precios. En consecuencia, los negocios que buscan desplegar sus aplicaciones usando el paradigma *serverless* se enfrentan con un rango de opciones, como la elección del proveedor y por cada proveedor, una variedad de opciones de configuración. Por otro lado, cada una de estas elecciones de diseño tiene un impacto en el *performance* de la aplicación [6].

Por lo tanto, evaluar los servicios candidatos de nube es crucial y beneficioso para los clientes (ej. En el análisis de costo beneficio) y proveedores (ej. En la gestión de optimización). Además, cuando se trata de realizar cualquier evaluación de *performance*, una metodología adecuada es requerida inevitablemente para dirigir las implementaciones experimentales [7].

1.2 Objetivo de la investigación

1.2.1 Objetivo general

Se busca reducir la dificultad de elección entre las diversas alternativas de tecnología de despliegue de software que ofrecen los proveedores de nube para garantizar un adecuado performance, en particular esta investigación se enfocará en las tecnologías *serverless* del tipo FaaS y CaaS de AWS. Para lo cual se elaborará una propuesta de método de evaluación de *performance* y un sistema de experimentación que permita mejorar la efectividad del método propuesto y facilite la repetición de los experimentos.

1.2.2 Objetivos específicos

O1: Elaborar una propuesta metodológica orientada a experimentación para servicios *serverless* de AWS que permite el despliegue de aplicaciones con optimización de recursos en la ejecución del experimento.

O2: Diseñar y realizar experimentación que verifique la efectividad del método propuesto. Incluyendo el diseño de sistema para ejecutar las pruebas a cualquier cantidad de APIs y en forma automatizada mediante el uso del *framework* AWS CDK.

O3: Validar la metodología propuesta en base a prácticas actuales de experimentación de *performance* en nube.

O4: Elaborar matrices comparativas con las métricas eficiencia de desempeño para la transaccional de *benchmark* desplegada en FaaS y CaaS.

1.3 Hipótesis, métodos e instrumentos

1.3.1 Hipótesis

Seguir un proceso riguroso y sistemático basado en buenas prácticas de investigación científica permite diferenciar claramente que tipo de servicio *serverless* proporciona mejores resultados de rendimiento para aplicaciones transaccionales.

1.3.2 Métodos empleados

- Ejecución de una revisión sistemática de literatura.
- Elaboración de un método de evaluación basado en buenas prácticas de experimentación en plataformas de nube.
- Desarrollo de software de *benchmark* y despliegue sobre plataforma de nube para la ejecución de los experimentos.
- Análisis de los gráficos y datos de las herramientas estadísticas para evaluar las hipótesis de comparación de performance entre FaaS y CaaS.

1.3.3 Instrumentos

- ***Software* JMeter:** Se ha configurado para generar las pruebas de carga.
- **Plataforma de nube de Google:** Se ha usado el servicio de máquinas virtuales para el despliegue del código de JMeter.

- ***Software* de *benchmark*:** Se ha desarrollado para ser el sistema bajo pruebas de referencia en la comparación de performance.
- **Plataforma de nube de AWS:** Se ha usado los servicios de despliegue FaaS y CaaS así como servicios de monitoreo para la ejecución del *software* de *benchmark* y generación de archivos CSV con los tiempos de respuesta de ambos servicios *serverless*.
- ***Software* RStudio:** Se ha usado para aplicar las herramientas estadísticas sobre los archivos CSV generados por la plataforma de nube de AWS.
- ***Framework* de *software* CDK:** Se he generado un script para automatizar el despliegue sobre la plataforma de nube de AWS.

1.4 Conclusiones más relevantes

Se ha identificado que el servicio de despliegue FaaS tiene mejor relación costo-beneficio, asociado al performance, que el servicio de CaaS en todos los escenarios planteados.

También, se ha comprobado la utilidad del *framework* CDK para automatizar el despliegue de infraestructuras en la plataforma de nube de AWS.

Por otro lado, se ha verificado la efectividad de los métodos estadísticos no paramétricos, así como la utilidad del *test* de Brunner-Munzel para los casos en los que no se cumple con el supuesto estadístico de igualdad de varianzas.

2 MARCO CONCEPTUAL

2.1 Serverless

Paradigma en el cual las aplicaciones de software ya no requieren la administración de servidores, es decir, los usuarios de servicios *serverless* ya no necesitan dedicar tiempo y utilizar recursos en el aprovisionamiento, mantenimiento, actualización, escalamiento y planeamiento de capacidad de los servidores debido a que todas estas tareas y capacidades son manejadas por la plataforma [6] .

Además, este concepto describe un modelo de despliegue en el cual las aplicaciones son descompuestas en múltiples funciones independientes. Estas funciones solamente son ejecutadas en respuesta a disparadores (*triggers*), como las interacciones de usuario, eventos de mensajes o cambios en base de datos. Estas características también son atribuidas a los servicios *function-as-a-service* (Faas) [8] .

Finalmente, este tipo de servicios generalmente son gestionados por un proveedor, actualmente los mayores proveedores que ofrecen soluciones para este tipo de servicio son AWS, Azure, Google e IBM, cuyas plataformas permiten la ejecución, escalamiento y facturación en respuesta exacta a los recursos consumidos [8] .

2.2 Contenedor como Servicio (CaaS)

Es un modelo computación en la nube recientemente mejorado como serverless y denominado Fargate por AWS, el cual fue liberado en el 2017 [9] . Su principal característica es el uso de virtualización basada en contenedores.

En comparación con FaaS, CaaS agrega una capa adicional en forma de contenedor que agrupa las aplicaciones y puede ser. El contenedor es desplegado en una plataforma de nube. Cuando se ejecuta un disparador (*trigger*), la instancia de contenedor (*container*) es aprovisionada o desaprovisionada en el camino, dependiendo de la demanda. Esta envoltura (*wrapping*) de contenedores, si bien agrega

complejidad adicional, también ofrece más capacidades para configurar entornos de ejecución. Los principales ejemplos de estos tipos de contenedores elásticos son AWS Fargate, Google *Cloud Run* y Azure *container instances* [2].

También se considera que CaaS combina el concepto de *serverless* y contenedores ya que se reduce la complejidad operacional y se aplica un modelo de precios de acuerdo con el uso. Aunque hay documentación disponible por los proveedores de nube y artículos al respecto, el modelo CaaS es relativamente nuevo, por ejemplo, Google *Cloud run* se liberó completamente el 3 de noviembre del 2019 [10].

2.3 Lambda

Es un servicio informático basado en eventos y de pago por uso que permite la ejecución de código sin necesidad de aprovisionar ni administrar servidores [11].

2.4 Contenedor Docker

Un contenedor es una unidad estándar de software que empaqueta código y todas sus dependencias para que la aplicación se ejecute en forma rápida y confiable desde un ambiente informático hacia otro. Y una imagen de contenedor Docker es un paquete de *software* ejecutable, ligero y autónomo que incluye todo lo necesario para ejecutar una aplicación: código, tiempos determinados de ejecución, un sistema de herramientas, un sistema de librerías y configuraciones [12].

2.5 Fargate

Es un motor informático *serverless* que funciona tanto con Amazon *Elastic Container Service* (ECS) como con Amazon Elastic Kubernetes *Service* (EKS) [13].

2.6 AWS CDK

Es un marco de desarrollo *open source* que define la infraestructura de *software* de nube como código mediante lenguajes de programación modernos y se implementa con AWS Cloudformation [14].

2.7 Pruebas de rendimiento de software

Se refiere a las actividades de ejecución de pruebas y evaluaciones para validar el rendimiento y capacidad de un sistema. Tiene 3 objetivos principales, el primero es validar si se cumple con los requerimientos de rendimiento para un determinado producto. El siguiente es identificar la capacidad del producto y sus límites. El tercero es descubrir problemas de rendimiento, degradaciones, mejoras o cuellos de botella de un sistema de software y sus componentes para contribuir con la solución de problemas y ajustes de configuraciones de rendimiento [15].

En este *paper* se considerará que los términos rendimiento de software, rendimiento de ejecución, eficiencia de la aplicación son equivalentes al término *performance*.

3 REVISIÓN SISTEMÁTICA DE LA LITERATURA

En esta sección se presenta la revisión sistemática de literatura (*Systematic Literature Review* - SLR), la cual se define como un medio de identificación, evaluación e interpretación de las investigaciones relevantes a un tema en particular [16]. En el presente estudio, se ha decidió realizar 2 SLR, la primera

denominada técnica para estar enfocado en los *papers* que incluyan experimentos y la segunda denominada metodológica para identificar buenas prácticas de evaluación.

3.1 Proceso de la revisión

Se ha elaborado el diagrama de la figura 3.1 en base a lo recomendado en [17] [18] para representar las etapas a seguir en el proceso de revisión.

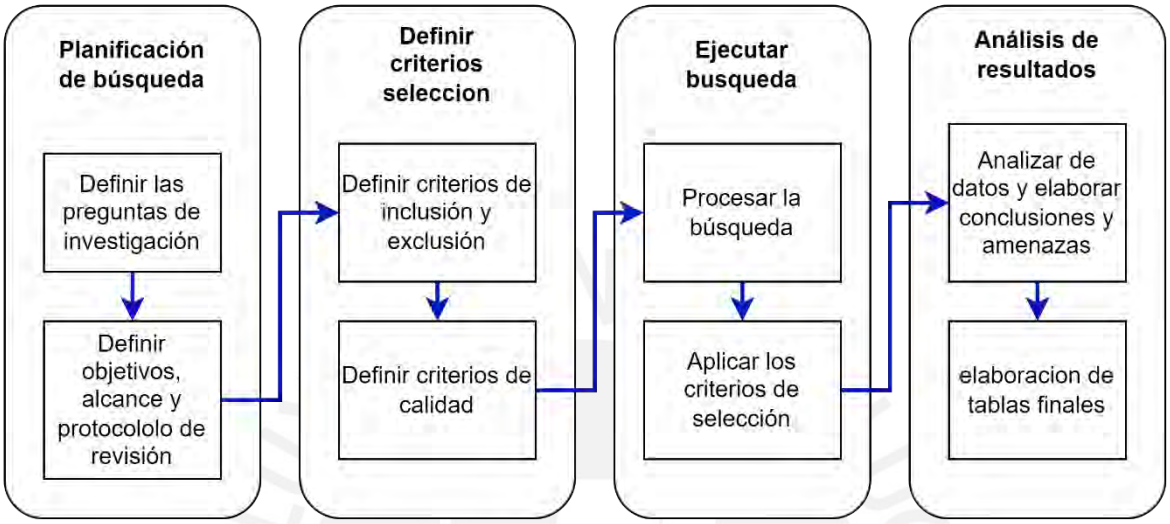

Figura 3.1: Proceso seguido en las revisiones sistemáticas planteadas

3.1.1 Revisión sistemática técnica

Se ha realizado esta SLR el 5 de julio del 2022 para identificar las investigaciones que aborden la temática de evaluación de performance mediante la comparación de tecnologías despliegue FaaS y CaaS en nube y de esa forma validar que se está realizando un nuevo aporte sobre el tema.

3.1.1.1 Preguntas de investigación

Para identificar las preguntas de investigación se aplicó la técnica PICo mostrada en la tabla 3.1.

Tabla 3.1: Estrategia PICo para SLR técnico

Concepto	**Ámbito de investigación**
Population	Servicios de proveedores de nube para desplegar *software*
Interest	Servicios de nube de paradigma *serverless* del tipo FaaS y CaaS
Context	Evaluación de *performance*

Se definen las siguientes preguntas de investigación:

RQ1: ¿Cuantos artículos abordan la evaluación de *performance* basado en experimentos para tecnologías de infraestructura de despliegue de software en nube?

RQ2: De las evaluaciones de *performance* propuestas en los artículos seleccionados ¿Qué tipo de aplicaciones utilizan en sus experimentos?

RQ3: De las evaluaciones de *performance* propuestas en los artículos seleccionados ¿Cuáles tratan la comparación entre los paradigmas FaaS y CaaS para aplicaciones transaccionales?

3.1.1.2 Criterios de inclusión, exclusión

Para iniciar la selección de artículos que se obtendrán de la cadena de búsqueda se definieron los siguientes criterios:

Inclusión:

1. El artículo debe incluir en sus comparaciones a la tecnología FaaS.
2. Los experimentos deben ser realizados en AWS y adicionalmente en otros proveedore de nube.
3. Los experimentos se enfocaron en el *backend*, por ejemplo, mediante solicitudes hacia las APIs REST de la aplicación.
4. Las aplicaciones estudiadas son transaccionales.

Exclusión:

1. Las aplicaciones estudiadas que tratan sobre *machine learning* o aplicaciones científicas.
2. Las comparaciones de una misma tecnología de despliegue en distintas plataformas de nube.
3. Si no se realizan experimentos durante la evaluación.
4. Libros y artículos de opinión.

3.1.1.3 Estrategia de búsqueda

Se definieron las cadenas de búsqueda mostradas en la tabla 3.2 para que sean usadas en las bases de datos de IEEE, ACM y Scopus.

Tabla 3.2: Cadenas de búsqueda PICo para SLR técnico

Concepto	**Cadena de Búsqueda**
Population	cloud OR aws OR amazon OR azure OR microsoft OR gcp OR google OR ibm OR alibaba OR iaas OR "infrastructure-as-a-service" OR paas OR "platform-as-a-service
Interest	serverless OR "server-less" OR faas OR "function-as-a-service" OR caas OR "container-as-a-service"
Context	(evaluat* OR assess* OR measur* OR experiment* OR analy* OR investigat* OR examin* OR stud* OR research* OR test*) AND (perform* OR nonfunctional OR "architecture attribute" OR "architecture requirement" OR "architecture requirements" OR "quality attribute" OR "quality attributes" OR "quality characteristic" OR "quality characteristics" OR nfr)

3.1.1.4 Resultados aplicando criterios de inclusión y exclusión

Los resultados de búsqueda luego de aplicar los criterios de selección en BD se muestran en las tablas 3.3, 3.4 y 3.5.

Tabla 3.3: Criterios de papers relevantes para SLR técnico

Base de Datos	Resultado inicial	Sin repetidos	Revisión de *abstract*	Revisión completa
ACM	222	222	7	2
IEEE	288	272	5	1
Scopus	497	198	6	1

Tabla 3.4: Artículos seleccionados en lectura de abstract

Nombre	Fuente	Referencia
Serverless Containers – Rising Viable Approach to Scientific Workflows	IEEE	[2]
BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms	IEEE	[19]
Operating Latency Sensitive Applications on Public Serverless Edge Cloud Platforms	IEEE	[20]
Towards Latency Sensitive Cloud Native Applications: A Performance Study on AWS	IEEE	[21]
Serverless Computing: An Investigation of Factors Influencing Microservice Performance	IEEE	[22]
Optimizing Latency Sensitive Applications for Amazon's Public Cloud Platform	ACM	[23]
SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing	ACM	[3]
An Evaluation of Serverless Data Processing Frameworks	ACM	[24]
FaaSdom: A Benchmark Suite for Serverless Computing	ACM	[25]
A Case Study on the Stability of Performance Tests for Serverless Applications	ACM	[26]
An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems	ACM	[27]
Cost Comparison of Running Web Applications in the Cloud Using Monolithic, Microservice, and AWS Lambda Architectures	ACM	[28]

On the Performance Implications of Deploying IoT Apps as FaaS	Scopus	[29]
A traffic analysis on serverless computing based on the example of a file upload stream on aws lambda	Scopus	[30]
Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions	Scopus	[31]
Cloud Deployment Tradeoffs for the Analysis of Spatially Distributed Internet of Things Systems	Scopus	[32]
Microservices vs serverless: A performance comparison on a cloud-native web application	Scopus	[33]
An Event-Driven Serverless ETL Pipeline on AWS	Scopus	[34]

Tabla 3.5: Artículos seleccionados en lectura completa

Nombre	Fuente	Referencia
BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms	IEEE	[19]
Microservices vs serverless: A performance comparison on a cloud-native web application	Scopus	[33]
A Case Study on the Stability of Performance Tests for Serverless Applications	ACM	[26]
Cost Comparison of Running Web Applications in the Cloud Using Monolithic, Microservice, and AWS Lambda Architectures	ACM	[28]

3.1.1.5 Criterios de calidad

Para completar la selección de *papers* se definieron los siguientes criterios de calidad:

QA1: ¿Se utiliza *benchmarks* de *apps* públicas indicando su nombre exacto y fuente o en caso se desarrolle una *app* para el experimento, se explica su arquitectura y se indica donde se encuentra el código fuente?

QA2: ¿Se muestran los diagramas de arquitectura de despliegue en nube, además de explicar los aspectos de su configuración que afectan el experimento?

QA3: ¿Se aplican conceptos estadísticos para sustentar sus resultados?

QA4: ¿Se indica claramente la cantidad de métricas que fueron parte del experimento?

QA5: ¿Se describe el perfil de *workload* indicando el detalle de su configuración o se hace referencia a un *workload* ya desarrollado?

3.1.1.6 Resultados aplicando criterios de calidad

La tabla 3.6 muestra los resultados de aplicar los filtros de calidad, se observa que 1 *paper* tiene calificación 0 en QA3. Por lo tanto, solo 3 serán considerados en el análisis.

Tabla 3.6: Artículos seleccionados con filtro de calidad

Referencia	QA1	QA2	QA3	QA4	QA5
[19]	4	2	1	3	2
[33]	2	2	1	1	2
[26]	3	2	4	3	3
[28]	1	2	0	2	2

3.1.1.7 Discusión

Se observa que solo 1 *paper*, de los seleccionados por criterios de inclusión y exclusión, cumple con los mínimos criterios de calidad y presenta una propuesta similar al tema de esta tesis.

- **RQ1: ¿Cuantos artículos que abordan la evaluación de *performance* basado en experimentos para tecnologías de infraestructura de despliegue de software en nube?**
  Se identificaron 18 artículos en la lectura de *abstract* y 5 en la lectura completa, los cuales cumplen con los criterios de inclusión y exclusión.

- **RQ2: De las evaluaciones de *performance* propuestas en los artículos seleccionados ¿Qué tipos de *apps* se utilizan en sus experimentos?**
  Se identificaron las *apps* mostradas en la tabla 3.7.

- **RQ3: De las evaluaciones de *performance* propuestas en los artículos seleccionados ¿Cuáles tratan la comparación entre los paradigmas FaaS y CaaS para aplicaciones transaccionales?**
  Solo se identificó 1 *paper*, de los 3 filtrados por calidad, que hace este tipo de evaluación. En el cual se denomina serverless a FaaS y microservicios a CaaS. En ese paper se recomienda usar CaaS en lugar de FaaS para aprovechar las ventajas de costos de CaaS cuando el tamaño de los paquetes de tráfico de red es relativamente pequeño y regular. En cambio, cuando se necesita estabilidad en los tiempos de respuesta sin considerar la complejidad de los paquetes de red se recomienda el uso FaaS en lugar de CaaS, ya que los servicios de CaaS necesitan de elementos de balanceo de tráfico que aumentan los tiempos de respuesta aleatoriamente.

Tabla 3.7: Artículos ordenados por tipo de aplicación

Tipo de aplicaciones	Total
Aplicación transaccional	4
IoT	3
Operaciones científicas	3
Operaciones transaccionales	2

	Aplicación científica	2
	Operac. transacc., científ., machine learn.	2
	Data analytics y big data	1
	Aplicaciones científicas	1
	Total general	**18**

3.1.2 Revisión sistemática metodológica

Se ha realizado el SLR el 6 de julio del 2022 para identificar *papers* relevantes de procesos de selección de servicios *cloud*. Además de identificar las buenas prácticas recomendadas y combinarlas en un método enfocado a implementar la temática técnica indicada.

3.1.2.1 Preguntas de investigación

Para identificar las preguntas de investigación se aplicó la técnica PICo mostrada en la tabla 3.8.

Tabla 3.8: Estrategia PICo para SLR metodológico

Concepto	**Ámbito de investigación**
Population	Servicios de nube de los principales proveedores de nube usado por desarrolladores
Interest	Métodos y metodologías
Context	Evaluación de performance

Se definen las siguientes preguntas de investigación:

RQ1: ¿Cuantos artículos tratan sobre metodologías o métodos de evaluación de *performance* de tecnologías de despliegue en *cloud*?

RQ2: De las metodologías y métodos propuestos en los artículos seleccionados ¿Cuáles son las etapas recomendadas para una experimentación de *performance*?

RQ3: De las metodologías y métodos propuestos en los artículos seleccionados ¿Cuáles son los proveedores usados en la experimentación?

RQ4: De las metodologías y métodos propuestos en los artículos seleccionados ¿Cuáles son las recomendaciones para optimizar costos?

3.1.2.2 Criterios de inclusión, exclusión

Para iniciar la selección de artículos que se obtendrán de la cadena de búsqueda se definieron los siguientes criterios:

Inclusión:

1. El artículo debe poder aplicarse a tecnologías de nubes CaaS y FaaS.
2. El artículo incluye análisis de costos.

Exclusión:

1. Los artículos que tratan sobre *machine learning* o aplicaciones científicas.

2. Los artículos que se especializan en el análisis del CPU.

3.1.2.3 Estrategia de búsqueda

Se definieron las cadenas de búsqueda mostradas en la tabla 3.9 para que sean usadas en las bases de datos de IEEE, ACM y Scopus.

Tabla 3.9: Cadenas de búsqueda PICo para SLR metodológico

Concepto	**Cadena de Búsqueda**
Population	cloud OR aws OR amazon OR azure OR microsoft OR gcp OR google OR ibm OR alibaba OR iaas OR "infrastructure-as-a-service" OR paas OR "platform-as-a-service" OR serverless OR "server-less" OR faas OR "function-as-a-service" OR caas OR "container-as-a-service"
Interest	method* OR framework OR technique OR approach OR procedure OR strategy
Context	(evaluat* OR assess* OR measur* OR experiment* OR analy* OR investigat* OR examin* OR stud* OR research* OR test*) AND (perform* OR nonfunctional OR "architecture attribute" OR "architecture requirement" OR "architecture requirements" OR "quality attribute" OR "quality attributes" OR "quality characteristic" OR "quality characteristics" OR nfr)

3.1.2.4 Resultados aplicando criterios de inclusión y exclusión

Los resultados de búsqueda luego de aplicar los criterios de selección en BD se muestran en la tabla 3.10, 3.11 y 3.12.

Tabla 3.10: Criterios de papers relevantes para SLR metodológico

Base de datos	Resultado inicial	Sin repetidos	Revisión de abstract	Revisión completa
ACM	49	49	5	1
IEEE	72	57	3	1
Scopus	199	109	11	2

Tabla 3.11: Artículos seleccionados con filtro de lectura de abstract

Nombre	*Fuente*	*Ref*
A Statistics-Based Performance Testing Methodology for Cloud Applications	ACM	[4]
PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications	ACM	[35]
Performance Evaluation of Cloud Systems: A Behavioural Approach	ACM	[36]
A Multi-Cloud Framework for Measuring and Describing Performance Aspects of Cloud Services Across Different Application Types	ACM	[37]
A Factor Framework for Experimental Design for Performance Evaluation of Commercial Cloud Services	ACM	[38]
Methodological Principles for Reproducible Performance Evaluation in Cloud Computing	IEEE	[18]
An Approach for the Performance Evaluation of Multi-tier Cloud Applications	IEEE	[39]
On a framework for analyzing cloud system performance	IEEE	[40]
Testing methodologies for cloud performance	Scopus	[41]
A transparent approach to performance analysis and comparison of infrastructure as a service providers	Scopus	[42]
Smart CloudBench—A framework for evaluating cloud infrastructure performance	Scopus	[5]
Performance evaluation and modeling method research based on IaaS cloud platform	Scopus	[43]
A performance measurement framework of cloud storage services	Scopus	[44]
Performance evaluation of cloud computing platforms using statistical methods	Scopus	[45]
A framework of performance measurement monitoring of cloud service infrastructure system for service activation	Scopus	[46]
A nonparametric multivariate method for performance analysis of virtual machines in cloud computing systems	Scopus	[47]
A evaluation method of system performance based on cloud theory	Scopus	[48]
Performance monitoring and evaluation methods of computing system under cloud mode	Scopus	[49]
Performance measurement technique of cloud storage system	Scopus	[50]

Tabla 3.12: Artículos seleccionados con filtro de lectura completa

Nombre	Fuente	Ref
A Statistics-Based Performance Testing Methodology for Cloud Applications	ACM	[4]
Methodological Principles for Reproducible Performance Evaluation in Cloud Computing	IEEE	[18]
Testing methodologies for cloud performance	Scopus	[41]
Smart CloudBench—A framework for evaluating cloud infrastructure performance	Scopus	[5]

3.1.2.5 Criterios de calidad

Para completar la selección de *papers* se definieron los siguientes criterios de calidad:

QA1: ¿Se presenta un *workflow* o se describe secuencialmente los pasos del método propuesto?

QA2: ¿Se realizan experimentos para validar y su metodología?

QA3: ¿Se muestran recomendaciones de análisis estadístico?

3.1.2.6 Resultados aplicando criterios de calidad

La tabla 3.13 muestra los resultados de aplicar los filtros de calidad, se observa que 1 *paper* tiene calificación 0 en QA2 y QA3. Por lo tanto, solo 3 serán consideradas en el análisis.

Tabla 3.13: Artículos seleccionados con filtro de calidad

Referencia	QA1	QA2	QA3
[4]	4	3	4
[18]	3	4	4
[41]	1	0	0
[5]	4	3	3

3.1.2.7 Discusión

Se observa la importancia de utilizar diversos *workloads*, la optimización de costos y diversos enfoques estadísticos.

Se presenta las respuestas a las preguntas de investigación:

- **RQ1: ¿Cuantos artículos tratan sobre metodologías o métodos de evaluación de *performance* de tecnologías de despliegue en *cloud*?**
  Se identificado 20 artículos que cumplen con los criterios de inclusión en la lectura de *abstracts* y 4 en lectura completa.

- **RQ2: De las metodologías y métodos propuestos en los artículos seleccionados ¿Cuáles son las etapas recomendadas para una experimentación de *performance*?**

En la tabla 3.14 se muestran las etapas de los 3 *papers* filtrados por calidad.

- **RQ3: De las metodologías y métodos propuestos en los artículos seleccionados ¿Cuáles son los proveedores usados en la experimentación?**
  Para el *paper* [4] son 2: Chameleon (CHM) y AWS (EC2). En el caso de [18] son 3: Apache CloudStack, AWS (EC2) y la infraestructura de supercomputadoras ASCI distribuidas (DAS) para universidades holandesas. Y para [5] son 2: Google Compute Engine y el Deutsche Borse *Cloud Exchange Marketplace*.

- **RQ4: De las metodologías y métodos propuestos en los artículos seleccionados ¿Cuáles son las recomendaciones para optimizar costos?**
  En el *paper* [4] se recomienda reducir al mínimo la cantidad y tiempo de ejecución de pruebas de experimentación de tal forma que se cumpla con los requerimientos estadísticos necesarios. En el caso de [18], documentar el modelo de costos y los gastos reales generados para cuantificar las ventajas económicas entre experimentos. Y para [5], automatizar el inicio y fin de las instancias de servidor de pruebas.

Tabla 3.14: Etapas de experimentación por paper

Paper	**Etapas**
A Statistics-Based Performance Testing Methodology for Cloud Applications	1.Ejecutar la aplicación de pruebas repetidamente, 2.Calcular la distribución de performance, 3.Comparar las distribuciones con técnicas estadísticas, 4.Inicializar las pruebas estadísticas si la comparación no es satisfactoria.
Methodological Principles for Reproducible Performance Evaluation in Cloud Computing	1.Decidir la cantidad de experimentos a ejecutar, 2.Definir cargas de trabajo similar a casos reales, 3.Decidir qué software y hardware se usará en la experimentación, 4.Preparar la documentación de los experimentos para ser publicados, 5.Completar las tablas estadísticas de los resultados, 6.Realizar evaluaciones estadísticas de los resultados, 7.Elaborar reportes de resultados indicando las unidades de medición, 8.Documentar el modelo de costos usado en los experimentos.
Smart CloudBench—A framework for evaluating cloud infrastructure performance	1.Seleccionar el proveedor, 2.Seleccionar el *benchmark*, 3.Especificar el plan de pruebas, 4.Seleccionar instancias de servidores *cloud*, 5.Monitorear el inicio del consumo de recursos, 6.Ejecutar el *benchmark*, 7.Monitorear el fin del consumo de recursos, 8.Adicionar los resultados, 9.Finalizar el uso de instancias, 10.Evaluar y analizar los resultados.

3.2 Amenazas a la validez

- Debido a la alta cantidad de artículos de la primera búsqueda por *strings*, alguno importante pudo haber sido ignorado al momento de evaluarlo por lectura de *abstract,* sobre todo, para el caso de la revisión sistemática metodológica.
- Se pudo ampliar la búsqueda a más bases de datos.
- No se ha realizado una búsqueda por referencias de los artículos filtrados.

3.3 Conclusiones

- Existen muy pocas investigaciones de evaluación FaaS vs CaaS con un mínimo de calidad de experimentación ya que sólo se encontró un *paper* que lo cumple.
- En los *papers* sobre metodologías y métodos de evaluación de performance de *apps* desplegadas en nube se realiza la experimentación de su propuesta usando un solo tipo de tecnología de despliegue (IaaS o CaaS o FaaS).
- Existen algunos artículos que presentan sistemas automatizados de evaluación de *performance* de comparación con buena calidad de experimentación, pero no indican claramente cuál ha sido la metodología o método seguido para poder replicar las pruebas sin necesidad de usar todo su sistema.

4 PROPUESTA DE MÉTODO DE EVALUACIÓN Y SISTEMA DE EXPERIMENTACIÓN

En esta sección se presenta la propuesta de evaluación con sus componentes así como un sistema de ejecución del experimento de validación.

4.1 Diseño del método

Para el diseño del método se ha considerado como primera referencia del método presentado en la figura 4.3 al proceso presentado en la figura 4.1 del libro [15], además se consideraron los 8 principios indicados en [18]. Además, se considera que este proceso será usado dentro de un proceso de desarrollo de software, específicamente en el contexto de las tareas realizadas según la figura 4.2 de [51]. Este último punto complementa las recomendaciones de las 2 fuentes indicados para poder generar una propuesta de método práctica que pueda ser usada por analistas de ejecución de pruebas de *performance* en comunicación con desarrolladores y arquitectos.

En la tabla 4.1 se indica un mapeo sobre los pasos de los métodos propuestos en las 2 principales referencias [15] y [18].

Tabla 4.1: Mapeo de las recomendaciones consideradas en la propuesta del método

Pasos de método propuesto	**Proceso según** Gao et al. [15]	**Principios según** Papadopoulos et al. [18]
1.Definición del contexto y evaluación de la arquitectura de la aplicación de *benchmark*	• Comprensión y ajuste de los requerimientos de performance. • Identificar los puntos principales para las pruebas según los gestores de desarrollo y analistas de sistemas	• No está presente
2. Definición de las métricas, variables, factores, parámetros, y preguntas de investigación	• Definición de las métricas de performance	• Indicar las unidades de medición
3. Definición de la complejidad del perfil de carga	• No está presente	• Alcance de configuraciones de *workload*
4. Planificación del experimento	• Identificar las necesidades de las pruebas de performance, incluyendo técnicas, recursos y herramientas	
5.Diseño del experimento	• Especificar el plan de pruebas de performance	• Descripción de la configuración del experimento • Definir la cantidad de repeticiones del experimento
6.Configuración y ejecución de pruebas de validación del experimento	• Desarrollo y despliegue de las pruebas de performance • Ejecución de las pruebas de performance	• No está presente

7.Ejecución preliminar del experimento y evaluación parcial de resultados	• Ejecución de las pruebas de performance	• Describir las características estadísticas de los resultados • Indicar el nivel de significancia de las comparaciones estadísticas
8.Resolución de observaciones en la app de *benchmark*	• No está presente	• No está presente
9.Resolución de observaciones técnicas y de costos en la infraestructura	• No está presente	• Indicar el modelo de costos usado
10.Ejecución completa del experimento y evaluación preliminar completa de resultados	• Ejecución de las pruebas de performance • Validación de resultados y elaboración de reporte	• Describir las características estadísticas de los resultados • Indicar el nivel de significancia de las comparaciones estadísticas
11.Procesado final y análisis la información	• Validación de resultados y elaboración de reporte	• Hacer disponibles los archivos y código fuente usado en el experimento
12.Presentación de resultados al área de arquitectura de desarrollo e infraestructura	• No está presente	• No está presente

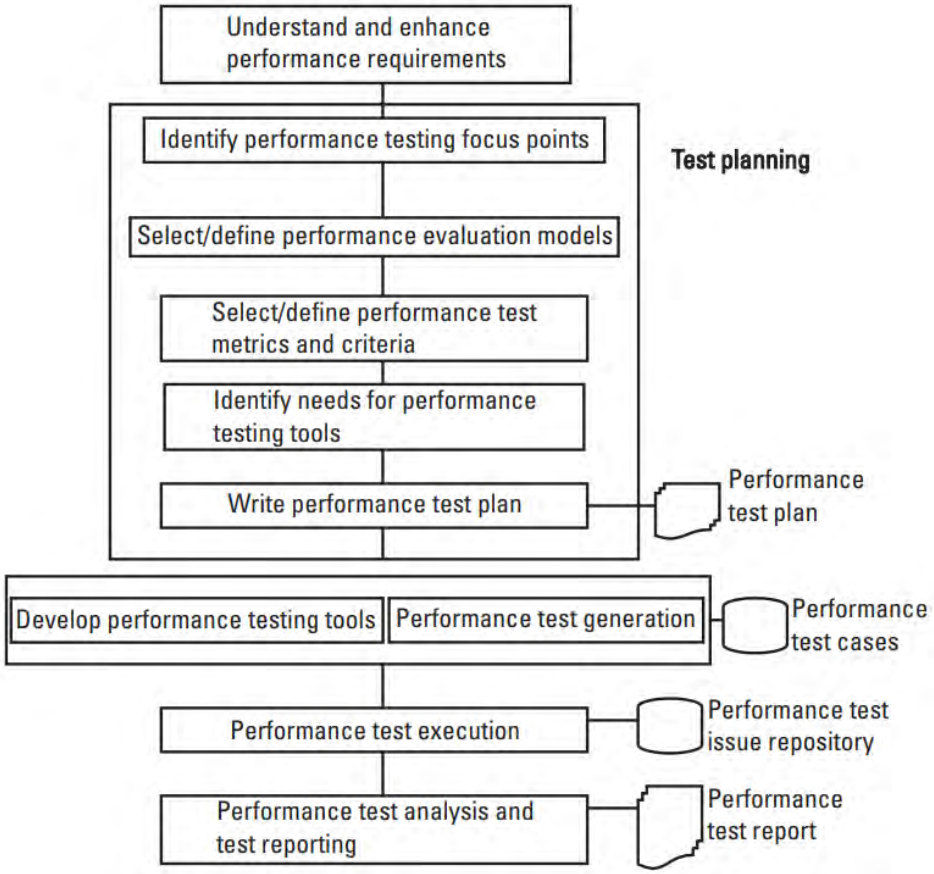

Figura 4.1: Proceso de pruebas de performance según Gao et al. [15]

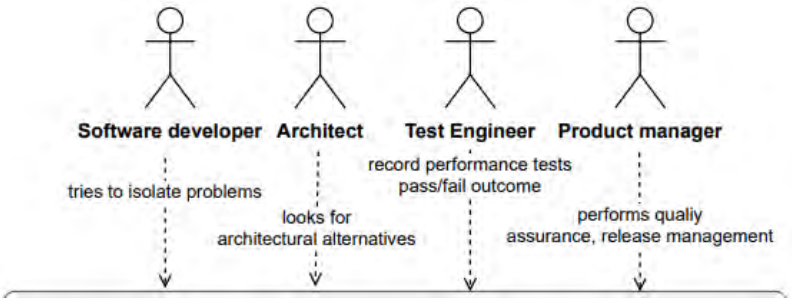

Figura 4.2: Actividades en el contexto de evaluación de performance en un proceso de desarrollo de software según Avritzer et al. [51]

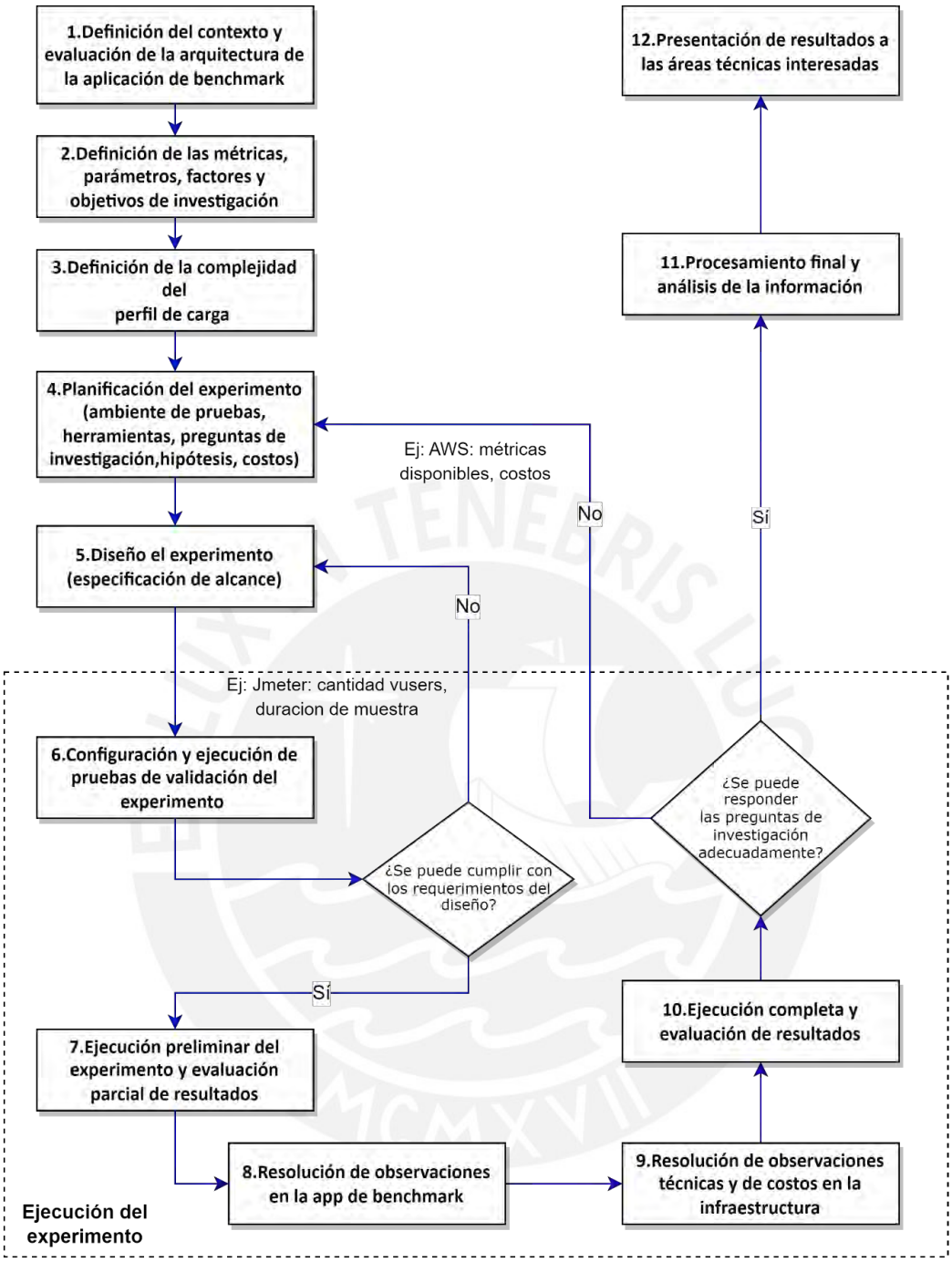

Figura 4.3: Diagrama del método propuesto

4.2 Componentes del método

En las siguientes secciones se describen los pasos de los componentes del método.

4.2.1 Definición del contexto y evaluación de la arquitectura de aplicación de *benchmark*

- Definir cuál es el alcance de las funciones de la *app* de *benchmark.*
  - Si la aplicación es de tipo *e-commerce.*
  - Si todas funcionalidades están basadas en CRUD.

    - Si existen funcionalidades de alto consumo de recursos. En este caso, realizar pruebas locales para determinar el tiempo de procesamiento.
- Definir el alcance de los requerimientos de eficiencia y confiabilidad de rendimiento(*performance*).
    - Cuáles son los tiempos de respuestas máximos esperados por el negocio.
    - Cuál es la capacidad real proyectada de usuarios.
    - Cuál es la tolerancia en % de errores.
    - Cuál es el % de disponibilidad mínima.
- Definir el alcance de los requerimientos de arquitectura de infraestructura.
    - Si la lógica de negocio es compleja de tal forma que requiera el uso del uso de patrones de orquestación de eventos.
    - Si la aplicación se va a desplegar en nube sobre tecnologías *serverless.*
    - Cuáles son los tipos de servicios de nube que serán usados en producción.

4.2.2 Definición de las métricas, parámetros, factores y objetivos de investigación

- Definir las métricas que serán evaluadas.
    - Proponer una lista de posibles métricas a medir, las cuales serán luego depuradas en las primeras pruebas de experimentación.
- Definir los parámetros a usar del proveedor de nube.
    - Revisar las configuraciones del proveedor de nube, las cuales se modificarán en los escenarios de experimentación.
- Definir los objetivos de la investigación.
    - Priorizar los objetivos de acuerdo con lo solicitado por los interesados.
    - Ajustar los objetivos de acuerdo con las limitaciones técnicas conocidas.

4.2.3 Definición de la complejidad del perfil de carga

- Analizar cuál es la distribución de carga que se busca replicar mediante la revisión de reportes de tráfico de red existentes o en base a la experiencia de otras empresas con *apps* que generan patrones de tráfico de red similares.
- Definir el tipo de configuraciones que se debería realizar en la herramienta de pruebas de carga para conseguir la distribución real de tráfico de red esperado.

4.2.4 Planificación del experimento

- Definir cómo se va a interactuar con la herramienta de despliegue, es decir quien se va a encargar de desarrollar el código.
- Definir el alcance del ambiente de pruebas.
- Definir el alcance de las herramientas.
- Especificar las preguntas de investigación.
- Especificar las hipótesis.
- Definir los límites de costos.
- Revisar la documentación actualizada del proveedor de nube para conocer las últimas opciones y límites de los servicios de nube que se utilizarán en el despliegue de la infraestructura.

4.2.5 Diseño del experimento

- Especificar en donde va a estar disponible la última versión del código fuente del *benchmark* y de la infraestructura de despliegue (asumiendo que se usa una herramienta que permita representar la infraestructura como código).
- Especificar cual será el ambiente de pruebas.
- Decidir si las pruebas se realizarán en un proveedor de nube o a nivel local, y como se realizará el monitoreo de recursos.
- Especificar las herramientas de interacción con el código fuente de *benchmark* e infraestructura.
- Especificar la herramienta de ejecución de pruebas de carga.
- Evaluar los costos que se pueden generar en base a la información obtenida de los proveedores de infraestructura y herramientas.
- Evaluar el tiempo de trabajo de las personas involucradas en las pruebas.
- Especificar los valores a configurar en las opciones de los servicios de nube.

4.2.6 Configuración y ejecución de pruebas de validación

- Configurar las herramientas de despliegue de infraestructura y *benchmark.*
- Configurar las herramientas de pruebas de carga.
- Configurar el ambiente de pruebas.
- Desplegar la infraestructura de *benchmark.*
- Configurar la herramienta de pruebas de carga en el ambiente local.
- Desplegar la herramienta de pruebas de carga en el ambiente final de pruebas.
- Validar a nivel local que se puede cumplir con los objetivos del experimento.
- Realizar pruebas de carga con la mayor cantidad de usuarios en simultaneo y en el menor tiempo de duración.
- En caso se encuentren problemas en la ejecución retornar a la etapa de diseño del experimento para realizar o gestionar los ajustes que permitan la solución de las observaciones.

4.2.7 Ejecución preliminar del experimento y evaluación de resultados

- Definir valores cercanos al escenario real de experimentación.
    - Por ejemplo, si el objetivo es probar 50 usuarios virtuales en simultáneo, probar con el 50% o con la misma cantidad, pero con menor tiempo de duración total de pruebas.
- Monitorear el consumo de recursos de la infraestructura desplegada.
- Monitorear el consumo de costos.
- Monitorear el uso de recursos del ambiente de pruebas.
- Analizar, a nivel estadístico, los resultados automatizando las pruebas en cuanto sea posible.
- Realizar los ajustes necesarios para cumplir con los objetivos de validación estadística.
    - Por ejemplo, la cantidad de muestras y repeticiones de la experimentación.
- Reportar las observaciones de generación de errores y costos a quienes corresponda.

4.2.8 Resolución de observaciones en la app de *benchmark*

- Gestionar la resolución de observaciones de la evaluación preliminar de *performance con el* equipo de desarrollo.

4.2.9 Resolución de observaciones técnicas y de costos en la infraestructura

- Mientras se espera la respuesta del equipo de desarrollo, eliminar temporalmente la arquitectura desplegada mientras se resuelven las observaciones.
- Retomar la ejecución de las pruebas cuando los ajustes y la solución sea confirmada.

4.2.10 Ejecución completa del experimento y evaluación de resultados

- Desplegar las configuraciones finales.
- Validar el funcionamiento de los monitoreos.
- Ejecutar todas las validaciones estadísticas.

4.2.11 Procesamiento final y análisis de la información

Generar gráficos estadísticos y tablas a partir de los datos obtenidos para que se pueda realizar el análisis detallado de los resultados por parte de otras áreas.

4.2.12 Presentación de resultados a las áreas técnicas interesadas

Elaborar presentaciones de los resultados a los actores interesados en la ejecución de las pruebas de performance.

4.3 Diseño del sistema de ejecución del experimento

Para brindar un sistema eficiente de experimentación se presenta el proceso de la figura 4.5, que ha sido elaborado considerando como principal referencia a la figura 4.4 obtenida de [5].

Los componentes de este proceso se muestran en la tabla 4.2, en los cuales se visualiza el mapeo de las operaciones propuestas con las operaciones del *workflow* según [5] y con las actividades realizadas en el experimento. Las operaciones complementarias, que no están presentes en [5], se han propuesto en base a lo aprendido en la ejecución de la experimentación.

Tabla 4.2: Mapeo de las recomendaciones consideradas en la propuesta del método

Operación propuesta	***Workflow* según** Baruwal et al. [5]	**Tareas realizadas en la experimentación**
Ejecución del SUT	Proveedores para el SUT	Despliegue de la infraestructura FaaS y CaaS
Ajustes en SUT	Configuraciones del *benchmark*	Validación de configuraciones en consola de AWS ejecutadas por CDK
Monitoreo de recursos del SUT	Inicio de monitoreo y final de monitoreo	Monitoreo de recursos de FaaS y CaaS mediante Cloudwatch
Monitoreo de costos	No está presente	Visualización de *dashboard* y tablas de *billing*, *free tier*, costos

Obtención de última versión de *benchmark*	Selección de *benchmark*	Mediante CDK se accede a la ruta de los archivos zip de código fuente para FaaS o mediante AWS CLI se sube las imágenes Docker al ECR para CaaS
Ejecutor de despliegue de *benchmark* en arquitecturas	Obtener instancias y ejecutar *benchmark*	Mediante CDK se crean las configuraciones de AWS que permiten la creación y el auto escalamiento de Lambda o Fargate
Realizar cambios sobre el despliegue para comparaciones	Configuraciones del *benchmark*	Mediante CDK se realizan cambios que luego se despliegan mediante su CLI
Eliminar despliegue de infraestructura	Finalizar instancias	Mediante CDK se puede eliminar la infraestructura FaaS o CaaS mediante su CLI
Configuración y despliegue de herramienta de pruebas de carga	Especificación de configuración de pruebas	Las configuraciones de la herramienta se hacen a nivel local, se exportan como archivos y luego se suben manualmente al ambiente de pruebas
Ejecución de herramienta de pruebas de carga	Obtener instancias	Se configura la ejecución automática de los archivos que contienen las configuraciones de pruebas de carga
Monitoreo de uso de recursos de herramienta de pruebas de carga	No está presente	Se monitorea el uso de recursos del ambiente de pruebas para garantizar que no haya interrupciones en la ejecución
Recolección de reportes	Agrupación de resultados	Se descargan los reportes generados por las herramientas en formato CSV

Generación de reportes	Generación de reporte	Se utiliza las herramientas de procesamiento de datos y análisis estadísticos con los archivos CSV

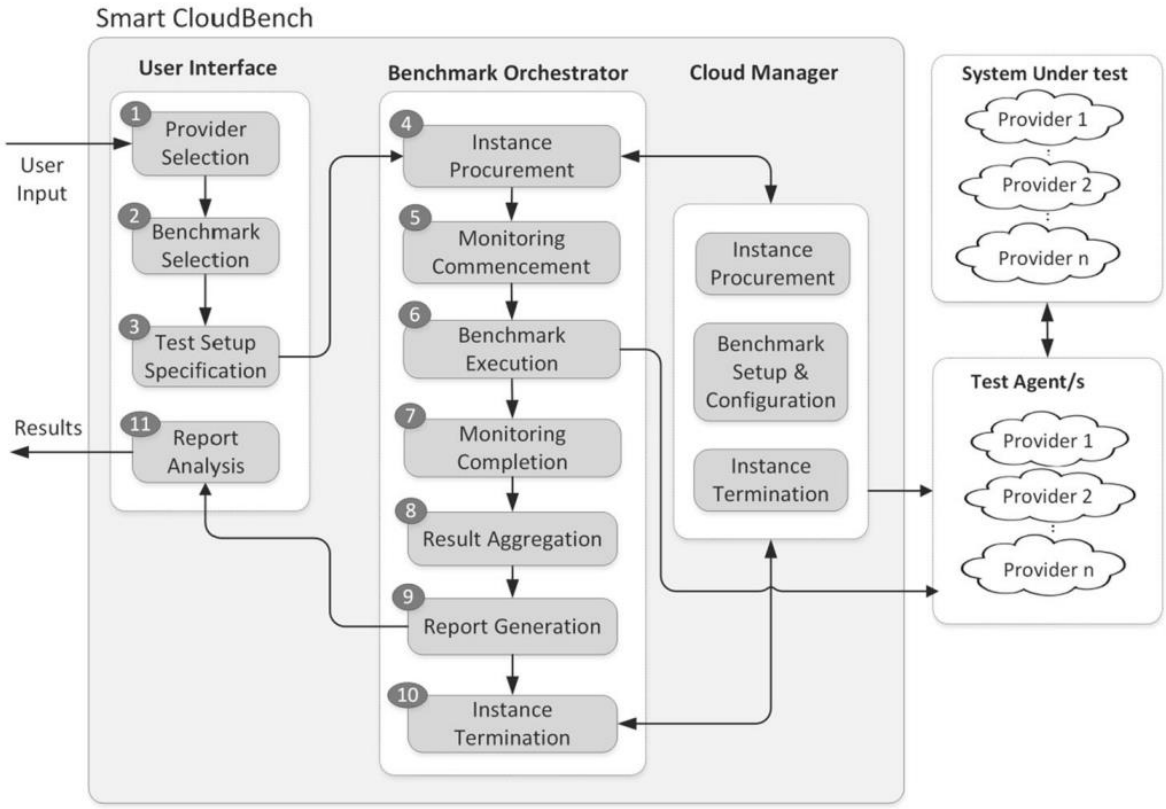

Figura 4.4: Workflow del framework cloudbench presentado en Baruwal et al. [5]

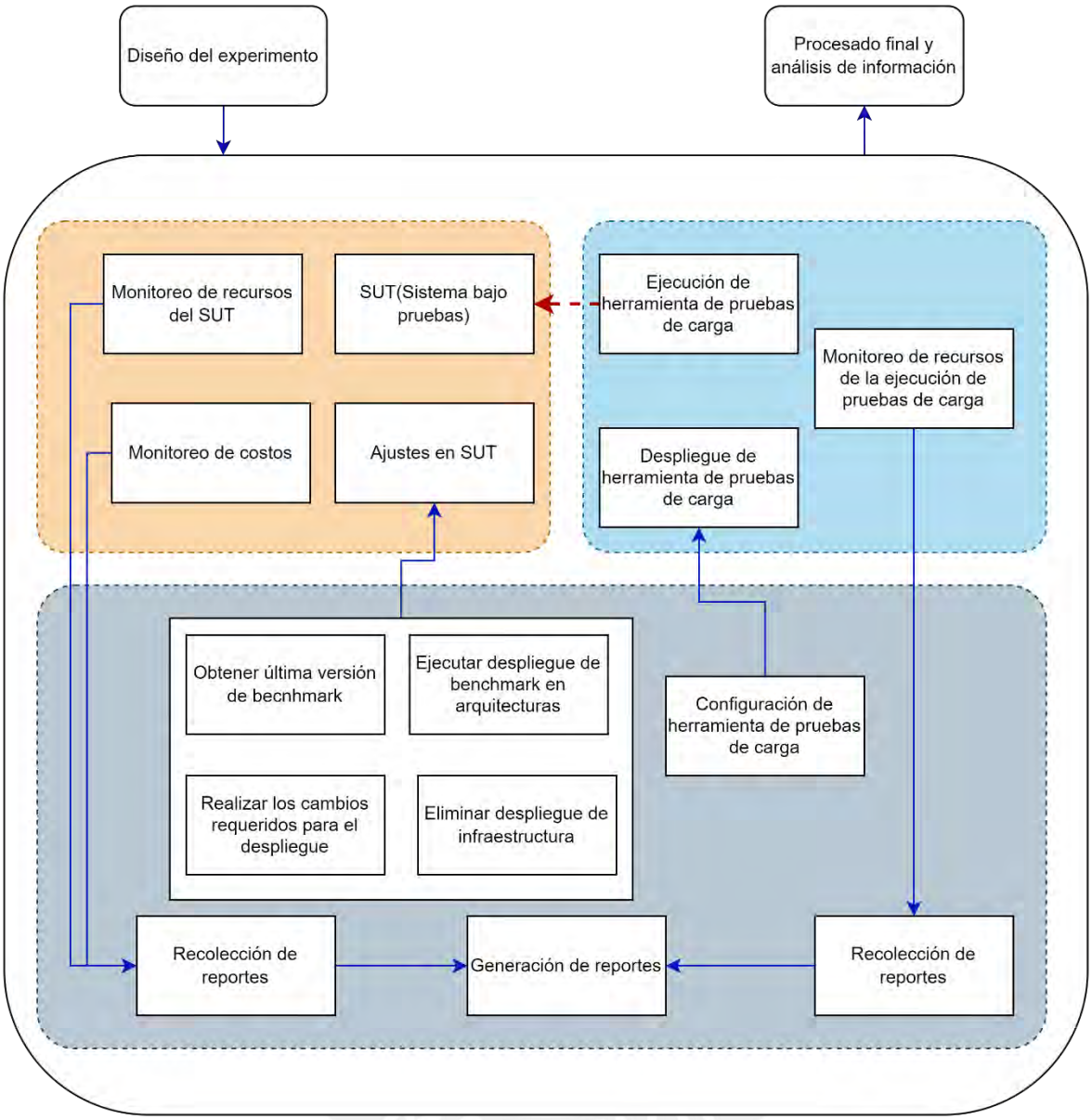

Figura 4.5: Diseño de operaciones del sistema de ejecución de performance

5 PLANEAMIENTO DEL EXPERIMENTO

En esta sección se presenta la planificación y diseño del experimento elaborado en base a lo indicado en la sección de propuesta de método de evaluación.

5.1 Planificación del experimento

5.1.1 Objetivos

O1: Validar que las etapas del proceso incluyen las tareas necesarias para completar las pruebas.

O2: Definir cuál de las 2 alternativas de despliegue de infraestructura es conveniente a nivel de *performance* y costos.

O3: Realizar los experimentos al menos durante 3 días. Cada día se elige un nivel de configuración de memoria diferente.

5.1.2 Preguntas de investigación

RQ1: ¿Cuál es la diferencia en tiempo de respuesta total entre las tecnologías *serverless* FaaS y CaaS para APIs con tiempo de ejecución menor a 1 segundo?

RQ2: ¿Cuál es la diferencia en porcentaje de errores de invocación de funciones entre las tecnologías *serverless* FaaS y CaaS para APIs con tiempo de ejecución menor a 1 segundo?

RQ3: ¿Cuál es la diferencia en tiempo de respuesta total entre las tecnologías *serverless* FaaS y CaaS para APIs con tiempo de ejecución de varios segundos?

RQ4: ¿Cuál es la diferencia en porcentaje de errores de invocación de funciones entre las tecnologías *serverless* FaaS y CaaS para APIs con tiempo de ejecución de varios segundos?

RQ5: ¿Cuál es la diferencia en costos entre las tecnologías *serverless* FaaS y CaaS?

RQ6: ¿Cuál es la diferencia en el patrón de escalabilidad entre FaaS y CaaS?

5.1.3 Instrumentos del experimento

- Se utilizará el software RStudio para el procesamiento estadístico.
- Se utilizará el software JMeter para la ejecución de pruebas de carga.
- Se usará la nube de Google para el despliegue de las configuraciones de JMeter.

5.1.4 Parámetros y variables

- Métricas: Tiempo de respuesta total, tiempo de ejecución, escalabilidad, costos.
- Variable dependiente: Tiempo de respuesta total, tiempo de ejecución, número de errores de invocación, costos.
- Factores y variables dependientes: Niveles de intensidad de *workload*, niveles de configuración de memoria.
- Parámetros y constantes: Acoplamiento de arquitectura, configuración de capacidad máxima en tiempo y conexiones simultáneas.

5.1.5 Hipótesis de investigación

Lista de hipótesis para las 4 APIs con tiempo de ejecución menor a 1 segundo:

H1: La media del tiempo de respuesta total de FaaS es menor que la de CaaS en cada nivel de configuración de memoria de las 4 APIs.

H2: La media del porcentaje de errores por falta de recursos en la infraestructura de despliegue del tipo CaaS es menor que la de FaaS en cada nivel de configuración de memoria de las 4 APIs.

H3: La media del costo total de FaaS es menor que la de CaaS en cada nivel de configuración de memoria de las 4 APIs.

H4: La cantidad de nuevas instancias creadas crece con mayor rapidez en FaaS que en CaaS en cada nivel de configuración de memoria de las 4 APIs.

Lista de hipótesis para la API con tiempo de ejecución de varios segundos:

H5: La media del tiempo de respuesta total de FaaS es menor que la de CaaS en cada nivel de configuración de memoria de la API.

H6: La media del porcentaje de errores por falta de recursos en la infraestructura de despliegue del tipo CaaS es menor que la de FaaS en cada nivel de configuración de memoria de la API.

H7: La media del costo total de FaaS es menor que la de CaaS en cada nivel de configuración de memoria de la API.

H8: La cantidad de nuevas instancias creadas crece con mayor rapidez en FaaS que en CaaS en cada nivel de configuración de memoria de la API.

5.2 Diseño del experimento

5.2.1 Parámetros y variables

Las variables dependientes son los niveles de configuración de memoria y de vCPU, y la variable dependiente es el tiempo de respuesta. El valor constante en cada prueba serán los 50 usuarios virtuales. Y la cantidad máxima de usuarios en solicitudes en simultaneo será 10.

5.2.2 Perfil de *workload*

El perfil de *workload* se basa en la simulación de un usuario virtual que interactúa con las 5 APIs: búsqueda de vuelos, selección de vuelo, registro de participación, generación de *ticket*, búsqueda de resultados. Lo cual se aplica para 50 usuarios virtuales con periodos de prueba de 10 minutos, de modo que en cada periodo se puede ejecutar varias interacciones con las 5 APIs, y de esta forma generar al menos 50 experimentos en cada muestra por cada API.

Se debe considerar periodos entre 10 y 30 segundos como *delay* para la ejecución de una API, de esta forma se simula el tiempo que demoraría un usuario si interactuará con el *frontend* en un caso real.

Durante el tiempo de *delay*, se debe considerar que aparecen 2 nuevos usuarios cada 10 segundos, tanto al inicio como al final del experimento, de esta forma se simula un crecimiento de usuarios a velocidad moderada, que representa un caso real.

5.2.3 Herramienta de ejecución de pruebas de carga

Se usará la herramienta JMeter, la cual se configurará por GUI.

Se exportará la configuración en archivos JMX, que luego serán subidos al ambiente de prueba para que se ejecuten automáticamente por CLI.

El despliegue final será de en Google *Cloud* sobre una VM de Linux, el cual enviará tráfico hacia las URLs de las APIs FaaS y CaaS desplegadas en AWS.

5.2.4 Configuraciones principales de la herramienta que ejecuta las pruebas de carga

Para las pruebas oficiales, se elaborará una lista en CSV de 50 usuarios para que puedan ser cargados en la herramienta de pruebas de carga.

Se validará si 10 minutos es un tiempo suficiente para generar al menos 50 experimentos por muestra por cada API, ya que el tiempo de ejecución de cada API es de menos de 1 segundo, a excepción del generador de *ticket.* Además, los *delays* entre APIs duran alrededor de 15 segundos.

Para la pruebas preliminares y oficiales se configurará una cantidad máxima de 10 usuarios virtuales en simultáneo, debido a restricciones de AWS. En consecuencia, las solicitudes HTTP de los 50 usuarios virtuales se distribuirán con ese límite de 10 en simultáneo.

5.2.5 Ejecución de pruebas de carga

Cada prueba durará 1 hora, entonces durante el día se pueden realizarán hasta 12 pruebas desde las 9am para representar los 3 periodos del día (mañana, tarde, noche).

Las pruebas se realizarán durante los 7 días de una semana, obteniéndose hasta 84 pruebas equivalentes-muestras. Serán ejecutadas en simultaneo entre las 9am y 9pm y en caso no haya errores la diferencia de pruebas será de máximo 1 hora.

Además, las pruebas serán programadas para ser ejecutadas automáticamente durante los 3 días, sin embargo, se podría hacer modificaciones antes del inicio o después del final del experimento diario si se encuentran anomalías luego de analizar los resultados preliminares o si se encuentra incidentes en el monitoreo de uso de recursos de la VM de Linux.

Para la evaluación estadística se usará un rango de tiempo menor a 12 horas para descartar los eventos que podrían alterar el experimento al inicio y fin de cada día.

5.2.6 Herramienta de despliegue de infraestructura

Se usará el *framework* CDK de AWS para el despliegue automático de la infraestructura FaaS y CaaS.

5.2.7 Monitoreo de recursos y costos

El monitoreo de recursos de AWS consistirá en la validación del funcionamiento del auto-escalamiento del Fargate y los Lambdas mediante CloudWatch.

Se realizará el monitoreo de costos y uso de *free tier* mediante la consola de AWS, además del consumo de recursos de memoria y CPU de la máquina virtual Linux que está ejecutando los archivos JMX.

5.2.8 Interrupciones de pruebas ante observaciones de performance y costos

Se interrumpirá las pruebas en los siguientes 3 casos:

Cuando se encuentren errores en el uso de funcionalidades de la app de benchmark, se corregirá el código fuente y se volverá a desplegar tanto para FaaS como CaaS.

En caso se genere un alto porcentaje de errores de *timeout* en una de las infraestructuras, se revisará si se puede corregir mediante algún ajuste en las opciones de configuración de AWS, ya que principalmente se busca comparar cuál de las 2 opciones ofrece mejores tiempos de respuesta sin que se supere el *timeout* de 30 segundos.

Finalmente, si los costos reales (mostrados en la consola de AWS al día siguiente) u obtenidos por fórmula son de varios dólares por día.

5.2.9 Fórmulas de costos de AWS

A continuación, se describe los datos que serán necesarios para el uso de las 2 fórmulas planteadas para el cálculo del costo, una para FaaS y la otra para CaaS, esta última será más compleja.

En la tabla 5.1 se muestran los datos necesarios obtenidos de [52]–[55] para los 4 servicios de referencia para la evaluación de costos del experimento de AWS elegidos en la región EE. UU. Este (Norte de Virginia).

Hay que considerar que el término propuesto “por existencia” significa que se cobra el servicio por estar activo sin importar si es usado. Además, no se está considerando los costos de los *buckets* S3, API Gateway, logs de Cloudwatch por que la capa de *free tier* ofrece lo suficiente para las pruebas.

Tabla 5.1: Costos oficiales de AWS

Nombre de servicio	Tipo de costo	Costo
Lambda	Por duración	0,0000166667 USD por cada GB/segundo
Lambda	Por solicitudes	$0.20 por cada millón de solicitudes
Fargate	Por CPU por hora	$0.04048
Fargate	Por GB por hora	$0.004445
VPC PrivateLink	Por VPC endpoint por hora	$0.01
VPC NAT Gateway	Por existencia por hora	$0.045
VPC NAT Gateway	Por GB procesado	$0.045
DynamoDB	Por existencia para escritura por hora	$0.00065 per WCU (write capacity unit)
DynamoDB	Por existencia para lectura por hora	$0.00013 per RCU (read capacity unit)

Se considera que cada solicitud es un experimento, el cual se repite varias veces durante 10 minutos, luego se espera a que empiece la siguiente hora para realizar volver a realizar el experimento, entonces se considera que se genera 1 muestra de cada API en 10 minutos.

En las fórmulas de Lambda y Fargate se usará el término proporción de memoria, el cual se calcula dividiendo la memoria seleccionada entre 1024MB.

5.2.9.1 Fórmula y ejemplo de cálculo en FaaS

Las fórmulas para el cálculo final del tipo de costo de cada servicio son:

- Cargo de lambda por duración en cada muestra = costo de lambda por duración × tamaño de 1 muestra × tiempo medio de cada API x número de APIs equivalentes × proporción de memoria.
- Cargo de lambda por solicitudes en cada muestra = costo de lambda por solicitudes × tamaño de 1 muestra × número de APIs equivalentes × proporción de memoria.

Por ejemplo, si se considera que se tiene 5 APIs con características equivalentes a los datos del experimento. Entonces para el despliegue sobre una infraestructura FaaS con un Lambda de 512MB se aplica los costos de los recursos de la tabla 5.1, los datos de uso de servicios *cloud* de la tabla 5.2 y las fórmulas indicadas arriba. Por lo tanto, luego de la suma del cargo de lambda por duración en cada muestra con el cargo de lambda por solicitudes en cada muestra, se obtiene un costo por el total de la experimentación de 0.2986672 centavos de dólar en 1 hora.

Tabla 5.2: Uso de servicios de AWS FaaS en una muestra de ejemplo

Nombre de variable	Valor
Solicitudes por cada experimento de una API en una hora (tamaño de 1 muestra)	70
Tiempo medio de cada API en ms	1000

5.2.9.2 Fórmula y ejemplo de cálculo en CaaS

A continuación, se muestran las fórmulas de costos de los componentes Fargate y VPC de una infraestructura CaaS.

5.2.9.2.1 Costos de Fargate

- Cargo de Fargate por uso de CPU en cada muestra = costo de Fargate por uso de CPU × tamaño de 1 muestra × tiempo medio de cada API x número de APIs equivalentes × cantidad de vCPU × número de tareas Fargate activas.
- Cargo de Fargate por uso de memoria en cada muestra = costo de Fargate por uso de memoria × tamaño de 1 muestra × tiempo medio de cada API x número de APIs equivalentes × proporción de memoria × número de tareas Fargate activas.
- Cargo de Fargate por existencia del CPU en 1 hora continuamente en cada muestra = costo de Fargate por existencia de CPU en 1 hora × cantidad de vCPU × número de tareas Fargate activas.
- Cargo de Fargate por uso de memoria en 1 hora continuamente en cada muestra = costo de Fargate por existencia de memoria en hora × proporción de memoria × número de tareas Fargate activas.

5.2.9.2.2 Costo de VPC

- Cargo de los VPC *endpoints* en cada muestra = costo de VPC *endpoint* por uso por hora× número de VPC *endpoints.*
- Cargo del NAT Gateway por existencia en cada muestra = costo del NAT Gateway por existencia en 1 hora.
- Cargo del NAT Gateway por datos en GB usados en cada muestra = costo del NAT Gateway por datos en GB usados en 1 hora.

Por ejemplo, si se considera que se tiene 5 APIs con características equivalentes a los datos del experimento. Entonces para el despliegue sobre una infraestructura CaaS se aplica los costos de los recursos de la tabla 5.1, los datos de recursos elegidos de la tabla 5.3 y las fórmulas indicadas. Por lo tanto, luego de la suma de los cargos de Fargate con los cargos de VPC se obtiene un costo total de experimentación de 8.575983 centavos de dólar en 1 hora.

Tabla 5.3: Recursos elegidos para AWS CaaS de ejemplo

Nombre	Tipo	valor
vCPU	Fargate	1vCPU
Memoria	Fargate	2048MB
Número de tareas Fargate	Tareas Fargate	1.2 (promedio)
Número de VPC enpoints	VPC endpoint	8

5.2.10 Análisis estadístico

- Se utilizará la prueba estadística no paramétrica Wilcoxon sum rank *test* con un intervalo de confianza de 95%.
- Se ejecutará las pruebas de Wilcoxon mediante la opción two-sided para comparar solamente si existe una diferencia significativa en la distribución de medias, luego para decidir cuál de las 2 opciones supera a la otra, se revisarán los valores de las medias estadísticas obtenidas por el *script* de procesamiento estadístico, cuyos datos estarán presentados en tablas.
- Un valor mayor a 0.5 para el *p-value* en la prueba de Wilcoxon indica que no hay diferencia significativa entre las medias, por lo tanto, se puede considerar iguales.
- Se procesarán los resultados mediante la herramienta RStudio.

6 IMPLEMENTACIÓN DE COMPONENTES DEL EXPERIMENTO

En esta sección se presentan las interfaces desarrolladas para la aplicación de benchmark y los diagramas iniciales y finales de las arquitecturas de FaaS y CaaS. Así como el diagrama final del sistema de ejecución de pruebas.

6.1 Desarrollo de aplicación transaccional de benchmark

Se ha refactorizado la aplicación demo de AWS llamada SAB (Serverless airline booking) publicada en [56] para usarla como *benchmark* de comparación en las pruebas.

6.1.1 Software original

La aplicación demo originalmente está implementada para que un usuario se autentique mediante AWS Cognito, luego seleccione un vuelo y realice el pago para poder reservarlo. A nivel de *backend,* se implementan APIs con funciones Lambda comunicándose entre ellas mediante un AWS *Step Functions* para realizar operaciones CRUD (*create, read, update, delete*) mediante GraphQL sobre una base de datos NoSQL. El código fuente se despliega mediante la herramienta AWS Amplify.

A continuación se muestran las imágenes originales de la aplicación en la figura 6.1 y de la arquitectura original en la figura 6.2.

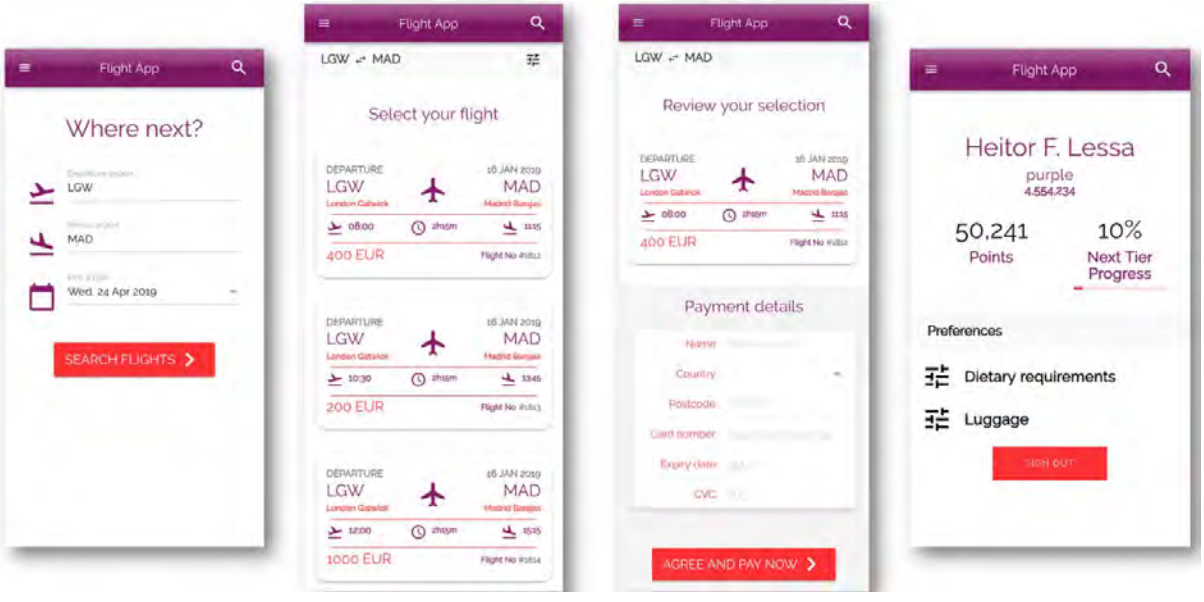

Figura 6.1: Pantallas originales de frontend del software demo (Serverless airline booking) publicado en [56]

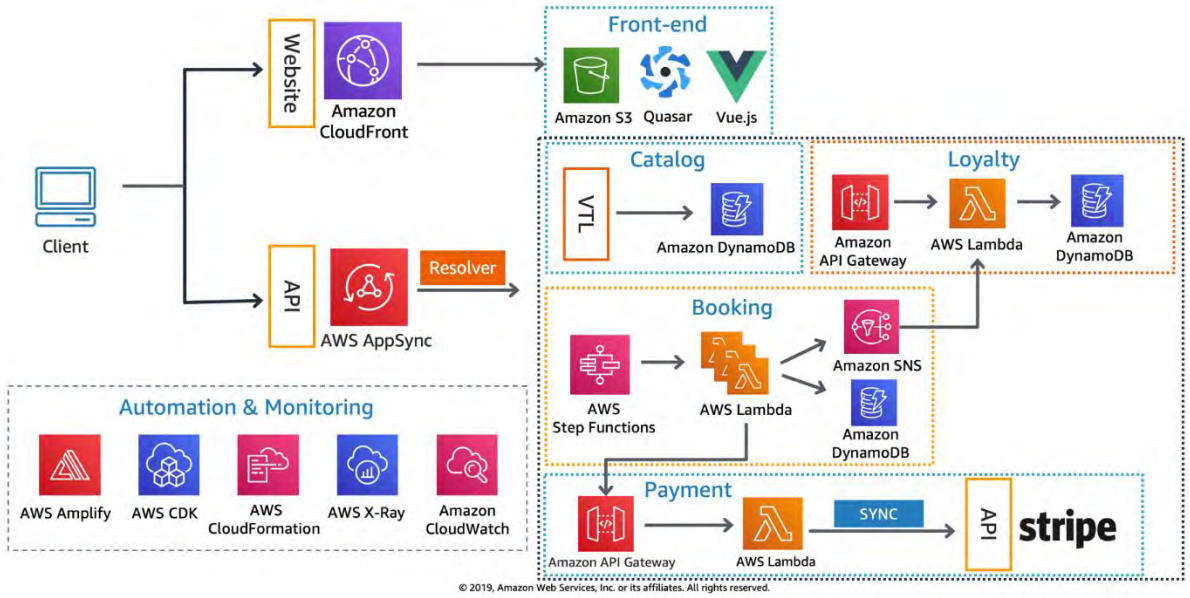

Figura 6.2: Arquitectura original del software demo original (Serverless airline booking) publicado en [56]

6.1.2 Modificaciones al software original

La aplicación ha sido modificada para que la reserva de vuelos se realice sin necesidad de autenticación, el usuario solo tiene que ingresar su nombre y fecha de nacimiento. Este uso de la aplicación se puede interpretar como una reserva preliminar para obtener *tickets* de avión.

Específicamente, se decidió realizar las siguientes modificaciones:

- Se rediseñó la arquitectura para que tuviera la menor cantidad de servicios de AWS, por lo tanto, se descartó el uso de una arquitectura basada en GraphQL, ya que su implementación óptima requiere del servicio AppSync. En lugar de GraphQL se usó REST usando las operaciones *create*, *read (one y all)*, *delete* mediante 4 APIs.
- Se descartó el uso de los servicios SQS y *Step functions.*
- Se ha creado un nuevo tipo de API denominado *ticket create one* que permite la generación de un ticket por solicitud, el cual es mostrado al usuario cuando se finaliza la ejecución como se muestra en la figura 6.7. Para esta API se ha agregado código de programación que realiza una operación de Fibonacci, de tal forma que el total de operaciones matemáticas de la API demoren más de 10 segundos.
- En total son 5 APIs con los nombres: *flights read all, flights read one, booking write, booking read all* y *ticket create one* que se desplegarán en las arquitecturas FaaS y CaaS como se muestran en las figuras 5.9 y 5.11 respectivamente.
- La API *flights read all* se encarga de leer toda la lista de vuelos disponibles de la BD *flights* y muestra los resultados en la primera pantalla del *frontend*. En la figura 6.3 se muestra una selección de origen y destino de vuelo.
- La API *flights read one* se encarga de obtener los datos del vuelo seleccionado en el *frontend* de la BD *flights* como se muestra en la figura 6.4.
- La API *booking write* se invoca al confirmar la inscripción en el vuelo como se indica en la figura 6.5, esta API agrega el usuario a la BD y verifica que no se haya registrado, si lo estuviera, muestra un mensaje de error como se muestra en la figura 6.6.
- El API *booking read all* permite la lectura de los 50 usuarios registrados como se muestra en la figura 6.8.
- Por otro lado, se decidió conservar el *framework* de desarrollo de *frontend* Vue debido que facilitaba la modificación de la interfaz de usuarios.

- Las pruebas de performance se realizaron contra las URLs de las APIs (*backend*) sin usar las URLs de páginas web (*frontend*), ya que cuando se realizaron las primeras pruebas de *performance* contra las URLs de las páginas web, se identificó que algunas dependían de las URLs de las APIs por lo cual se hubiera generado pruebas redundantes e incompletas.
  Se completó el desarrollo de la interfaz gráfica del *frontend* para validar manualmente que se cumple con el diseño a nivel de negocio y posteriormente las pruebas de *performance* se realizaron mediante automatización usando las URLs de las APIs.

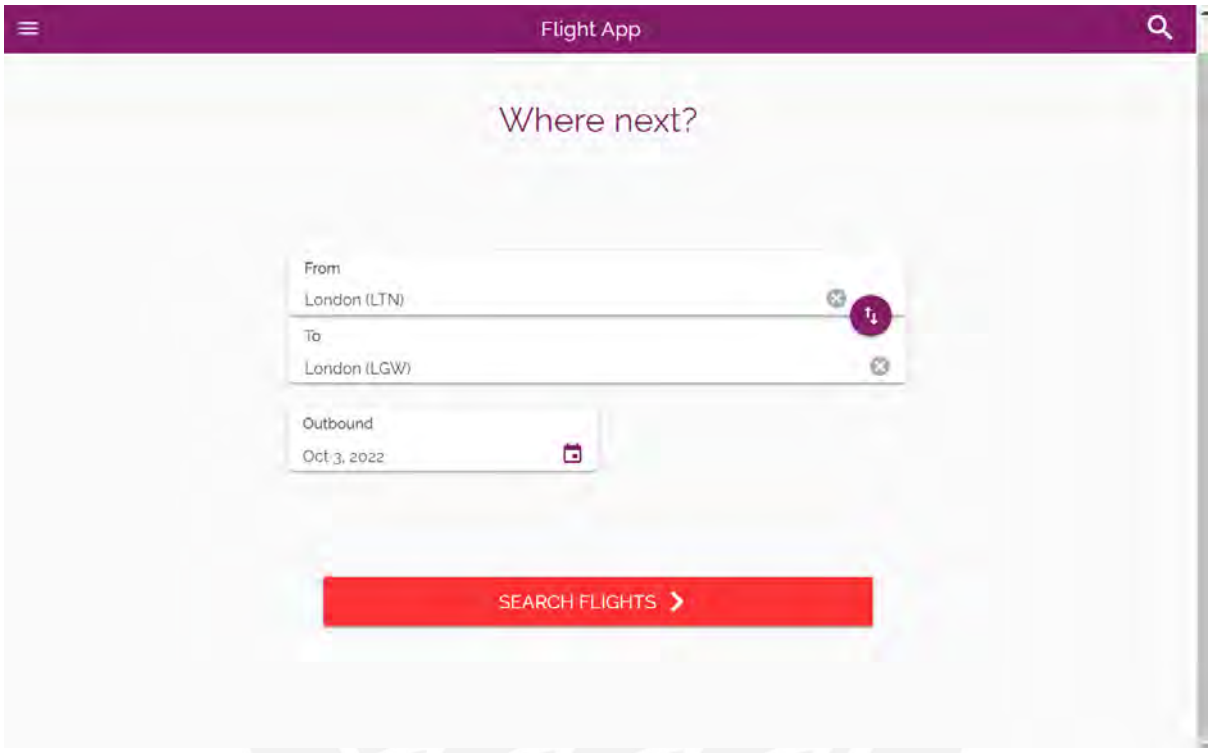

Figura 6.3: Primera pantalla que muestra los datos obtenidos por el API flights read all cuando se ha seleccionado como origen a LTN y destino a LGW

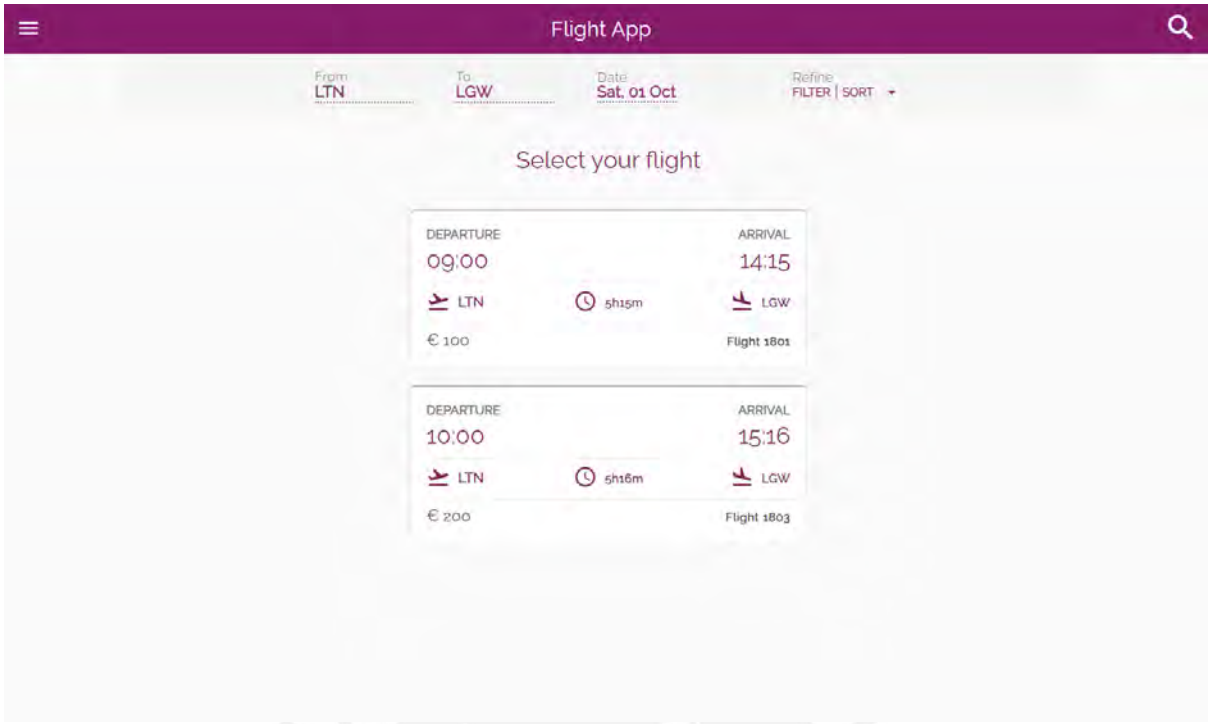

Figura 6.4: Pantalla para seleccionar un vuelo, lo cual invoca el API flights read one

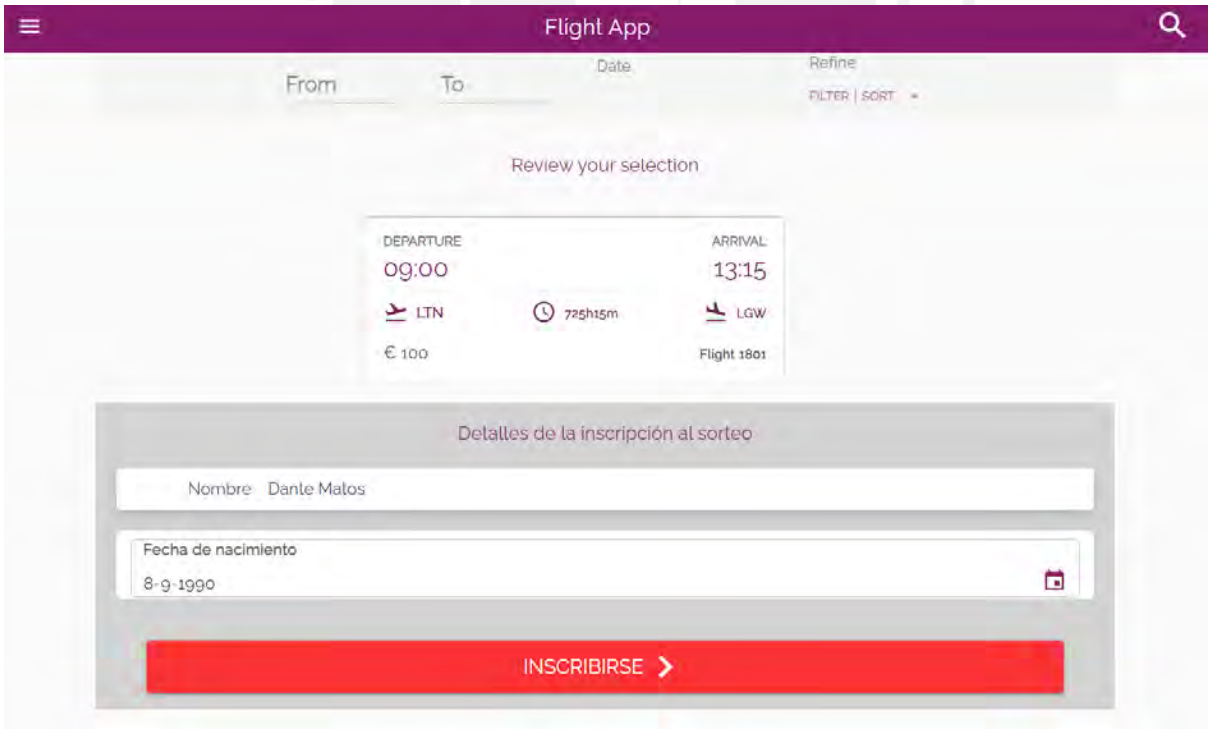

Figura 6.5: Pantalla para completar datos y dar click en el botón inscribirse que invoca el API booking write

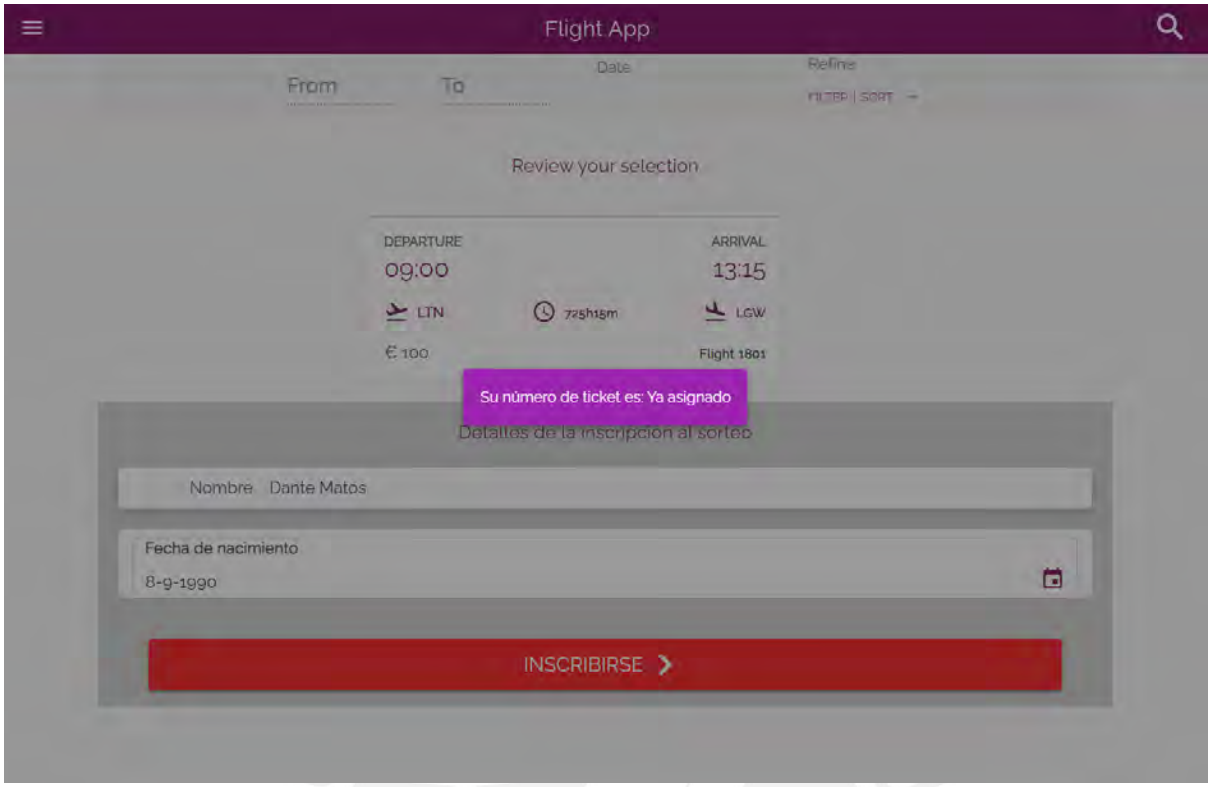

Figura 6.6: Pantalla mostrando mensaje de error personalizado cuando el API booking write valida que el usuario ingresado ya existe

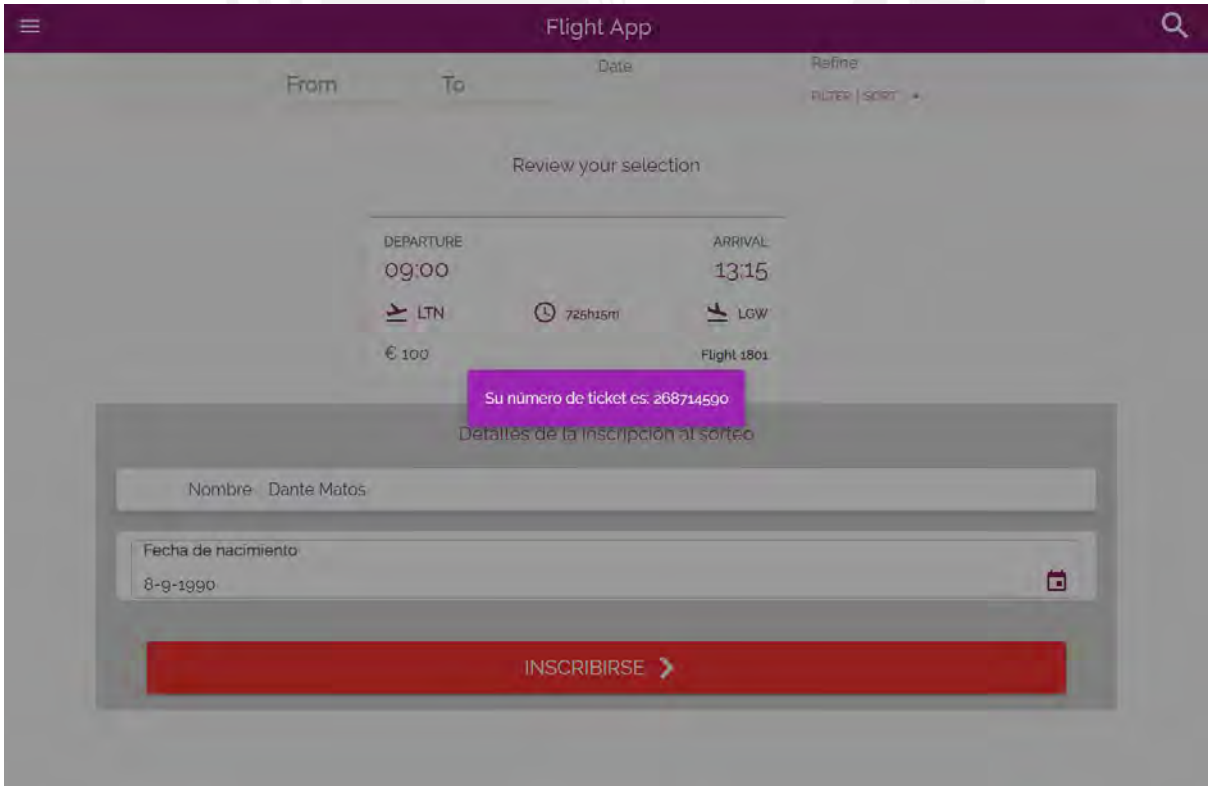

Figura 6.7: Pantalla con un número de ticket generado por el API ticket create one

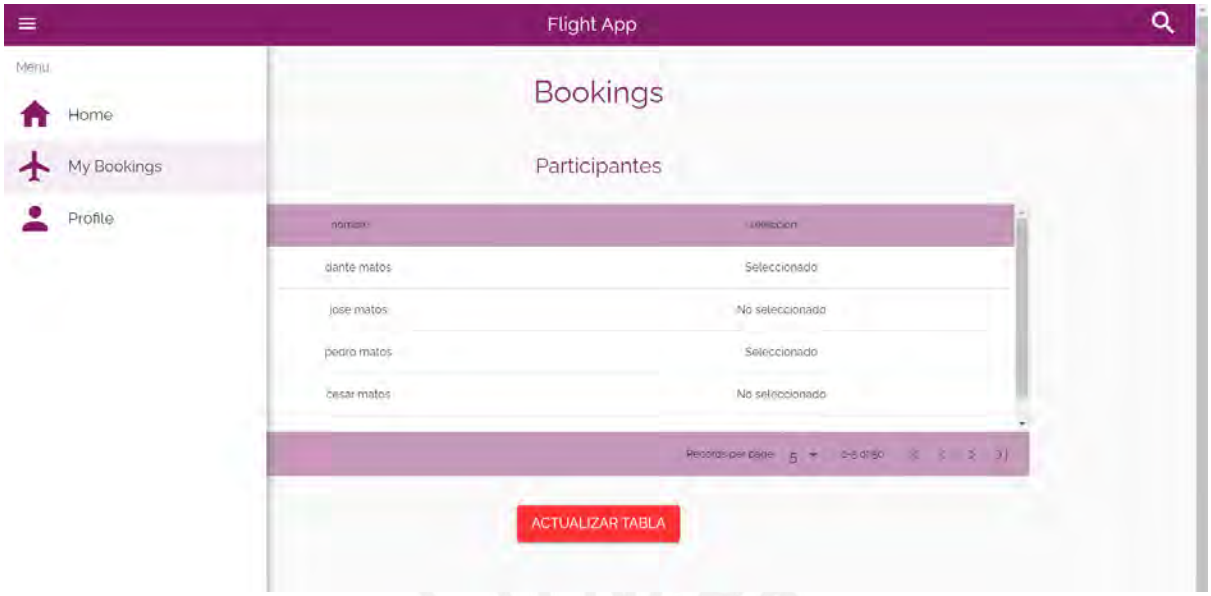

Figura 6.8: Pantalla mostrando los 50 usuarios con su estado selección generado por el API booking read all

6.2 Despliegue de infraestructura en la nube AWS

Se ha generado los componentes de la infraestructura usando principalmente la herramienta AWS CDK, y algunas configuraciones se realizaron directamente en la consola de AWS.

Por otro lado, se ha tomado de referencia algunas configuraciones y código creado por la herramienta de AWS Amplify, que inicialmente fue la principal opción para automatizar el despliegue pero se descartó, ya que se observó en las pruebas que genera configuraciones innecesarias para los objetivos del experimento.

6.2.1 Términos técnicos de AWS

1. CDK: Software *open-source* que permite la creación de recursos mediante lenguajes de programación ya conocidos por los desarrolladores como JavaScript y Python.

2. API Gateway: Permite la publicación de APIs. Normalmente son APIs del tipo REST expuestos a Internet.

3. NLB (*network load balancer*): Balanceador de carga de red que se ejecuta sobre recursos ECec2 (servicio de máquina virtual). El enrutamiento se basa en el protocolo TCP.

4. ALB (*aplication load balancer*): Balanceador de carga de aplicaciones que se ejecuta sobre recursos EC2. El enrutamiento se basa en el protocolo HTTP.

5. Security Group: Permite el control de tráfico de entrada y salida entre recursos de AWS mediante reglas de acceso basado en subredes.

6. VPC: Permite el aislamiento de internet de varios recursos de AWS. Normalmente, los recursos basados en EC2 se configuran dentro de una VPC.

7. DynamoDB: Base de datos NoSQl de AWS que no se encuentra dentro de una VPC.

6.2.2 Despliegue en FaaS

Las 5 APIs se desplegarán en Lambdas diferentes y serán publicadas a través de un API-Gateway.

Se muestra el diagrama de la arquitectura en la figura 6.9.

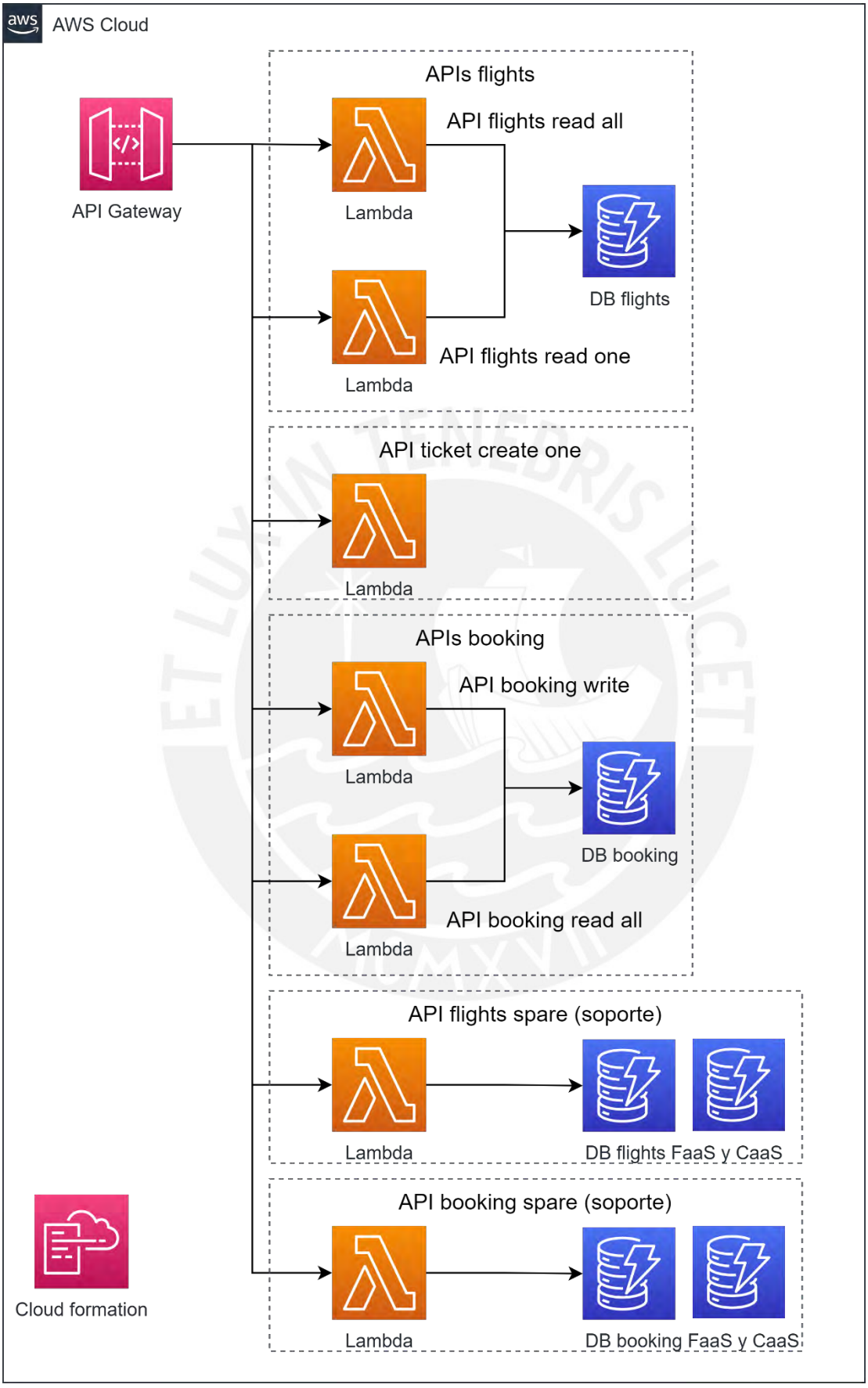

Figura 6.9: Arquitectura FaaS

6.2.3 Optimización de CaaS

Inicialmente se había planificado hacer el despliegue con la arquitectura de la figura 6.10, sin embargo se evidenció costos considerables en las pruebas, específicamente, se estaba cobrando por cada VPC NAT Gateway en cada Fargate, el cual tiene un costo de existencia de $0.045, entonces, se buscó la forma de optimizar y se identificó que usando un VPC *endpoint* se deja de usar el Internet Gateway. Por lo tanto, como los VPC *endpoint* tienen un costo de existencia (costo del servicio por estar activo sin considerar si está siendo usado) de $0.01, se puede reducir los costos al menos en la cuarta parte.

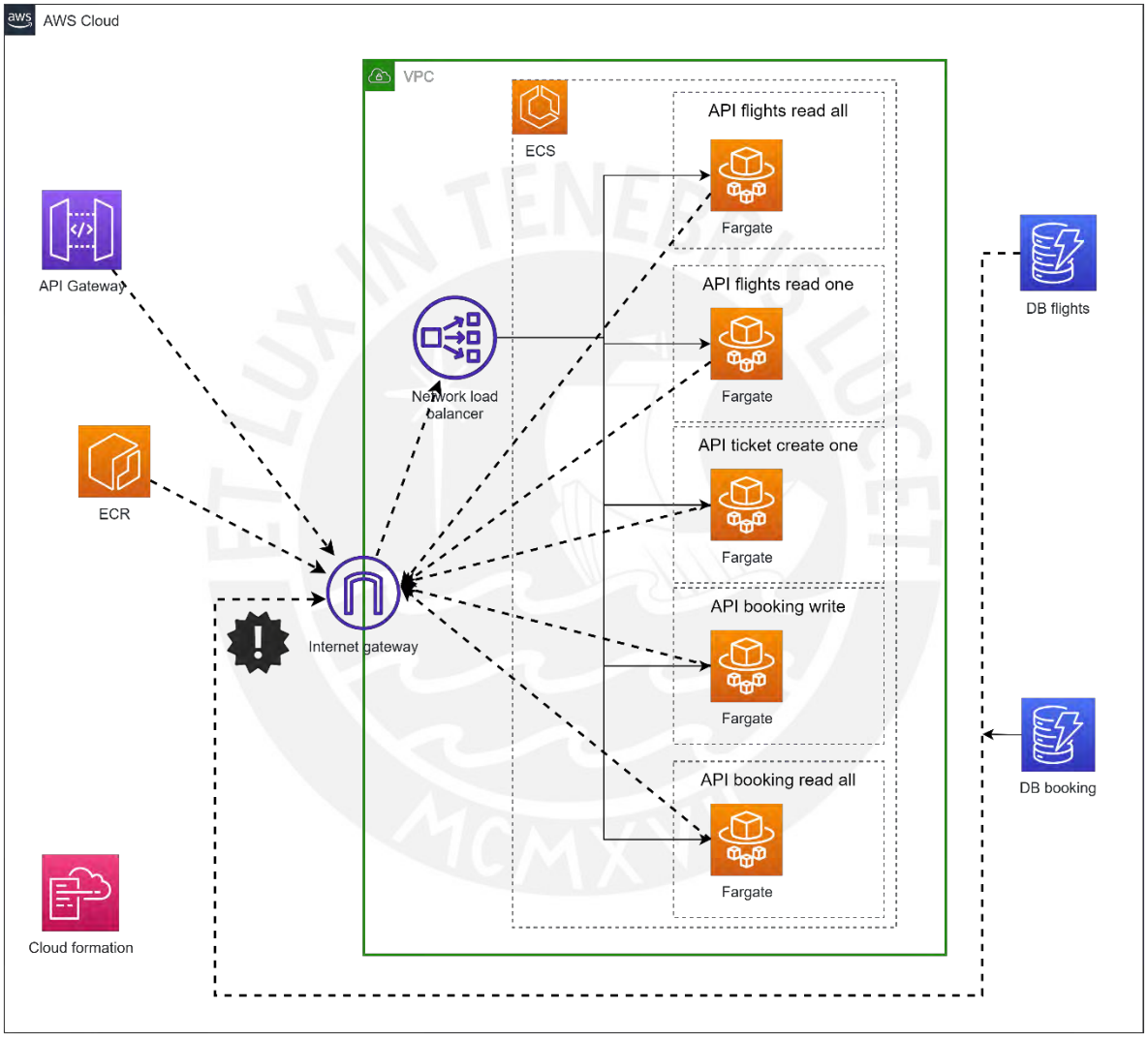

Figura 6.10: Arquitectura inicial de CaaS

6.2.4 Despliegue final en CaaS

La arquitectura final de CaaS se muestra en la figura 6.11, en la cual se observa que las 5APIs se desplegarán en servicios Fargate distintos sobre un mismo ECS, cada servicio de API tendrá su propio puerto de *listening* usando un mismo balanceador.

La configuración del VPC *endpoint* permite un ahorro de costos ya que no es necesario que la comunicación de los Fargate tenga que salir a internet a través del internet Gateway para alcanzar los otros recursos de AWS como el ECR y DynamoDB. Además, se configuró un PrivateLink entre el API-

Gateway y el balanceador de red para evitar la publicación del balanceador, el cual implica un costo relativamente alto.

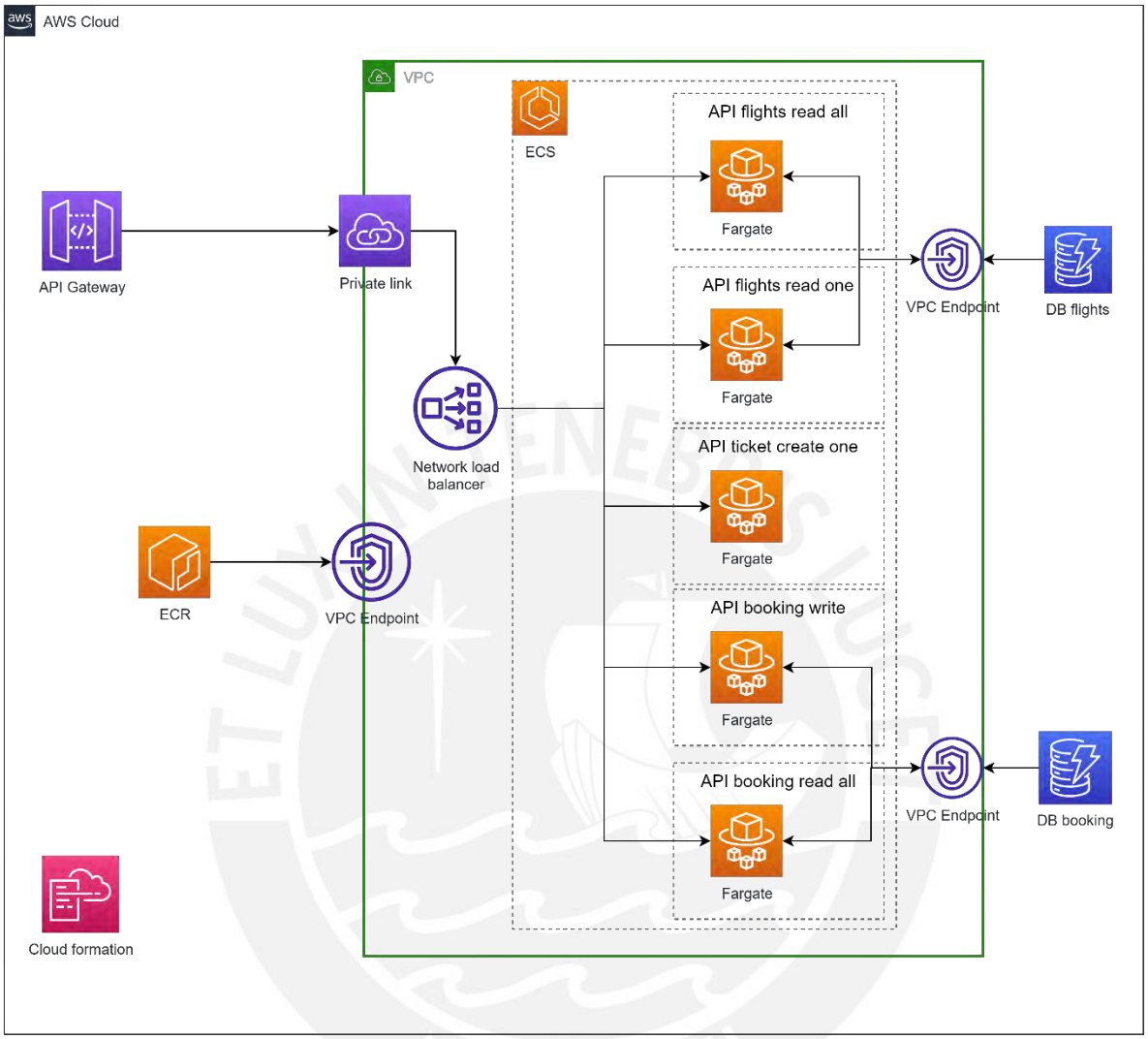

Figura 6.11: Diagrama de arquitectura de despliegue CaaS optimizado

6.3 Despliegue de sistema para la ejecución de pruebas de rendimiento

Se decidió desplegar la herramienta de ejecución de pruebas de carga en la nube de Google, denominada GCP, debido a que ofrece una capa gratuita de 3 meses para usar el servicio *compute engine,* el cual permite el uso de máquinas virtuales con S.O. Linux. Se descartaron AWS y Azure debido a que sus ofertas de capa gratuita no incluyen el uso de las máquinas virtuales.

La figura 6.12 muestra la implementación del sistema de ejecución de pruebas automáticas basada en *scripts* que sea ejecutará desde GCP hacia AWS.

6.3.1 Módulo GCP

Este módulo, mostrado como una zona celeste de la figura 6.12, contiene la máquina virtual Linux, al cual se accede por CLI y se configura un *script* (cron de Linux) que ejecuta los archivos JMX de JMeter cada hora por 10 minutos durante 3 días, estos *scripts* generan archivos CSV.

Desde esta máquina virtual se envía el tráfico hacia los dominios públicos de las APIs desplegadas en AWS. También se monitorea el uso de recursos (CPU y memoria) mediante un script de Python el cual genera un archivo de texto con valores separados por coma para ser procesados como CSV.

6.3.2 Módulo de AWS

Este módulo, mostrado como una zona naranja en la figura 6.12, contiene un área titulada "despliegue de APIs" que representa el despliegue de los 2 tipos de arquitecturas de despliegue FaaS y CaaS.

Para el monitoreo de consumo de recursos de AWS se usó el servicio de CloudWatch. Este servicio contiene *dashboards* para monitoreo y también genera archivos de logs, los cuales se pueden utilizar para resolver errores en el despliegue o para complementar el monitoreo de uso de recursos.

Luego de revisar varios archivos de *logs* extensos, se descubrió que se puede obtener información relevante a métricas con mayor rapidez mediante la generación de gráficos en la página web de métricas de la consola de AWS, donde cada gráfico permite la descarga de datos. Sin embargo, este servicio de AWS no permitía guardar y organizar las gráficas generadas, por lo cual se buscó un servicio complementario y se encontró un servicio de pago de AWS dentro de la misma web de CloudWatch dedicado a *dashboards*. Este último servicio se utilizó para organizar todos los *dashboards* y así finalmente aplicar los filtros de métricas necesarios y descargarlos como CSV.

Por otro lado, se accedía diariamente a los servicios de *cost* y *billing management* los cuales permiten verificar el costo real por hora y el consumo de *free tier* respectivamente. Ambos tipos de información se exportaron como CSV.

6.3.3 Módulo PC Local

Este módulo, mostrado como una zona gris en la figura 6.12, trata sobre el desarrollo del código fuente en TypeScript para la infraestructura usando el *framework* CDK de AWS, el cual se ha publicado y actualizado en https://github.com/gdmatosc/proyecto_AWS_CDK-Tesis_PUCP, en este repositorio también se agregará los archivos CSV de los resultados. Además, se incluye el desarrollo del código en JavaScript para el *frontend* y *backend* del app de *benchmark*, cuyo código fuente se ha publicado en https://github.com/gdmatosc/app_benchmark-Tesis_PUCP.

También, se desarrollaron *scripts* para el procesamiento de los CSVs descargados y la generación de gráficas y tablas estadísticas. Hay que considerar que el código de *scripts* del submódulo RStudio no es parte del alcance del *paper*, así como el código de *scripts* de la máquina virtual Linux.

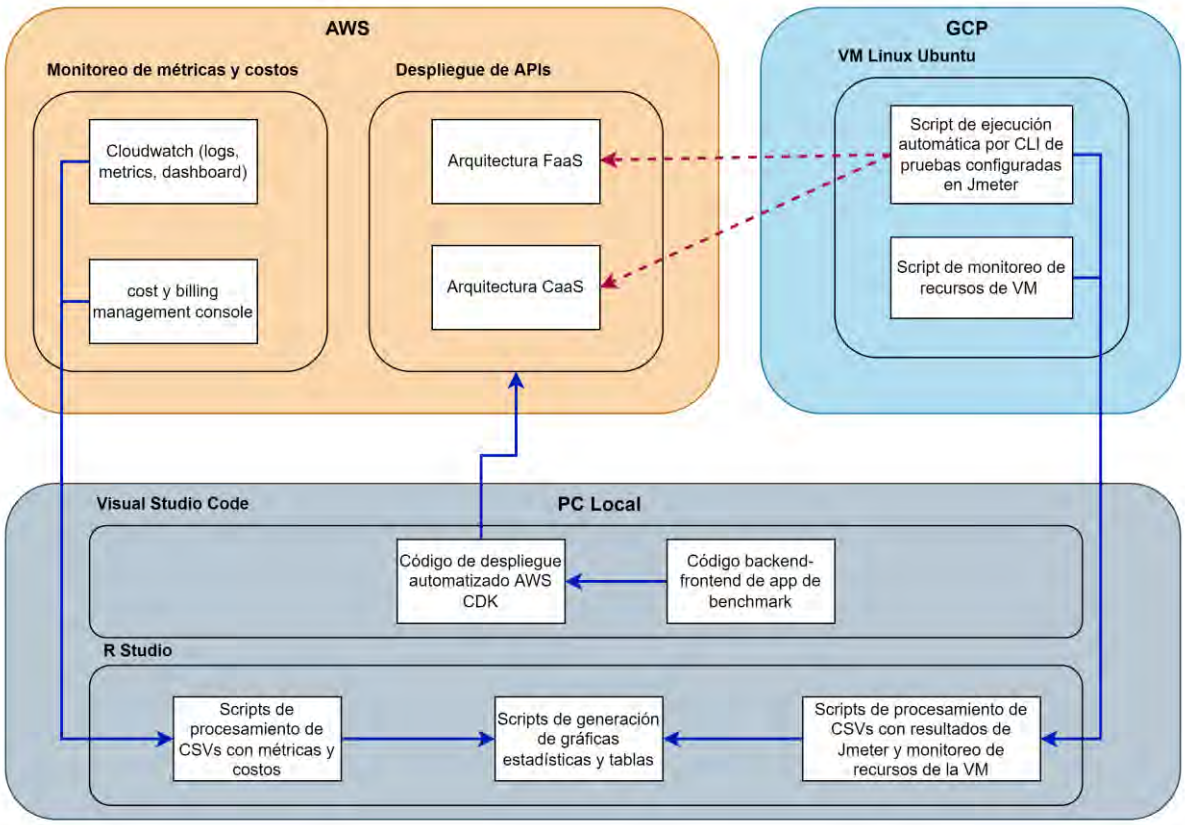

Figura 6.12: Diseño de sistema basado en script para la ejecución de pruebas automáticas

7 EVALUACIÓN DE RESULTADOS DEL EXPERIMENTO

En esta sección se presentan los resultados de los experimentos. El análisis de los datos de performance, generados por los servicios de monitoreo de uso de recursos de AWS. Así como la evaluación de los costos generados en diversos escenarios, basado en cálculos usando las fórmulas de AWS, las cuales se validaron previamente con los datos generados por el servicio de monitoreo de costos de la consola de AWS.

7.1 Ejecución del experimento

Se han realizados 2 tipos de prueba, las preliminares y las oficiales, a continuación, se describe cada uno:

- Las pruebas preliminares se realizaron con los siguientes valores de recursos: en FaaS, la memoria de las funciones Lambda es 512MB, y en CaaS, los Fargate tienen 0.5vCPU y 1GB de memoria. Estos valores corresponden a los días 10/10 para Local vs FaaS y 11/10 para FaaS vs CaaS.
- Para las pruebas oficiales se han considerado 3 grupos, cada uno en un día distinto, durante los días: viernes 18/11, sábado 19/11 y lunes 21/11. El grupo 1 consiste en las configuraciones de recursos FaaS 0.3vCPU-512MB y CaaS 0.5vCPU-1GB para las 5 APIs. El grupo 2 consiste en FaaS 0.15vCPU-256MB y CaaS 0.25vCPU-1GB para las 5 APIs. El grupo 3 consiste en FaaS 0.07vCPU-128MB y CaaS 0.25vCPU-512MB para las 4 APIs, y FaaS 1vCPU-1769MB y CaaS_1vCPU-2GB para el API *ticket*.

7.1.1 Desviaciones de la planificación

Se identificaron limitaciones de AWS durante la ejecución de las pruebas de carga, por lo cual se tuvo que restringir el alcance de las pruebas:

- Se había planificado hacer pruebas con 50 usuarios virtuales en simultáneo, pero AWS solo permite un máximo de 10. Para ampliar esta cantidad se debe realizar una solicitud formal a AWS.
- Se había planificado que la carga de la función de Fibonacci durara varios minutos, pero el API-Gateway solo permite un máximo de 30 segundos de tiempo de espera, por lo tanto, se programó la función para que dure menos segundos. Se determinó que el tiempo de referencia es de 10 segundos a nivel local ya que se verificó mediante pruebas que el tiempo de ejecución en una máquina local (AMD Ryzen 7, RAM 8GB) para el API *ticket* es menor en varios segundos que en los servicios en nube, tales como Lambda o Fargate.

7.2 Análisis de resultados

Las pruebas preliminares consistieron en 2 pruebas, la primera fue una de comparación Local vs FaaS para evidenciar la efectividad tanto de la generación de resultados en JMeter como de la comparación estadística, la segunda fue una de FaaS vs CaaS para validar que se pueda ejecutar sin inconvenientes las pruebas oficiales durante 3 días.

Respecto a las pruebas oficiales, hay que considerar que los resultados de todos los response HTTP se mostraron en forma resumida en tablas, sin embargo, en las gráficas solo se muestra lo ocurrido entre las 10am y 6pm ya que se considera que lo que está fuera de ese horario es parte de las validaciones pre y post prueba durante cada uno de los 3 días.

Por otro lado, las APIs *booking write* y *ticket create one* fueron desarrolladas para que acepten un máximo de 50 solicitudes HTTP exitosas que representan exactamente a los 50 usuarios virtuales cargados como CSV en las pruebas de JMeter. Esto se garantiza porque cuando la solicitud de un usuario, que ya existe, es procesado por la API *booking write,* inmediatamente es rechazada con un código 506. Mientras tanto, en JMeter se configuró la condición: si se recibe ese código de error en la llamada a la API *booking write,* entonces ya no se continua con las pruebas con la API *ticket create,* si no que se pasa directamente al API *booking read all.*

Respecto a las validaciones estadísticas, se utilizó la herramienta RStudio aplicando las pruebas de *test* de Levene (LVT) y Fligner-Killeen (FKT) para validar la homogeneidad de varianzas de 2 muestras, el cual es un supuesto necesario para usar la prueba no paramétricas de Wilcoxon-Mann-Whitney (WMW), también denominada Wilcoxon test (WCT), la cual permite validar la igualdad de medias de 2 muestras. En caso no se cumpla con este supuesto se aplicará las pruebas de Brunner-Munzel (BM) como se indica en el *paper* [57] con un intervalo de confianza de 95%. Las formas de aplicar ambas pruebas (WMW y BM) son muy similares ya que su hipótesis nula es la igualdad de medias entre 2 muestras.

Adicionalmente, para las pruebas WMW se considerará el tamaño de efecto, el cual es un valor que sirve para medir (entre 0 y 1) la magnitud de la diferencia de la métrica evaluada, lo cual permite asegurar la validez de la prueba. Por ejemplo, si el resultado de la prueba de WMW indica que se rechaza la hipótesis nula de igualdad de varianzas, es decir que hay una diferencia significativa en los valores de las medias, se espera que el tamaño de efecto; que puede ser bajo, moderado o alto; sea al menos moderado para considerar que la prueba es válida, y de esta forma interpretar que la diferencia entre las varianzas es relevante.

Además, hay que considerar que en caso se realice tanto las pruebas de WMW como BM para las mismas muestras, entonces se dará prioridad a WMW por ser muy usada en *papers* que aplican evaluaciones estadísticas no paramétricas.

Finalmente, se recomienda tener cuidado al observar los valores de *p-value* (pvalue wct) y *effect-size* (eff.size wct) de las pruebas de WMW que están presentes en las tablas de resultados de comparación ya que a diferencia de las tablas de validación, que solo presentan *p-values* para las pruebas LVT y FKT, se pueden confundir debido a que ambos valores (*p-value* y *effect-size*) son menores a 1 pero tienen interpretaciones diferentes. Para este caso, no dar importancia al valor de la columna de *p-value,* si no a la columna de magnitud de *effect size* que tiene como posibles resultados a *small, moderate, large.*

7.2.1 Prueba de rendimiento preliminar 1: Local vs FaaS

Las pruebas se realizaron en las siguientes condiciones:

- Ejecución de JMeter a nivel local por interfaz GUI.
- Las pruebas a nivel local se realizaron en una laptop personal con CPU de 8 núcleos 2.3GHz con 8GB de memoria.
- El tamaño de memoria configurado para los Lambdas de FaaS fue de 512MB.

7.2.1.1 Resultados de response code

En la tabla 7.1 se muestra el resumen de los *response code* por cada API, se observa que no hubo ningún error de JMeter (jmeter.error como código de respuesta) en todas las respuestas HTTP de las 5 APIs. Los errores 409 y 504 tratan sobre limitaciones en el uso de recursos de la infraestructura de despliegue y el 506 es un error personalizado de repetición de registro de usuario.

Las gráficas time-series del número de códigos de respuesta de estos 5 APIs se muestran en las figuras 7.1 y 7.2, se observa que la cantidad de *response code* para local y FaaS cambiaban irregularmente cada minuto, pero no superaron la cantidad de 10. Esta irregularidad se debe a que las pruebas se realizaron manualmente mediante el GUI de JMeter.

Tabla 7.1: Resumen de response code para la prueba preliminar local vs FaaS

Nombre del API	Fecha	Total Req	Res code	Res 506	Error
apiBusquedaDeVuelosDisponibles-FaaS	Lu:10-10	62	200	0	0
apiBusquedaDeVuelosDisponibles-local	Lu:10-10	80	200	0	0
apiSeleccionDeUnVuelo-FaaS	Lu:10-10	60	200	0	0
apiSeleccionDeUnVuelo-local	Lu:10-10	79	200	0	0
apiRegistroDeParticipante-FaaS	Lu:10-10	60	200, 506	10	0
apiRegistroDeParticipante-local	Lu:10-10	70	200, 409	0	20
apiBusquedaDeParticipantesSeleccionados-FaaS	Lu:10-10	59	200	0	0
apiBusquedaDeParticipantesSeleccionados-local	Lu:10-10	70	200	0	0
apiGeneracionDeNumeroDeTicket-FaaS	Lu:10-10	50	200, 504	0	32

apiGeneracionDeNumeroDeTicket-local	Lu:10-10	50	200	0	0

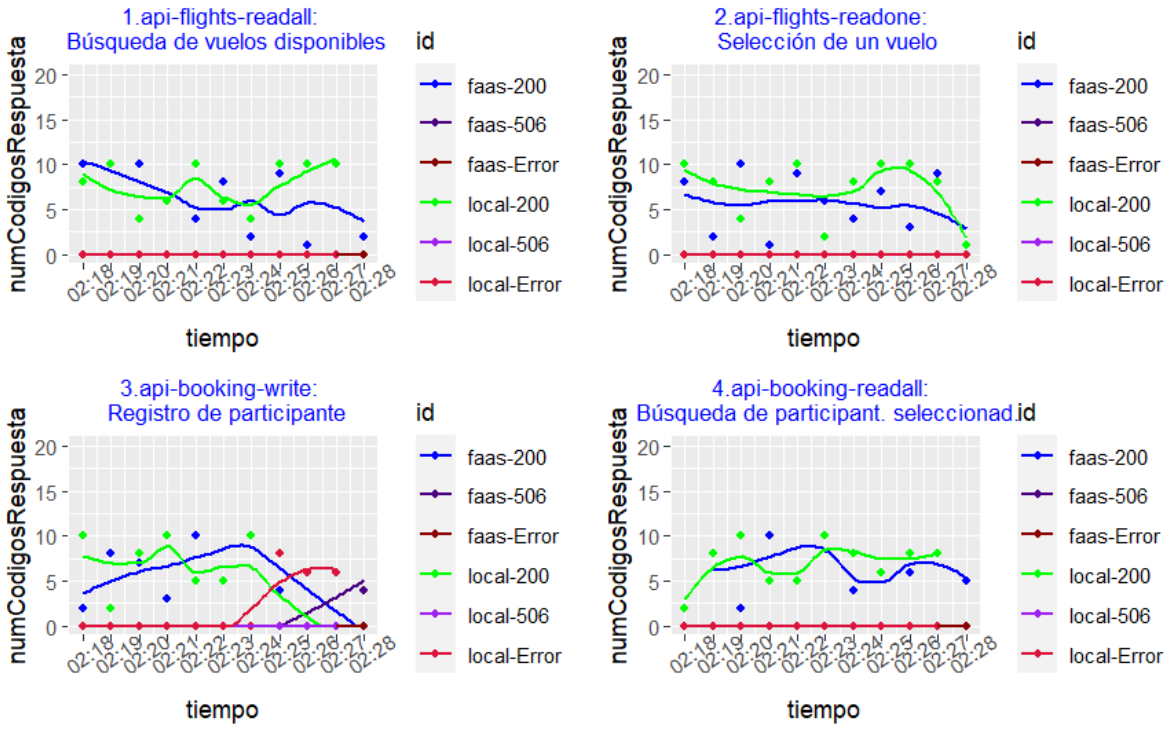

Figura 7.1: Gráfica time-series de response code de las 4 APIs para la prueba preliminar local vs FaaS

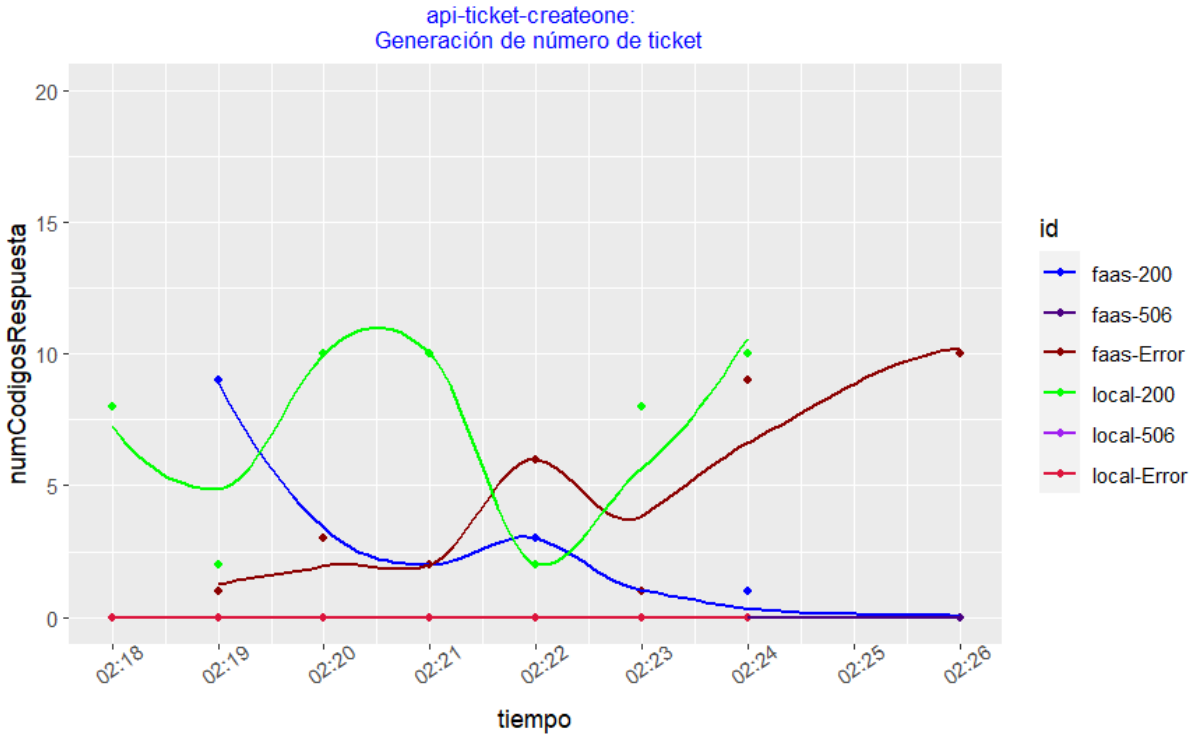

Figura 7.2: Gráfica time-series de response code del API ticket create one para la prueba preliminar local vs FaaS

7.2.1.2 Resultados de response time

En la tabla 7.2, se muestra el resumen estadístico de los *response time*.

En las figuras 7.3 y 7.4, se observa que las gráficas de *time-series* tienen una línea que representa los valores promedio de los *response code* y estos valores coinciden aproximadamente con el *mean* de la tabla 7.2. Además, se observa que para las 4 APIs de *flights* y *booking* el tiempo medio varía entre 9ms y 479ms, mientras que para el API *ticket create one* los tiempos medios varían entre 6 seg. y 24.7 seg.

Tabla 7.2: Resumen estadístico de response time para la prueba preliminar local vs FaaS

10-Oct	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
apiFlightsReadAll_FaaS	348	410	428.5	479.725806	535.75	797
apiFlightsReadAll_Local	4	7	7	9.55	10	124
apiFlightsReadOne_FaaS	159	206.75	214.5	219.083333	224	334
apiFlightsReadOne_Local	4	7	7	8.582278	10	23
apiBookingWrite_FaaS	152	219.5	235	238.316667	244.5	405
apiBookingWrite_Local	9	13	15	17	19	61
apiBookingReadAll_FaaS	213	401	421	422.186441	489	669
apiBookingReadAll_Local	4	9	12	12.314286	15	28
apiTicketCreateOne_FaaS	16218	17272	29165	24733.06	29168.75	29505
apiTicketCreateOne_Local	4853	5873	6149	6188.72	6500.5	7578

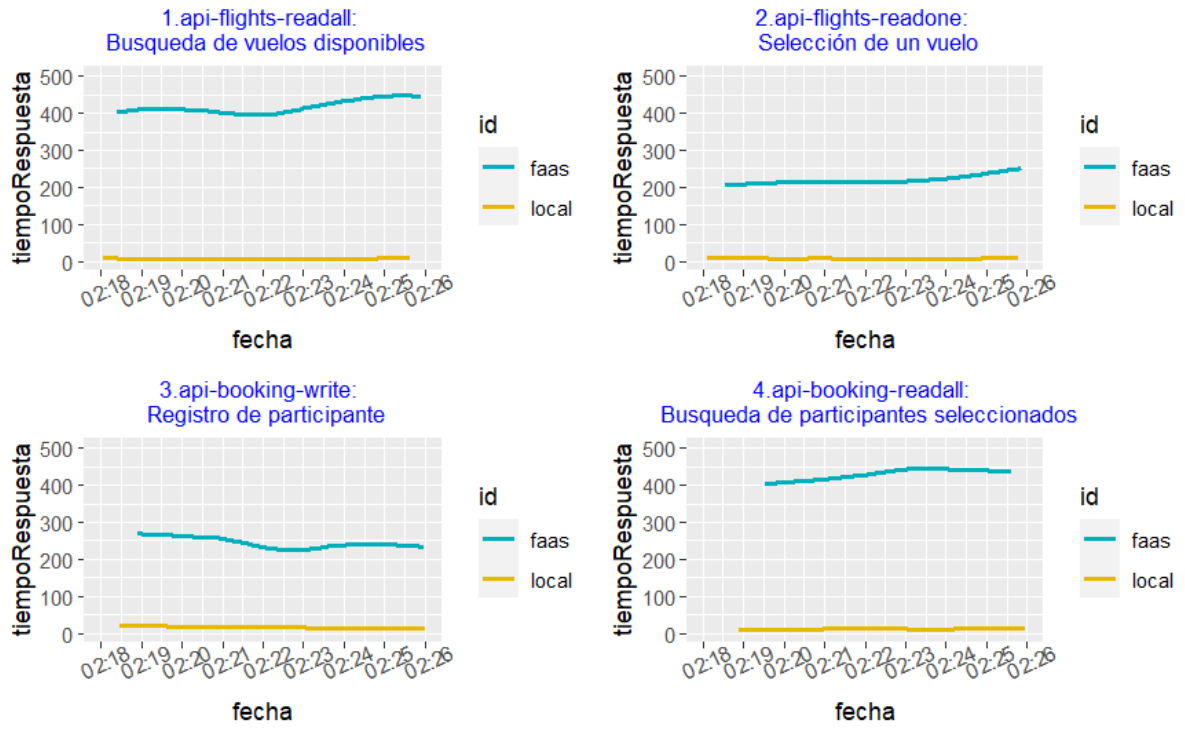

Figura 7.3: Gráfica time-series de response time de las 4 APIs para la prueba preliminar local vs FaaS

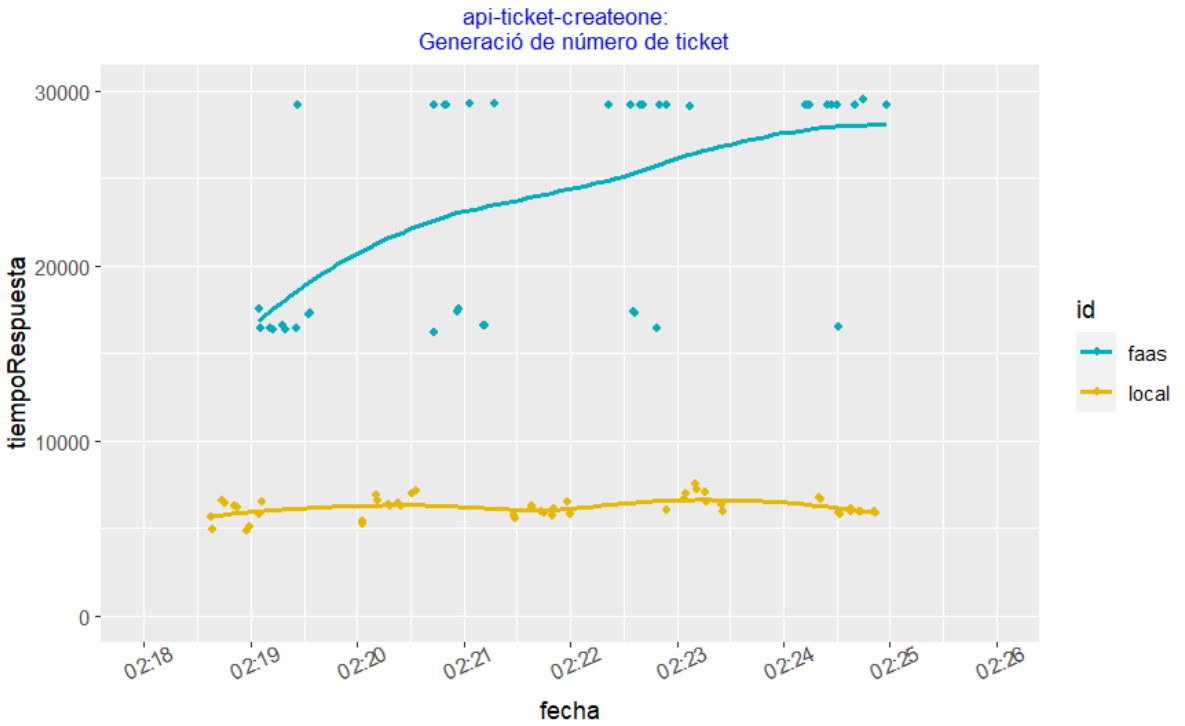

Figura 7.4: Gráfica time-series de response time del API ticket create one para la prueba preliminar local vs FaaS

7.2.1.3 Evaluación de diferencias estadísticamente significativa en los tiempos de respuesta

En las gráficas de *boxplot* de las figuras 7.18 y 7.19 se observa diferencias relevantes en las medias en las 4 APIs de *flights* y *booking*. Para el caso del API *ticket create one* también se observa una diferencia amplia. La confirmación de la diferencia de medias se determinó con los resultados de las tablas 7.3 y 7.4.

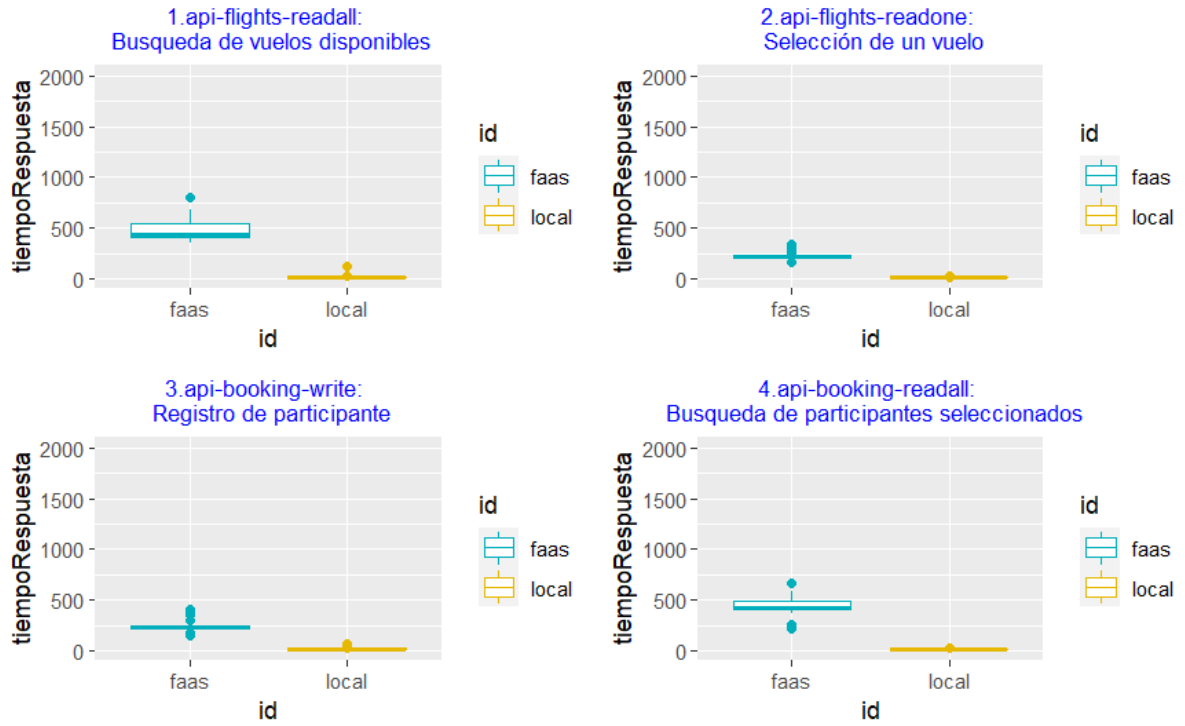

Figura 7.5: Gráfica boxplot de response time de las 4 APIs para la prueba preliminar local vs FaaS

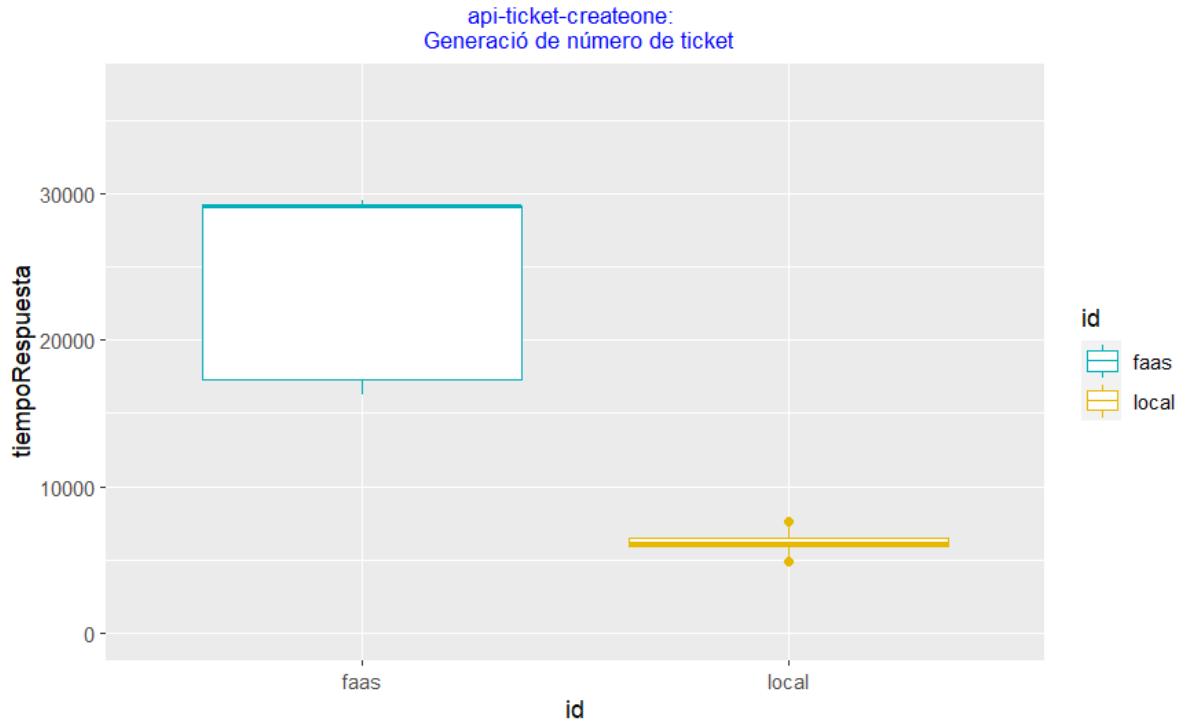

Figura 7.6: Gráfica boxplot de response time del API ticket create one para la prueba preliminar local vs FaaS

De acuerdo con los datos de la columna *pvalue-lvt* y *pvalue-fkt* de la tabla 7.3, se rechaza la hipótesis nula de igualdad de varianzas tanto del Levene *test* como del Fligner-Killeen *test* para las 4 APIs de vuelos y participantes ya que los valores de p-value son 0, por lo tanto, solo se realizarán las pruebas de Brunner-Munzel.

En el caso del API *ticket*, se acepta la hipótesis nula de igualdad de varianzas en las pruebas de LVT debido a que su *p-value* es mayor a 0.05, pero se rechaza en FKT debido a que su *p-value* es 0, por lo tanto, se usarán tanto las pruebas de WMW como la de BM.

Tabla 7.3: Validación de comparación estadística de response time para la prueba preliminar local vs FaaS

10-Oct	n1	n2	pvalue lvt	pvalue fkt	prueba valida
apiBusquedaDeVuelosDisponibles	62	80	0	0	BM
apiSeleccionDeUnVuelo	60	79	0	0	BM
apiRegistroDeParticipante	60	70	0	0	BM
apiBusquedaDeParticipantesSeleccionados	59	70	0	0	BM
apiGeneracionDeNumeroDeTicket	50	50	0.857	0	WMW (BM)

De acuerdo con los datos de la columna *pvalue-bmt* de la tabla 7.4, se rechaza la hipótesis nula de igualdad de medias para las 4 APIs de vuelos y participantes ya que los *p-value* son 0, por lo tanto, para esas 4 APIs, los resultados de comparación son "Diferentes".

En el caso del API *ticket create one,* se tienen 2 tipos de pruebas válidos, entonces se verifica primero las pruebas WCT. Se observa que el valor de la columna *pvalue-wct* es 0 y la magnitud del tamaño de efecto es "*large*", por lo tanto, se rechaza la hipótesis nula de igualdad de medias. Sucede lo mismo con la columna *pvalue-bmt*, en consecuencia, el resultado de la comparación es "Diferentes".

Tabla 7.4: Resultado de comparación estadística de response time para la prueba preliminar local vs FaaS

10-Oct	pvalue wct	eff.size wct	magn. wct	pvalue bmt	prueba valida	Resultado Comparación
apiBusquedaDeVuelosDisponibles	**0**	0.859	large	**0**	BM	**Diferentes**
apiSeleccionDeUnVuelo	**0**	0.858	large	**0**	BM	**Diferentes**
apiRegistroDeParticipante	**0**	0.861	large	**0**	BM	**Diferentes**
apiBusquedaDeParticipantesSeleccionados	**0**	0.86	large	**0**	BM	**Diferentes**
apiGeneracionDeNumeroDeTicket	**0**	0.862	large	**0**	WMW (BM)	**Diferentes** (Diferentes)

7.2.1.4 Discusión

Los resultados de la tabla 7.4 indican que los valores medios de *response time* son significativamente diferentes para las 5 APIs. Revisando nuevamente la tabla 7.2 con esta información se deduce que los tiempos de respuesta local para las 5 APIs son efectivamente menores que las pruebas en FaaS. Estas conclusiones se podían deducir directamente de las figuras 7.5 y 7.6, pero el objetivo era validar que las pruebas estadísticas podían verificarlo con precisión, lo cual ha sido comprobado.

7.2.2 Prueba de rendimiento preliminar 2: FaaS vs CaaS

Las pruebas se realizaron en las siguientes condiciones:

- Ejecución de JMeter a nivel local por interfaz GUI.
- El tamaño de memoria configurado para los Lambdas de FaaS fue de 512MB.
- La configuración de CPU de los Fargate de CaaS es 0.5vCPU y memoria 1GB.

7.2.2.1 Resultados de response code

En la tabla 7.5 se muestra el resumen de los *response code* por cada API, se observa que no hubo ningún error de JMeter (jmeter.error como código de respuesta) en todas las respuestas HTTP de las 5 APIs. El error 504 trata sobre una limitación en el uso de recursos de la infraestructura de despliegue y el 506 es un error personalizado de repetición de registro de usuario.

Las gráficas *time-series* del número de códigos de respuesta de las 5 APIs se muestran en las figuras 7.7 y 7.8, se observa que la cantidad de *response code* para FaaS y CaaS cambiaban irregularmente cada minuto, pero no superaron la cantidad de 10. Esta irregularidad se debe a que las pruebas se realizaron manualmente mediante el GUI de JMeter.

Tabla 7.5: Resumen de response code para la prueba preliminar FaaS vs CaaS

Nombre del API	Fecha	Total Req	Res code	Res 506	Error
apiBusquedaDeVuelosDisponibles-FaaS	Ma:11-10	76	200	0	0
apiBusquedaDeVuelosDisponibles-CaaS	Ma:11-10	72	200	0	0
apiSeleccionDeUnVuelo-FaaS	Ma:11-10	73	200	0	0
apiSeleccionDeUnVuelo-CaaS	Ma:11-10	70	200	0	0
apiRegistroDeParticipante-FaaS	Ma:11-10	70	506, 200	26	0
apiRegistroDeParticipante-CaaS	Ma:11-10	68	506, 200	24	0
apiBusquedaDeParticipantesSeleccionados-FaaS	Ma:11-10	70	200	0	0
apiBusquedaDeParticipantesSeleccionados-CaaS	Ma:11-10	66	200	0	0
apiGeneracionDeNumeroDeTicket-FaaS	Ma:11-10	44	200	0	0
apiGeneracionDeNumeroDeTicket-CaaS	Ma:11-10	44	200, 504	0	2

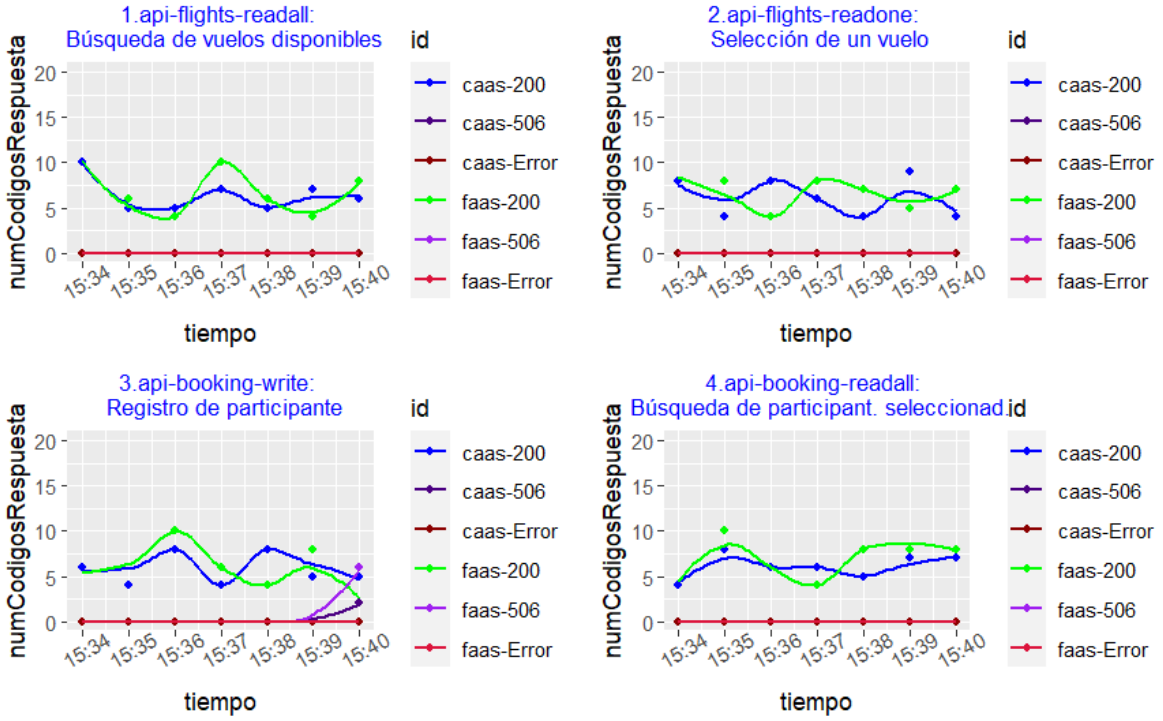

Figura 7.7: Gráfica time-series de response code de las 4 APIs para la prueba preliminar FaaS vs CaaS

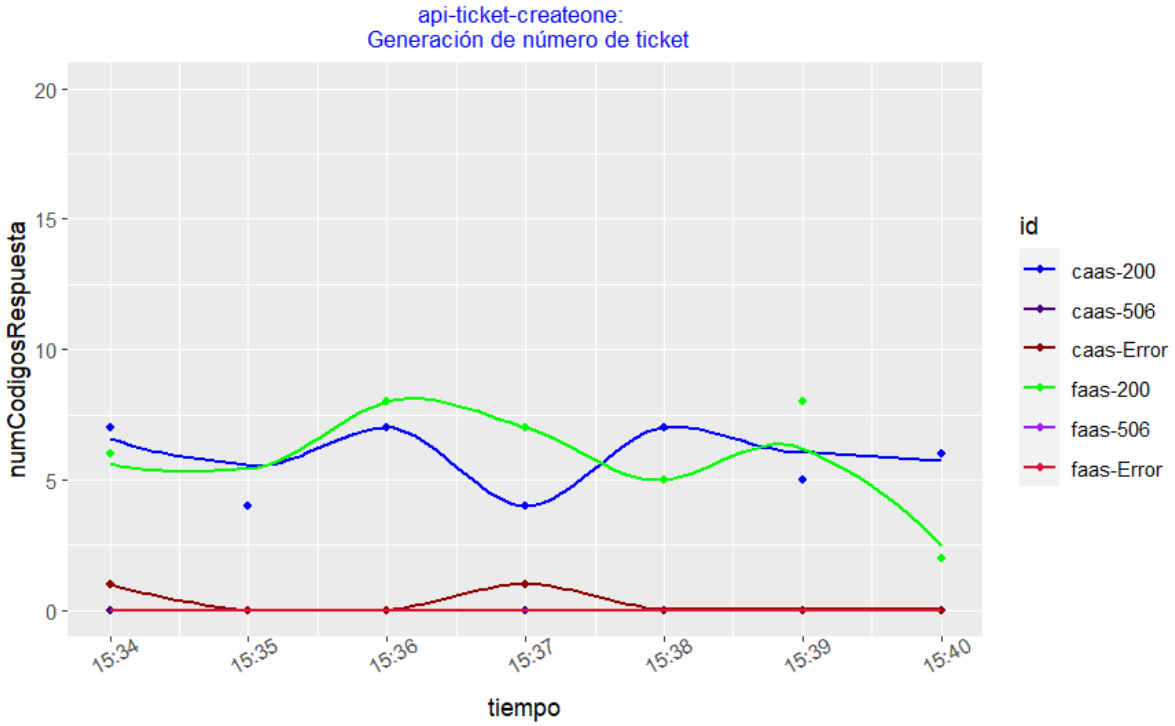

Figura 7.8: Gráfica time-series de response code del API ticket create one para la prueba preliminar FaaS vs CaaS

7.2.2.2 Resultados de response time

En la tabla 7.6, se muestra el resumen estadístico de los *response time*.

En las tablas 7.7 y 7.8, se observa los resultados de la comparación de evaluación de significancia estadística por cada API.

Las figuras 7.9, 7.10, 7.11 y 7.12 muestran las gráficas de *time-series* y *boxplot* de los *response time* de las 5 APIs.

Tabla 7.6: Resumen estadístico de response time para la prueba preliminar FaaS vs CaaS

11-Oct	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
apiFlightsReadAll_FaaS	337	400.75	418	572.7105	488.75	2266
apiFlightsReadAll_Caas	323	333.75	352	495.4583	427	1706
apiFlightsReadOne_FaaS	157	206	219	261.5479	227	787
apiFlightsReadOne_Caas	138	148	153.5	212.7286	164.75	880
apiBookingWrite_FaaS	163	215.25	231.5	360.6143	254	2396
apiBookingWrite_Caas	140	160.25	171	256.0441	187	846
apiBookingReadAll_FaaS	167	220	396	426.8571	490.75	1189
apiBookingReadAll_Caas	143	319.25	337	382.697	392	1097
apiTicketCreateOne_FaaS	6438	11702.75	11754	10984.7955	11932.25	12824
apiTicketCreateOne_Caas	6764	14114	14383.5	16677.2273	19416.25	29120

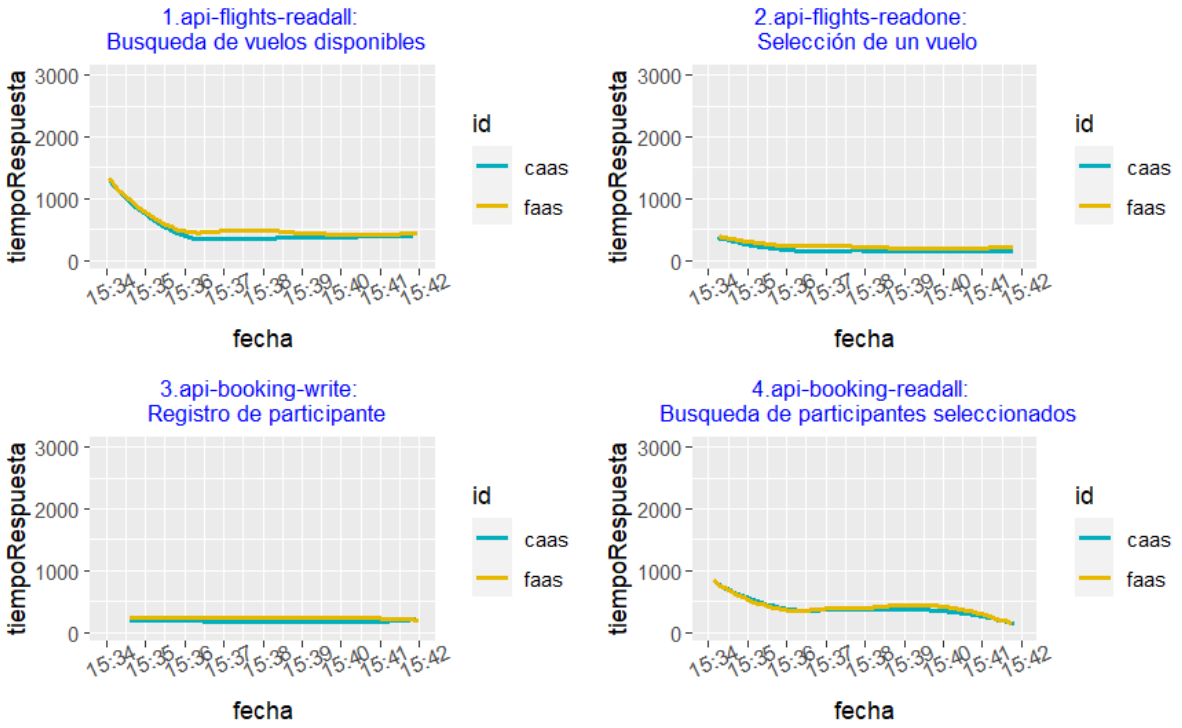

Figura 7.9: Gráfica time-series de response time de las 4 APIs para la prueba preliminar FaaS vs CaaS

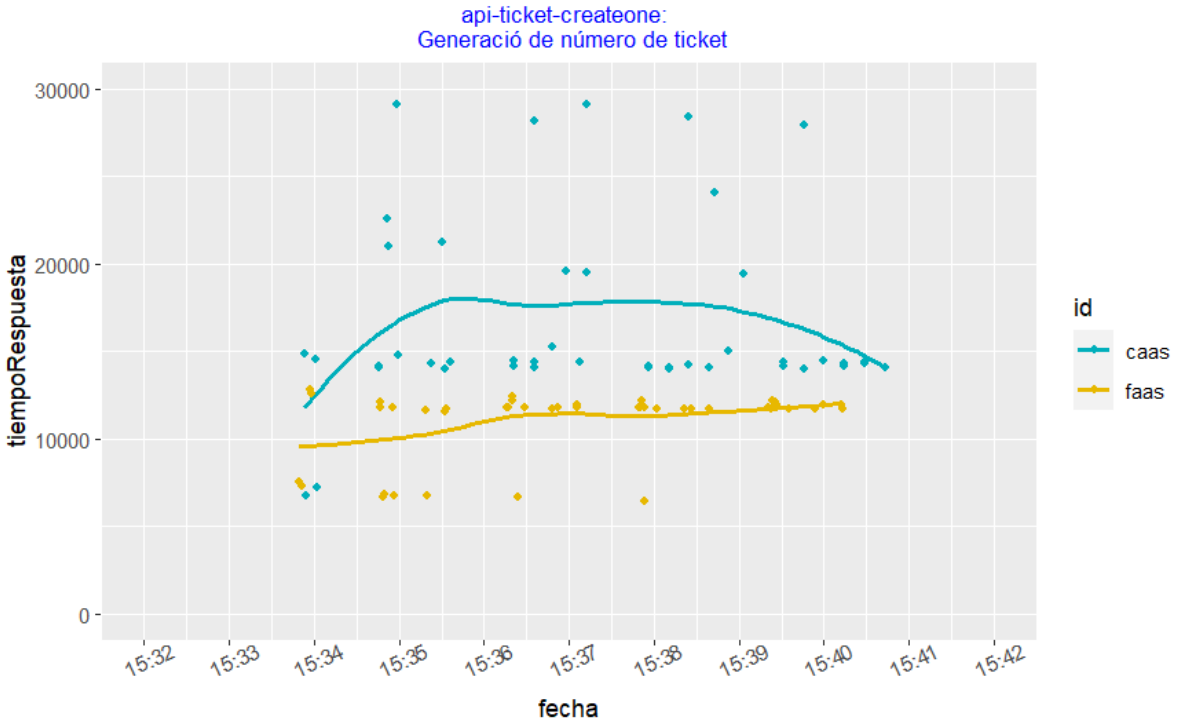

Figura 7.10: Gráfica time-series de response time del API ticket create one para la prueba preliminar FaaS vs CaaS

7.2.2.3 Evaluación de diferencias estadísticamente significativas en los tiempos de respuesta

En las gráficas de *boxplot* de las figuras 7.11 y 7.12 se observa que es difícil apreciar la diferencia de medias en las 4 APIs de *flights* y *booking*. En cambio, para el caso del API *ticket create one* sí se observa una diferencia relevante de medias. La confirmación de la diferencia de medias se determinará con los resultados de las tablas 7.3 y 7.4.

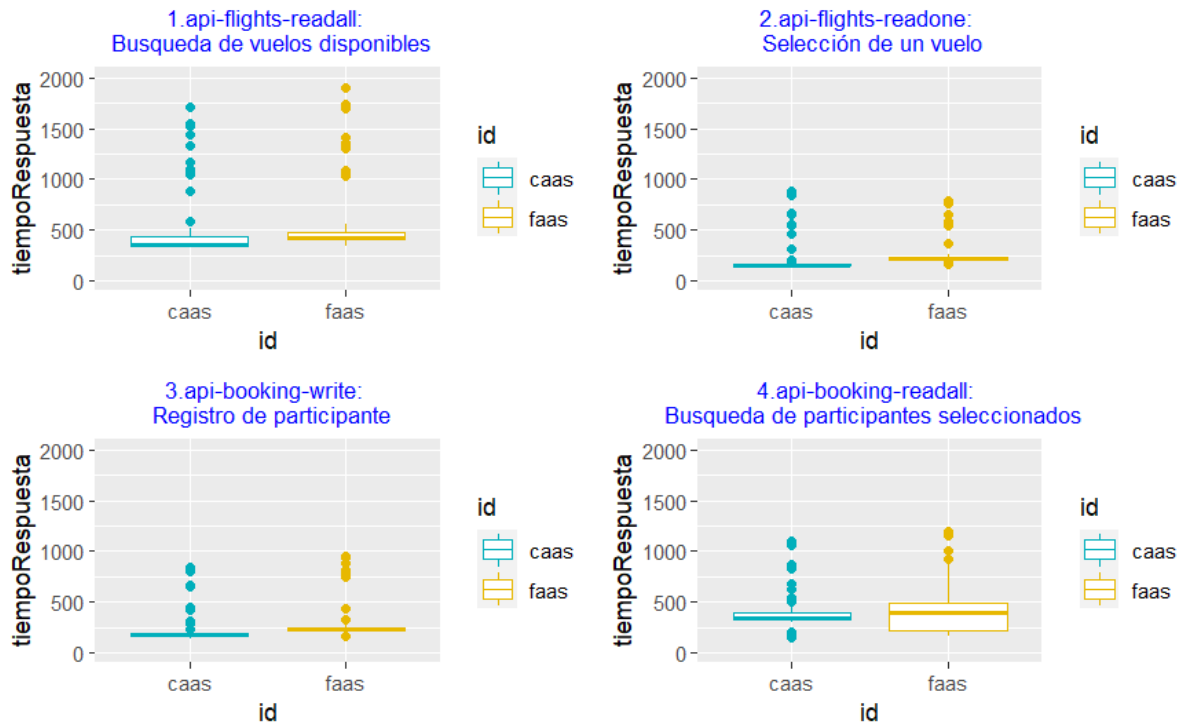

Figura 7.11: Gráfica boxplot de response time de las 4 APIs para la prueba preliminar FaaS vs CaaS

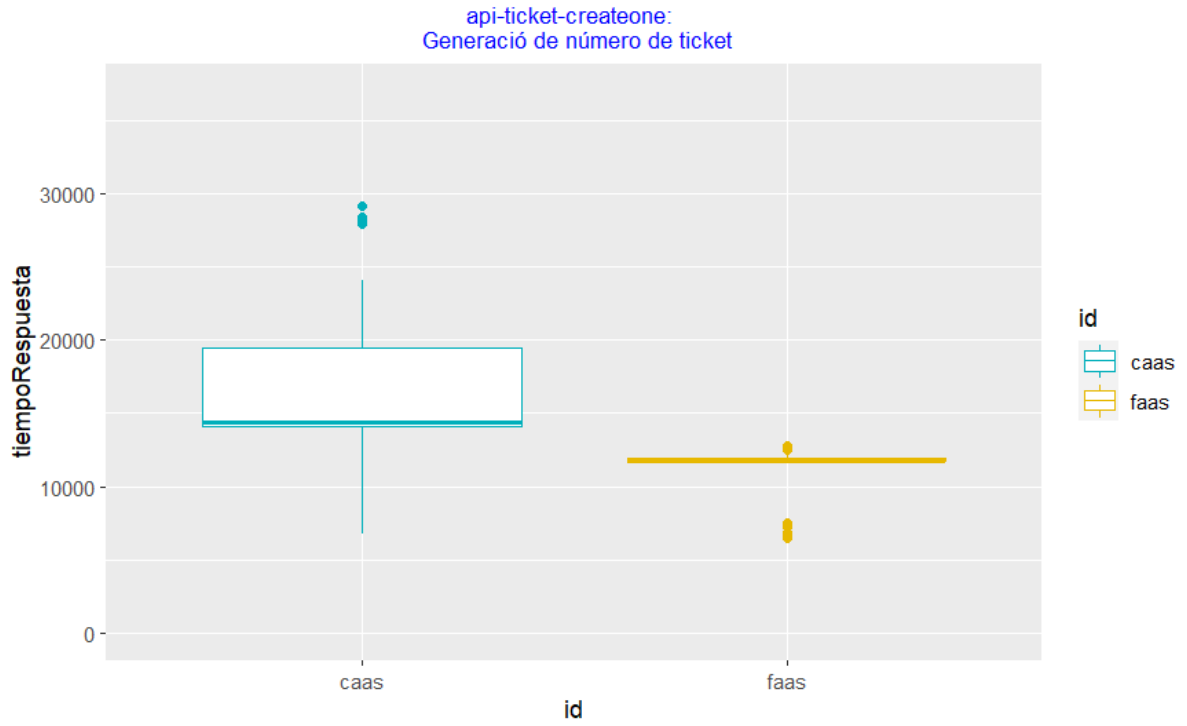

Figura 7.12: Gráfica boxplot de response time del API ticket create one para la prueba preliminar FaaS vs CaaS

De acuerdo con los datos de la columna *pvalue-lvt* y *pvalue-fkt* de la tabla 7.7, se acepta la hipótesis nula de igualdad de varianzas tanto del Levene *test* como del Fligner-Killeen *test* para las 4 APIs de vuelos y participantes, ya que los valores de *p-value* son mayores a 0.05, por lo tanto, solo se realizarán las pruebas de Wilcoxon-Mann-Whitney.

En el caso del API *ticket*, se rechaza la hipótesis nula de igualdad de varianzas tanto de las pruebas LVT como FKT, ya que los valores de *p-value* son menores a 0.05, por lo tanto, solo se usarán las pruebas de BM.

Tabla 7.7: Validación de comparación estadística de response time para la prueba preliminar FaaS vs CaaS

11-Oct	n1	n2	pvalue lvt	pvalue fkt	prueba valida
apiBusquedaDeVuelosDisponibles	72	76	0.693	0.752	WMW
apiSeleccionDeUnVuelo	70	73	0.144	0.844	WMW
apiRegistroDeParticipante	68	70	0.112	0.296	WMW
apiBusquedaDeParticipantesSeleccionados	66	70	0.089	0.212	WMW
apiGeneracionDeNumeroDeTicket	44	44	0.002	0.007	BM

De acuerdo con los valores de la columna *pvalue-wct* de la tabla 7.8 para las 3 APIs apiBusquedaDeVuelosDisponibles, apiSeleccionDeUnVuelo, apiRegistroDeParticipante, se se rechaza la hipótesis nula de igualdad de medias debido a que los *p-value* son 0 y la magnitud del tamaño de efecto son “moderate” o “large”, por lo tanto, los resultados de comparación son “Diferentes”.

Para el caso de la API api5BusquedaDeParticipantesSeleccionados, de acuerdo con el valor de la columna *pvalue-wct*, se debería rechaza la hipótesis nula de igualdad de medias debido a que el *p-value* es menor a 0.05, sin embargo, como la magnitud del tamaño de efecto es “small”, se considera que la prueba es inválida, en consecuencia, se considera que no existe la suficiente evidencia para afirmar que la diferencia de medias no se debe al azar.

Para el caso de la API api4GeneracionDeNumeroDeTicket de acuerdo con el valor de la columna pvalue-bmt, se rechaza la hipótesis nula de igualdad de medias debido a que el *p-value* es 0, por lo tanto, el resultado de la comparación es “Diferentes”.

Tabla 7.8: Resultado de comparación estadística de response time para la prueba preliminar FaaS vs CaaS

11-Oct	pvalue wct	eff.size wct	magn. wct	pvalue bmt	prueba valida	Resultado Comparación
apiBusquedaDeVuelosDisponibles	**0**	0.411	moder.	**0**	WMW	**Diferentes**
apiSeleccionDeUnVuelo	**0**	0.605	large	**0**	WMW	**Diferentes**
apiRegistroDeParticipante	**0**	0.516	large	**0**	WMW	**Diferentes**
apiBusquedaDeParticipantesSeleccionados	**0.04**	0.176	small	**0.043**	WMW	Diferentes (Inválido)
apiGeneracionDeNumeroDeTicket	**0**	0.793	large	**0**	BM	**Diferentes**

7.2.2.4 Discusión

Los resultados de la tabla 7.8 indican que los valores medios de response time son significativamente diferentes para las 5 APIs, a excepción del API búsqueda de participantes seleccionados, el cual no tiene un resultado de comparación válido. Revisando nuevamente la tabla 7.6 con esta información

se deduce que los tiempos de respuesta de CaaS son efectivamente menores que FaaS para las 3 APIs búsqueda de vuelos disponibles, selección de un vuelo y registro de participante. Igualmente, en el caso del API *ticket* se deduce que el tiempo de respuesta medio de FaaS es efectivamente menor que el de CaaS.

Para el caso del API búsqueda de participantes seleccionados se debería realizar una mayor cantidad de pruebas hasta observar diferencias estadísticamente significativas.

7.2.3 Prueba de rendimiento oficial 1: FaaS vs CaaS con la configuración de recursos 1

Las pruebas se realizaron en las siguientes condiciones:

- Ejecución de JMeter por CLI desde Google *Cloud.*
- El tamaño de memoria configurado para los Lambdas de FaaS fue de 512MB.
- La configuración de CPU de los Fargate de CaaS es 0.5vCPU y memoria 1GB.

7.2.3.1 Monitoreo de consumo de recursos de la máquina virtual

En la figura 6.13 se observa que no hay incidentes en el uso de recursos de la VM de Google *Cloud*.

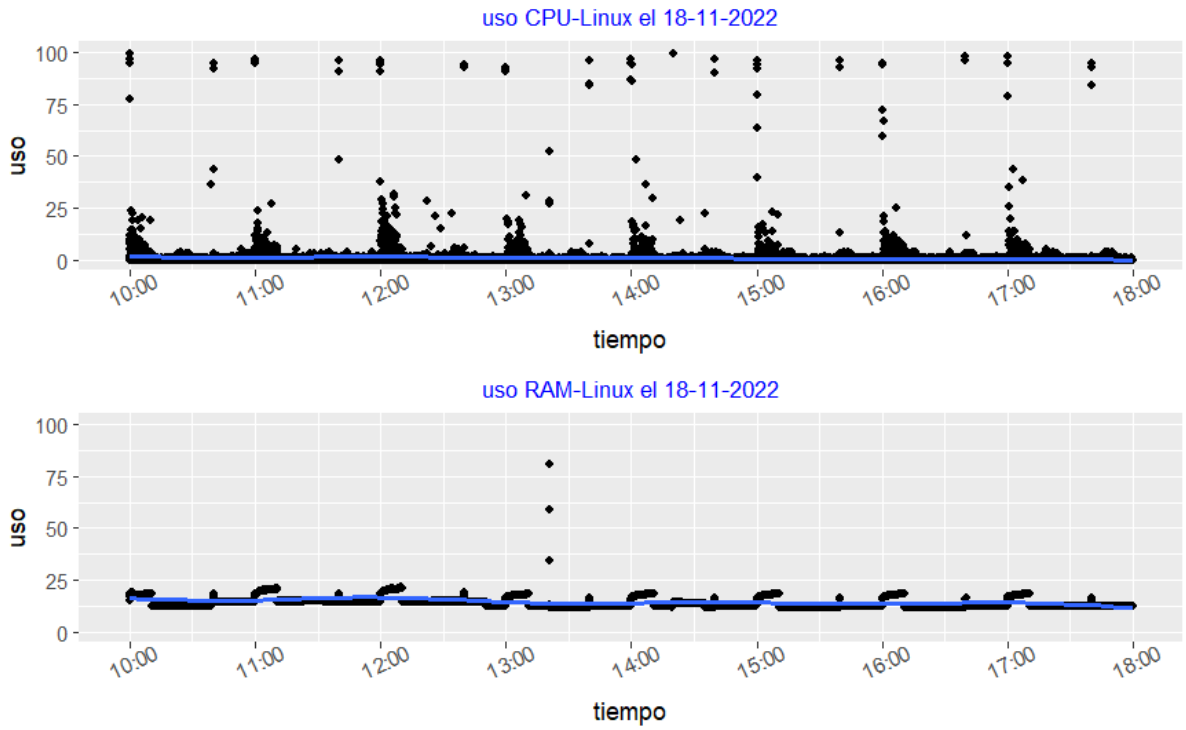

Figura 7.13: Consumo de recursos para la prueba oficial 1 de FaaS vs CaaS

7.2.3.2 Resultados de response code

En la tabla 7.9, se muestra el resumen de los *response code* por cada API, se observa que solo hubo 1 error de JMeter (jmeter.error como código de respuesta) en 980 respuestas HTTP para el API *flights read one*. Los errores 500 y 504 tratan sobre limitaciones en el uso de recursos de la infraestructura de despliegue y el 506 es un error personalizado de repetición de registro de usuario.

Las gráficas *time-series* del número de códigos de respuesta de las 5 APIs se muestran en las figuras 7.14 y 7.15, se observa que la cantidad de *response code* para FaaS y CaaS fueron similares y alrededor

de 70 por hora a excepción de las APIs *booking write* y *ticket create one,* ya que ambas están desarrolladas para que solo acepten 50 solicitudes HTTP exitosas según lo indicado en la sección de modificaciones al software original.

Tabla 7.9: Resumen de response code

Nombre del API	Fecha	Total Req	Res code	res 506	error
apiFlightsReadAll-FaaS	Vi:18-11	1002	200, 500	0	1
apiFlightsReadAll-CaaS	Vi:18-11	1036	200	0	0
apiFlightsReadOne-FaaS	Vi:18-11	980	200, jmeter.error	0	1
apiFlightsReadOne-CaaS	Vi:18-11	1000	200	0	0
apiBookingWrite-FaaS	Vi:18-11	980	200, 506	280	0
apiBookingWrite-CaaS	Vi:18-11	980	200, 506	280	0
apiBookingReadAll-FaaS	Vi:18-11	980	200	0	0
apiBookingReadAll-CaaS	Vi:18-11	979	200	0	0
apiTicketCreateOne-FaaS	Vi:18-11	700	200	0	0
apiTicketCreateOne-CaaS	Vi:18-11	700	200, 504	0	1

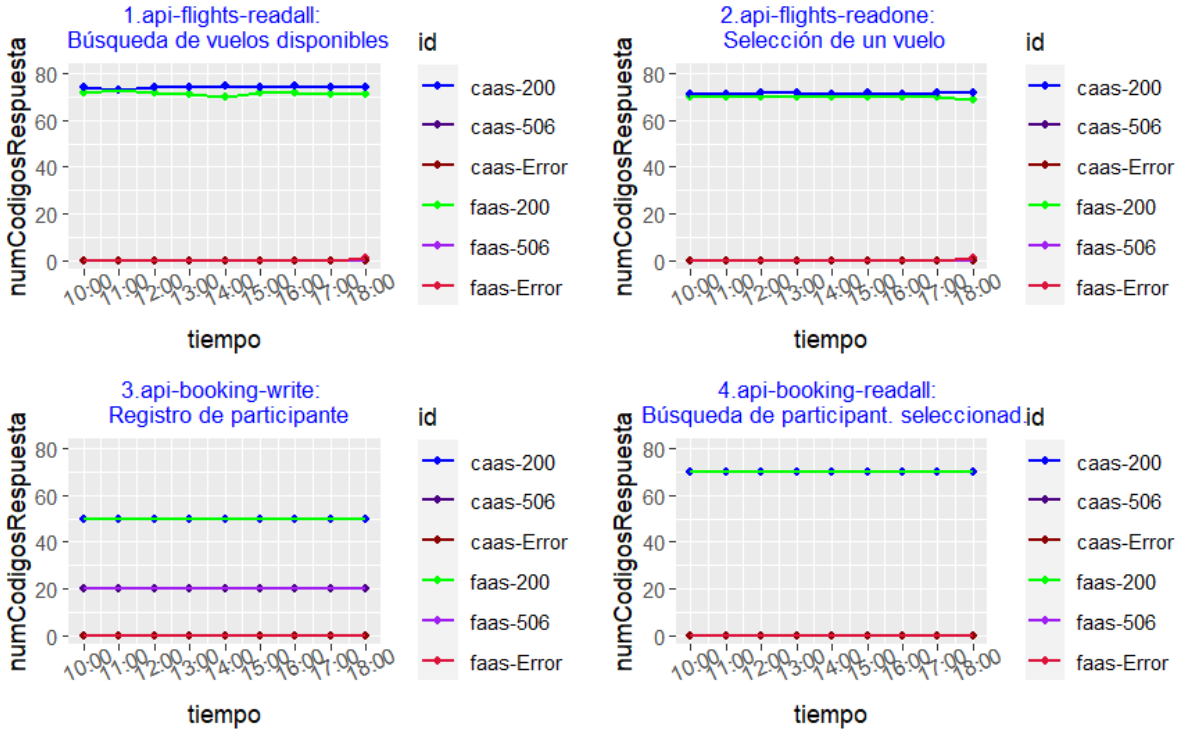

Figura 7.14: Gráfica time-series de response code de las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

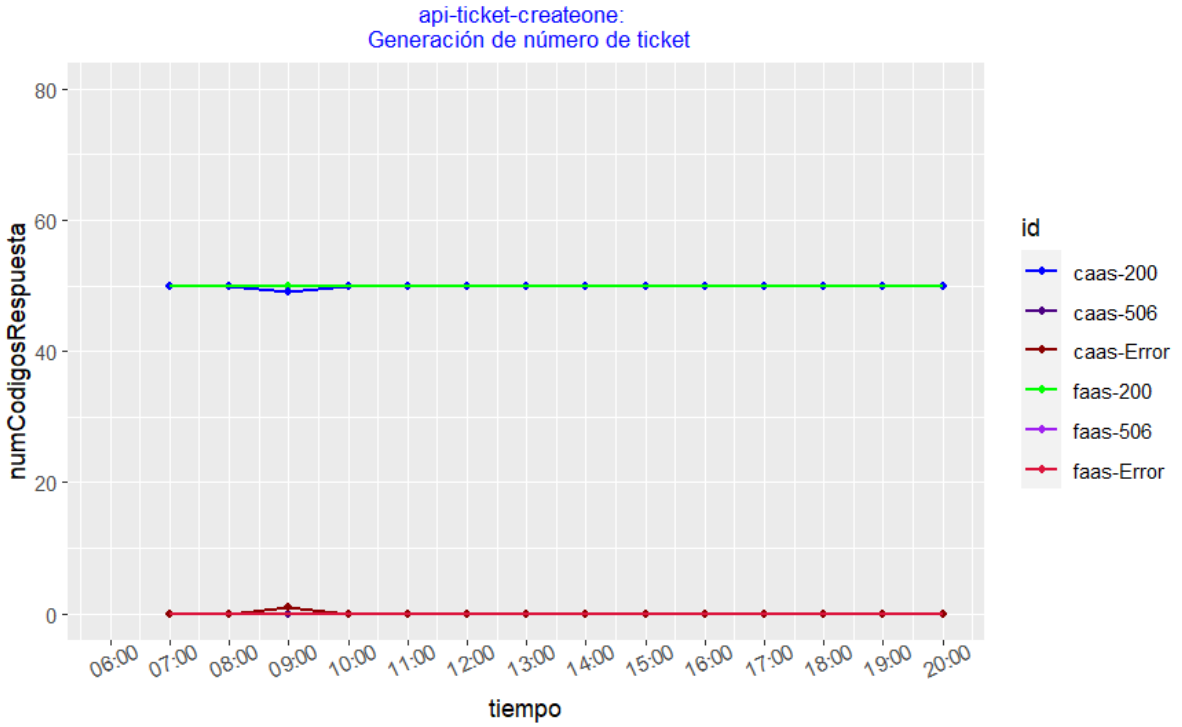

Figura 7.15: Gráfica time-series de response code del API ticket create one para la prueba oficial 1 de FaaS vs CaaS

7.2.3.3 Resultados de response time

En la tabla 7.10, se muestra el resumen estadístico de los *response time*.

En las figuras 7.16, 7.17, se observa que las gráficas de *time-series* contienen una línea que representa los valores promedio de los *response code* y estos valores coinciden aproximadamente con el *mean* de *response code* de la tabla 7.10. Además, se observa que para las 4 APIs de *flights* y *booking* el tiempo medio varía entre 189ms y 241ms, mientras que para el API *ticket create one* los tiempos medios están entre 9.9 seg. y 10.4 seg.

Tabla 7.10: Resumen estadístico de response time para la prueba oficial 1 FaaS vs CaaS

18-Nov	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
apiFlightsReadAll_FaaS	129	147	161	241.3696	273	2394
apiFlightsReadAll_CaaS	115	146	179	217.0121	272	826
apiFlightsReadOne_FaaS	0	125	135	207.8857	253	1633
apiFlightsReadOne_CaaS	103	120	133	181.9566	249	449
apiBookingWrite_FaaS	115	143	162	238.9929	276	3268
apiBookingWrite_CaaS	111	147	170	205.8729	275	665
apiBookingReadAll_FaaS	114	139	157	224.9329	269	1696
apiBookingReadAll_CaaS	100	124	159.5	189.89	253	417
apiTicketCreateOne_FaaS	5668	10946.25	11378.5	10498.756	11523	12046
apiTicketCreateOne_CaaS	6985	7156.75	7310	9909.568	13944.25	28436

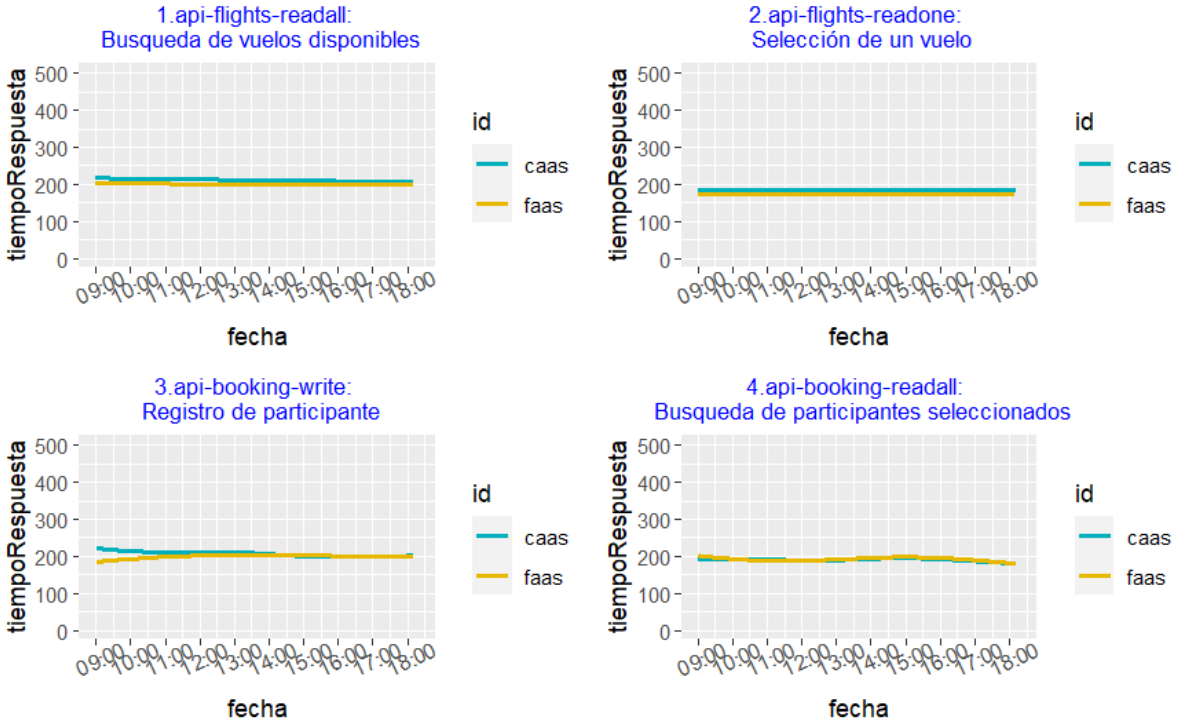

Figura 7.16: Gráfica time-series de response time de las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

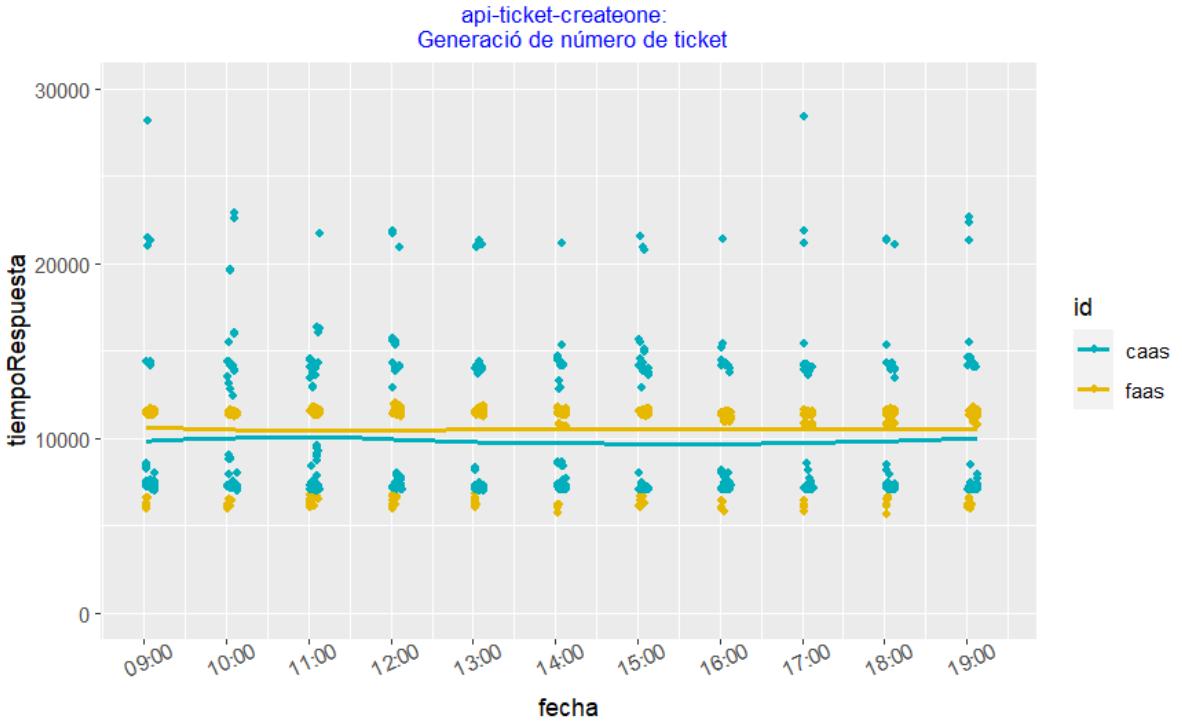

Figura 7.17: Gráfica time-series de response time del API ticket create one para la prueba oficial 1 de FaaS vs CaaS

7.2.3.4 Evaluación de diferencias estadísticamente significativas en los response time

En las gráficas de *boxplot* de las figuras 7.18 y 7.19 se observa que es difícil apreciar la diferencia de medias en las 4 APIs de *flights* y *booking*, mientras que en el caso del API *ticket create one* se observa una diferencia relevante de medias. El resultado final de la diferencia o igualdad de medias se determinará con los resultados de las tablas 7.11 y 7.12.

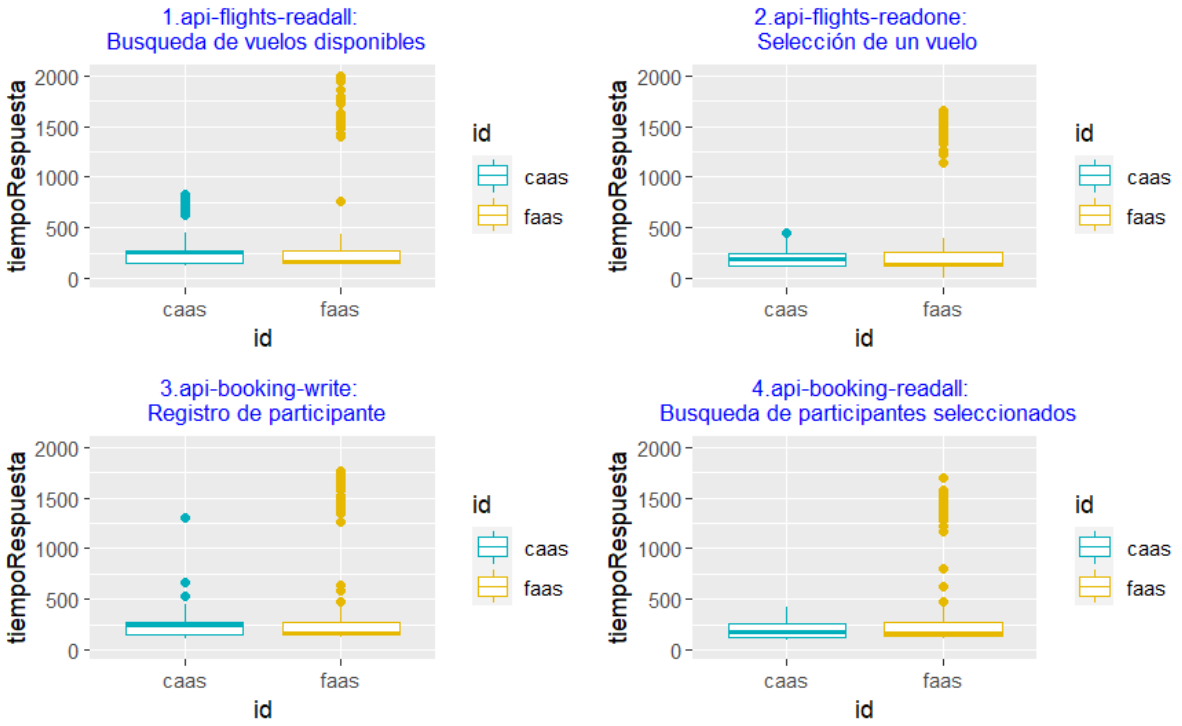

Figura 7.18: Gráfica boxplot de response time de las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

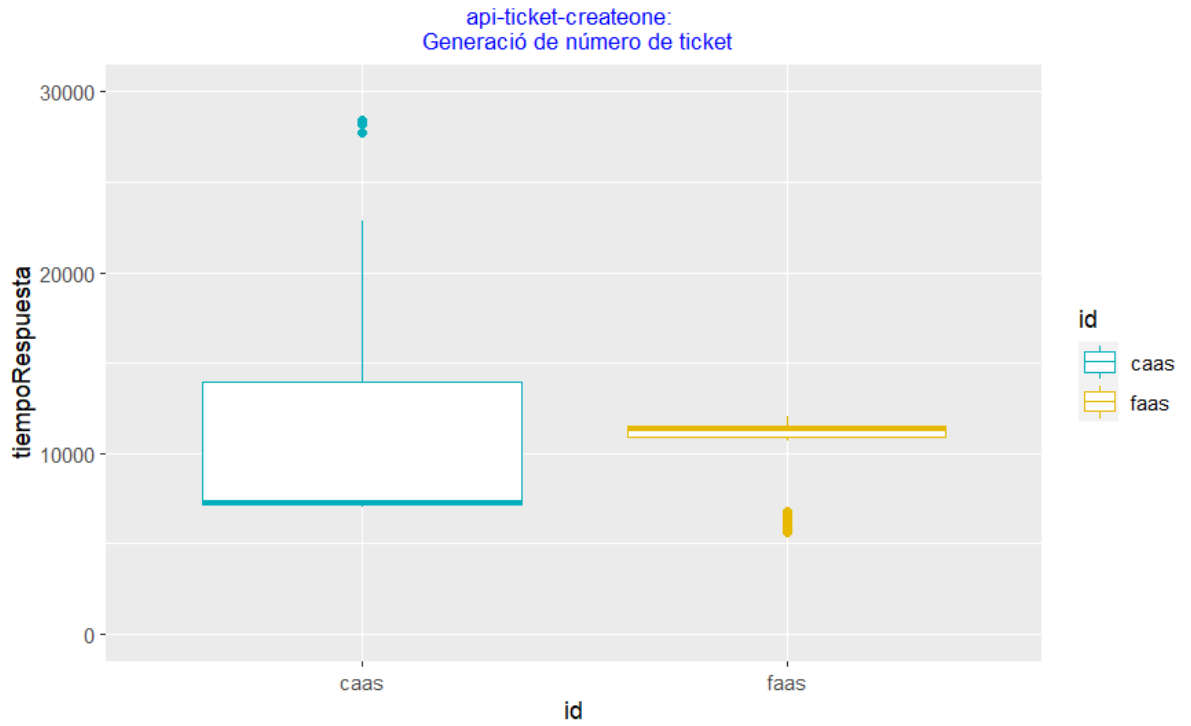

Figura 7.19: Gráfica boxplot de response time del API ticket create one para la prueba oficial 1 de FaaS vs CaaS

De acuerdo con los datos de la columna *pvalue-lvt* de la tabla 7.11, se acepta la hipótesis nula de igualdad de varianzas del Levene *test* para las 4 APIs de *flights* y *booking,* ya que los valores de *p-value* son mayores a 0.05, sin embargo, para las pruebas de Fligner-Killeen se rechaza la misma hipótesis nula para esas 4 APIs ya que los valores correspondientes de la columna *pvalue-fkt* son menores a 0.05, lo cual indica una contradicción en la validación de igualdad de varianzas para las 4 APIs. Por lo tanto, se realizarán las pruebas no paramétricas tanto de Wilcoxon-Mann-Whitney como de Brunner-Munzel.

En el caso del API *ticket create one,* se rechaza la hipótesis nula de igualdad de varianzas tanto en las pruebas de LVT como FKT debido a que sus *p-value* son 0. Por lo tanto, solo se usará la prueba de BM.

Tabla 7.11: Validación de comparación estadística de response time para la prueba oficial 1 FaaS vs CaaS

18-Nov	n1	n2	pvalue lvt	pvalue fkt	prueba valida
apiFlightsReadAll	1036	1002	0.74	0.001	WMW (BM)
apiFlightsReadOne	1000	980	0.201	0.004	WMW (BM)
apiBookingWrite	980	980	0.737	0.008	WMW (BM)
apiBookingReadAll	979	980	0.08	0	WMW (BM)
apiTicketCreateOne	700	700	0	0	BM

De acuerdo con el valor de la columna *pvalue-wct* de la tabla 7.12 para la API *flights read all* , se debería aceptar la hipótesis nula de igualdad de medias debido a que el *p-value* es mayor a 0.05, sin embargo, como el valor de la magnitud del tamaño de efecto es *"small",* esta prueba se considera inválida.

Entonces, para esa API solo se tomará en cuenta el segundo tipo de prueba válida, para la cual el valor de la columna *pvalue-bmt* también es mayor a 0.05*,* por lo tanto, el resultado de la comparación es "Iguales".

Para el caso de las 3 APIs *flights read one*, *booking write* y *booking read all,* de acuerdo con los valores de la columna pvalue-wct, se debería rechazar la hipótesis nula de igualdad de medias debido a que los *p-value* son menores a 0.05, sin embargo, como el valor de la magnitud del tamaño de efecto es "*small",* esta prueba se considera inválida. Entonces, para esas 3 APIs solo se tomará en cuenta el segundo tipo de prueba válida, para la cual, los valores de la columna *pvalue-bmt* son menores a 0.05, por lo tanto, se rechaza la hipótesis nula de igualdad de medias y los resultados de la comparación de medias son "Diferentes".

En el caso del API *ticket create one*, de acuerdo con el valor de la columna *pvalue-bmt*, se rechaza la hipótesis nula de igualdad de medias debido a que el *p-value* es 0, por lo tanto, el resultado de la comparación es "Diferentes".

Tabla 7.12: Resultado de comparación estadística de response time para la prueba oficial 1 FaaS vs CaaS

18-Nov	pvalue wct	eff.size wct	magn. wct	pvalue bmt	prueba valida	Resultado Comparación
apiFlightsReadAll	**0.236**	0.026	small	**0.242**	WMW (BM)	**Iguales** (Iguales)
apiFlightsReadOne	**0**	0.093	small	**0**	WMW (BM)	Diferentes (**Diferentes**)
apiBookingWrite	**0.003**	0.067	small	**0.003**	WMW (BM)	Diferentes (**Diferentes**)
apiBookingReadAll	**0**	0.117	small	**0**	WMW (BM)	Diferentes (**Diferentes**)
apiTicketCreateOne	**0**	0.126	small	**0**	BM	**Diferentes**

7.2.3.5 Monitoreo de nuevas instancias

En las figuras 7.20 y 7.21 se muestra el monitoreo de las nuevas instancias generadas para FaaS y CaaS.

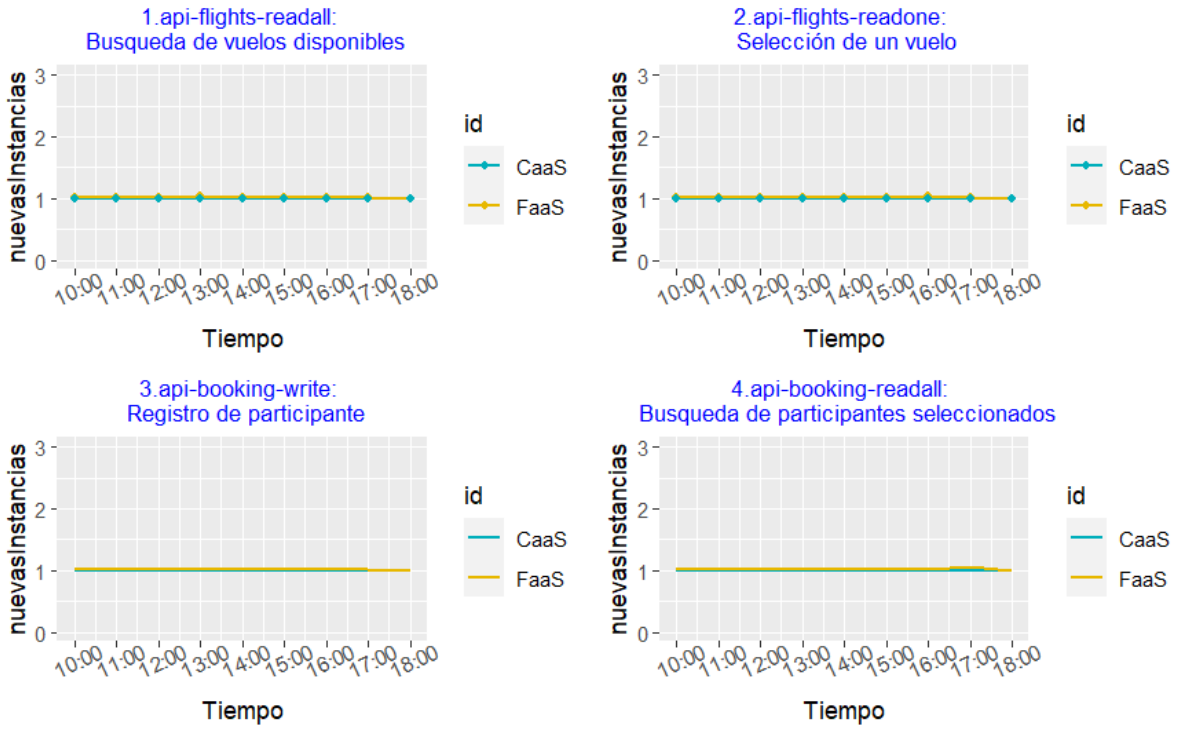

Figura 7.20: Gráfica time-series de las nuevas instancias generadas para las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

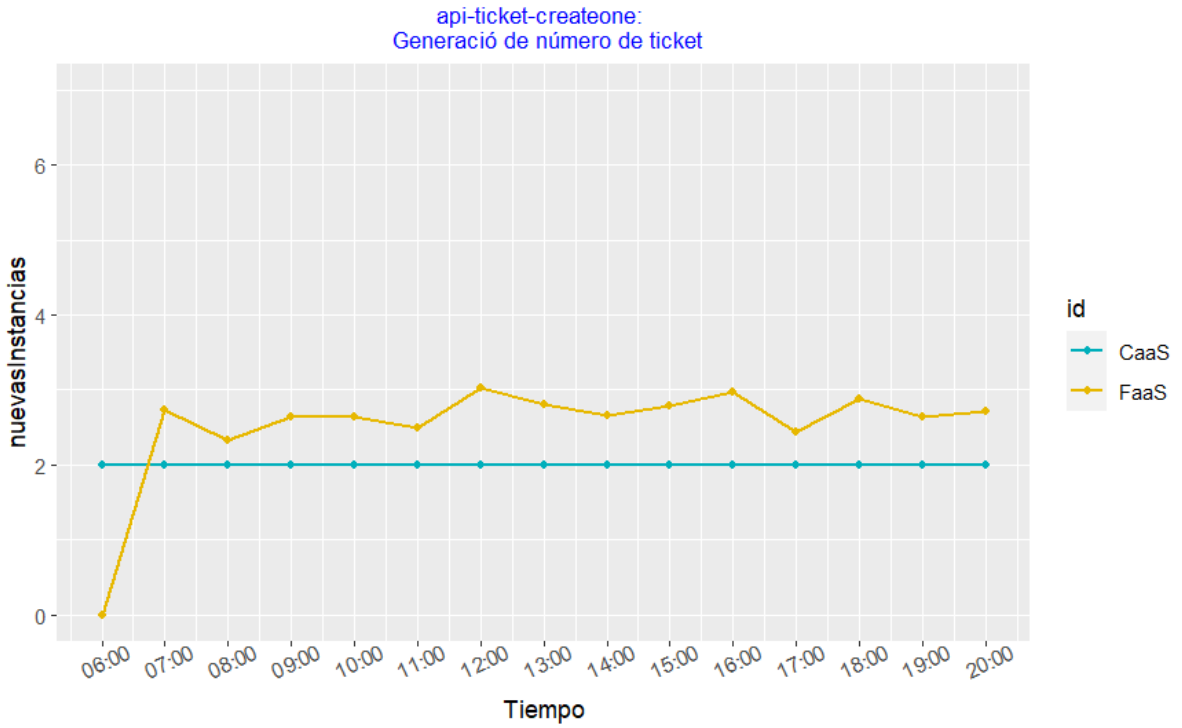

Figura 7.21: Gráfica time-series de las nuevas instancias generadas para el API ticket create one para la prueba oficial 1 de FaaS vs CaaS

7.2.3.6 Resultado de costos

En la figura 7.22 se muestra la suma de costos generados por los 5 APIs desplegados en FaaS con la configuración de memoria de 512MB y en CaaS con la configuración de CPU de los Fargate de CaaS con 0.5vCPU y memoria 1GB. Estos costos han sido calculados en 2 periodos, 10 y 60 minutos.

Se observa que en el periodo de 10 minutos el despliegue en CaaS cuesta alrededor de 16.62 centavos de dólar mientras que FaaS cuesta alrededor de 0.64 centavo de dólar.

Mientras que en el periodo de 60 minutos el despliegue en CaaS cuesta alrededor de 30.81 centavos de dólar mientras que FaaS mantiene su costo alrededor de 0.64 centavo de dólar.

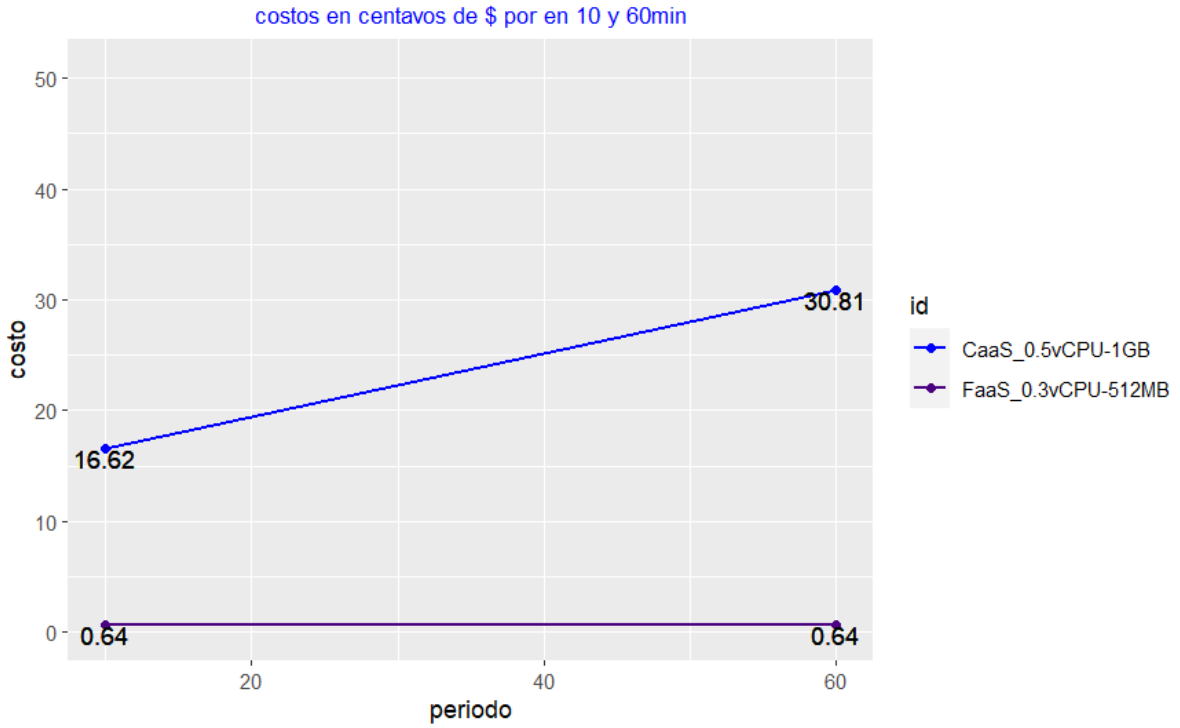

Figura 7.22: Gráfica de costos para todas las APIs para la prueba oficial 1 de FaaS vs CaaS

7.2.3.7 Discusión

- Los resultados de la tabla 7.12 indican que los valores medios de *response time* son significativamente diferentes para las 5 APIs, a excepción del API *flights read all*. Con esta información, se revisa nuevamente la tabla 7.10 y se deduce que los tiempos de respuesta de CaaS son efectivamente menores que FaaS para las 4 APIs *flights read one*, *booking write*, *booking read all* y *ticket create one*.
- Para el caso del API *flights read one* se considera que existe suficiente evidencia para afirmar que los valores medios de respuesta son iguales tanto para FaaS como CaaS
- En costos, para el periodo de 10 minutos, se observa que el despliegue en FaaS es al menos 25 veces más económico que CaaS.

7.2.4 Prueba de rendimiento oficial 2: FaaS vs CaaS con la configuración de recursos 2

Las pruebas se realizaron en las siguientes condiciones:

- Ejecución de JMeter por CLI desde Google *Cloud.*

- El tamaño de memoria configurado para los Lambdas de FaaS fue de 256MB.
- La configuración de CPU de los Fargate de CaaS es 0.25vCPU y memoria 1GB.

7.2.4.1 Monitoreo de consumo de recursos de la máquina virtual

En la figura 7.23 se observa que no hay incidentes en el uso de recursos de la VM de Google *Cloud*.

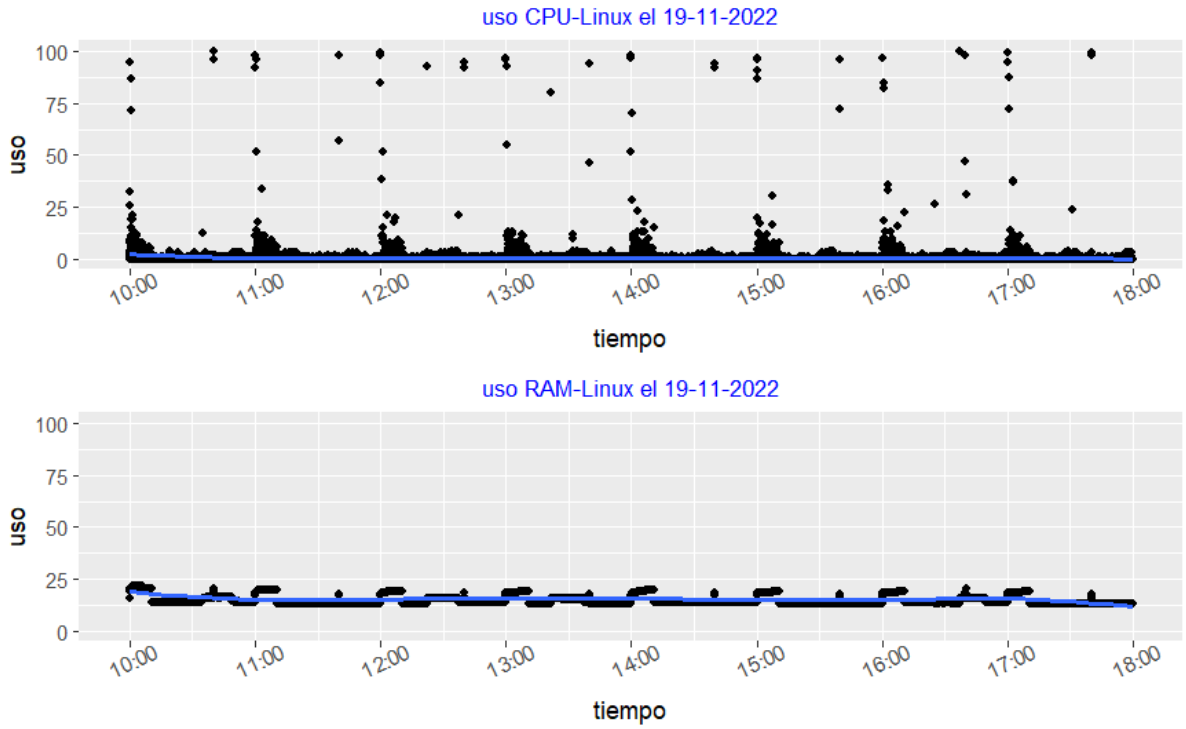

Figura 7.23: Consumo de recursos para la prueba oficial 1 de FaaS vs CaaS

7.2.4.2 Resultados de response code

En la tabla 7.13 se muestra el resumen de los *response code* por cada API, se observa que no hubo ningún error de JMeter (jmeter.error como código de respuesta) en todas las respuestas HTTP de las 5 APIs. El error 504 trata sobre sobre una limitación en el uso de recursos de la infraestructura de despliegue de las APIs y el 506 es un error personalizado de repetición de registro de usuario.

Las gráficas *time-series* del número de códigos de respuesta de las 5 APIs se muestran en las figuras 7.24 y 7.25, se observa que la cantidad de *response codes* exitosos (color verde) de FaaS siempre son mayores o iguales a CaaS y ambos no superan el valor de 70, aunque ese valor fue alcanzado en la prueba oficial 1 para el API *booking read all*. Esto se debe a una degradación del servicio CaaS y FaaS, ya que en esta prueba oficial se han reducido los recursos principales (CPU y RAM) a la mitad.

Para el caso del APIs *booking write,* si bien la cantidad de *response codes* exitosos se mantienen en 50 por hora respecto a la prueba oficial 1, la cantidad de respuestas de error personalizado 506 disminuyeron de 20 a 10, lo cual indica que la cantidad total de solicitudes HTTP disminuyeron de 70 a 60.

En el caso de la API *ticket create one,* la diferencia en la cantidad de *response codes* exitosos es muy grande a favor de FaaS, esto se debe a que CaaS presenta muchos errores de *timeout*, exactamente son 462 errores de 550 solicitudes HTTP. La causa raíz se puede deducir de la figura 7.31, presente en la sección 7.2.4.5 de monitoreo de nuevas instancias, que muestra como las instancias de FaaS escalan en mayor cantidad y más rápido que CaaS. Debido a estos resultados, se decidió ya no continuar con

la reducción de recursos para la prueba oficial 3, al contrario, se aumentaron respecto a la prueba oficial 1.

Tabla 7.13: Resumen de response code

Nombre del API	Fecha	Total Req	Res code	Res 506	Error
apiFlightsReadAll-FaaS	Sa:19-11	770	200	0	0
apiFlightsReadAll-CaaS	Sa:19-11	676	200	0	0
apiFlightsReadOne-FaaS	Sa:19-11	739	200	0	0
apiFlightsReadOne-CaaS	Sa:19-11	671	200	0	0
apiBookingWrite-FaaS	Sa:19-11	666	200, 506	116	0
apiBookingWrite-CaaS	Sa:19-11	666	200, 506	116	0
apiBookingReadAll-FaaS	Sa:19-11	660	200	0	0
apiBookingReadAll-CaaS	Sa:19-11	620	200	0	0
apiTicketCreateOne-FaaS	Sa:19-11	550	200	0	0
apiTicketCreateOne-CaaS	Sa:19-11	550	200, 504	0	462

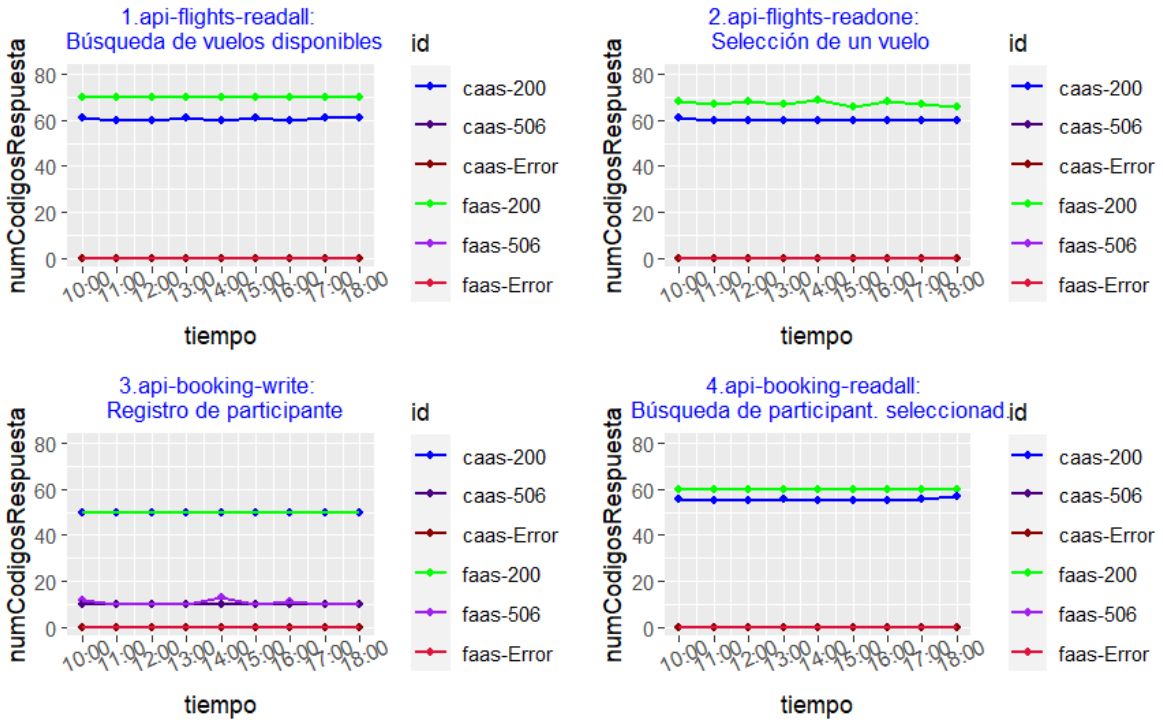

Figura 7.24: Gráfica time-series de response code de las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

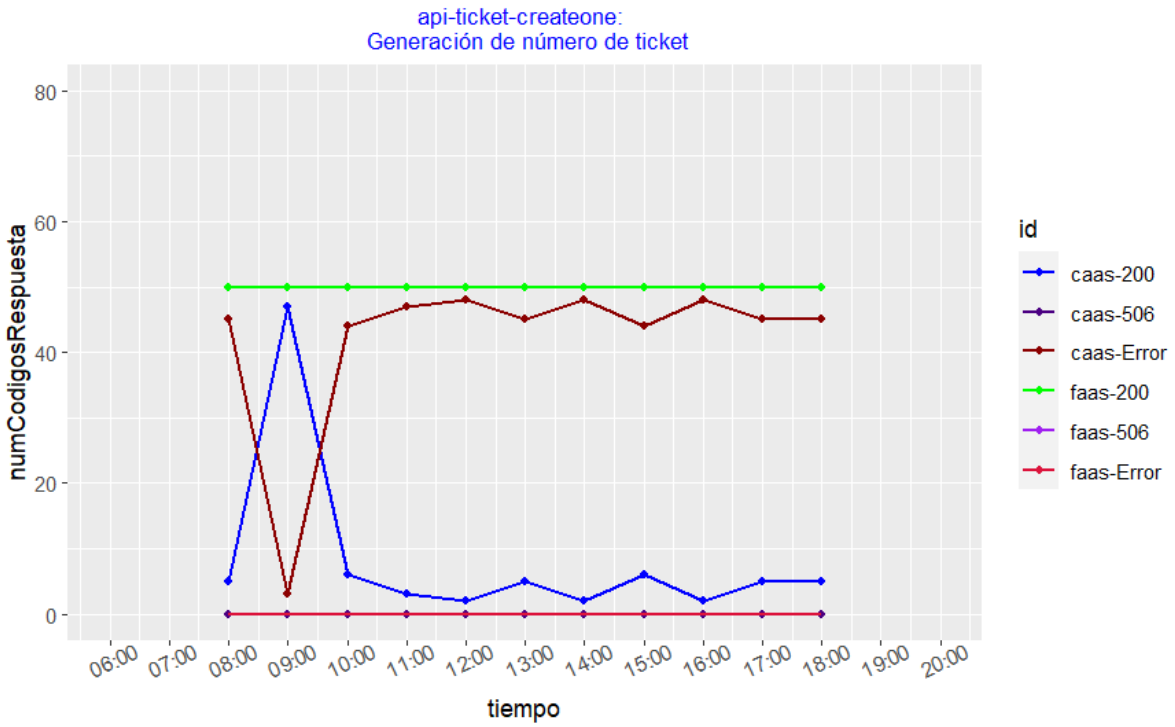

Figura 7.25: Gráfica time-series de response code del API ticket create one para la prueba oficial 1 de FaaS vs CaaS

7.2.4.3 Resultados de response time

En la tabla 7.14, se muestra el resumen estadístico de los *response time*.

En las figuras 7.26, 7.27, se observa que las gráficas de *time-series* tienen una línea que representa los valores promedio de los *response code* y estos valores coinciden aproximadamente con el *mean* de la tabla 7.14. Además, se observa que para las 4 APIs de *flights* y *booking,* el tiempo medio varía entre 203ms y 281ms, mientras que para el API *ticket create one* los tiempos medios están entre 19 seg. y 28 seg.

Tabla 7.14: Resumen estadístico de response time para la prueba oficial 2 FaaS vs CaaS

19-Nov	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
apiFlightsReadAll_FaaS	134	157	178	264.2889	287	2436
apiFlightsReadAll_CaaS	125	141	173	211.9706	271	782
apiFlightsReadOne_FaaS	113	136	151	232.4736	264	1762
apiFlightsReadOne_CaaS	107	120	240	190.4288	249	450
apiBookingWrite_FaaS	121	167	192	281.7344	305.75	1990
apiBookingWrite_CaaS	107	145	170	207.1389	275	435
apiBookingReadAll_FaaS	116	156	178	262.8278	284	1936
apiBookingReadAll_CaaS	103	140	170	203.906	271	349
apiTicketCreateOne_FaaS	11190	12891	22579.5	19142.4	22844	23801
apiTicketCreateOne_CaaS	6344	29079	29084	28174.4978	29209	29548

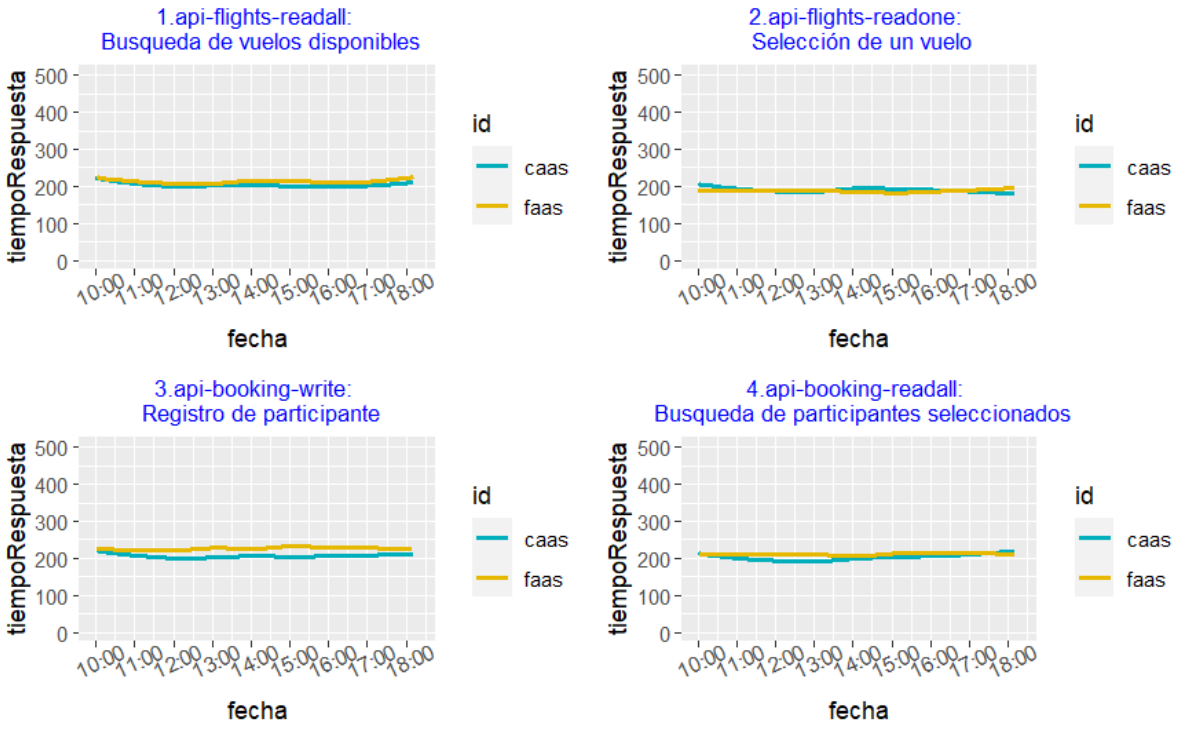

Figura 7.26: Gráfica time-series de response time de las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

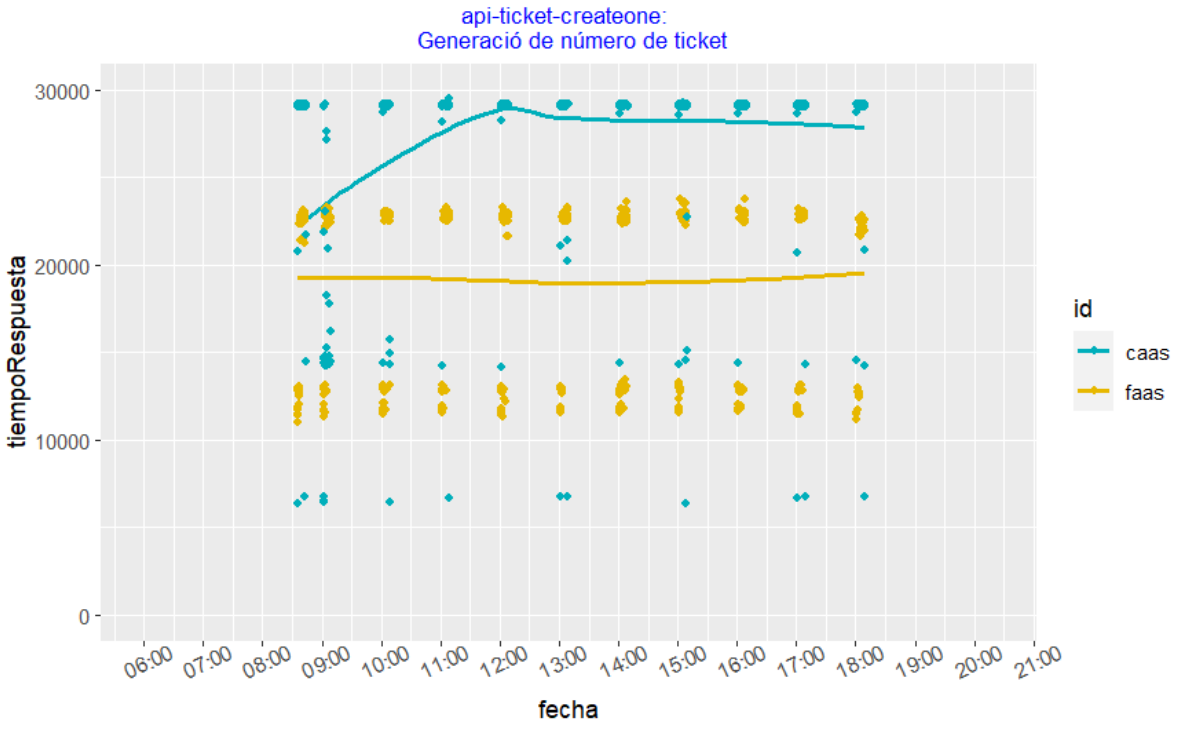

Figura 7.27: Gráfica time-series de response time del API ticket create one para la prueba oficial 1 de FaaS vs CaaS

7.2.4.4 Evaluación de diferencias estadísticamente significativa en los tiempos de respuesta

En las gráficas de *boxplot* de las figuras 7.28 y 7.29, se observa que es difícil apreciar la diferencia de medias en las 4 APIs de *flights* y *booking*, en cambio, para la del API *ticket create one* se observa una diferencia relevante de medias. El resultado final de la diferencia o igualdad de medias de las 5 APIs se determinará con los resultados de las tablas 7.15 y 7.16.

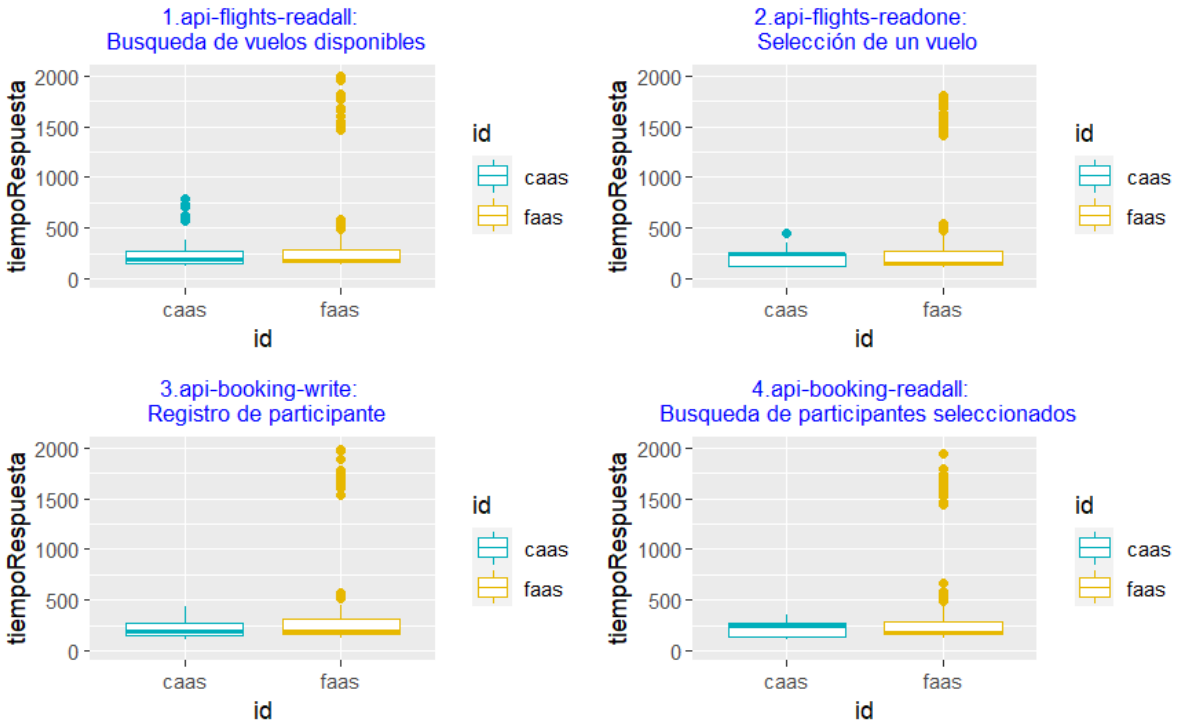

Figura 7.28: Gráfica boxplot de response time de las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

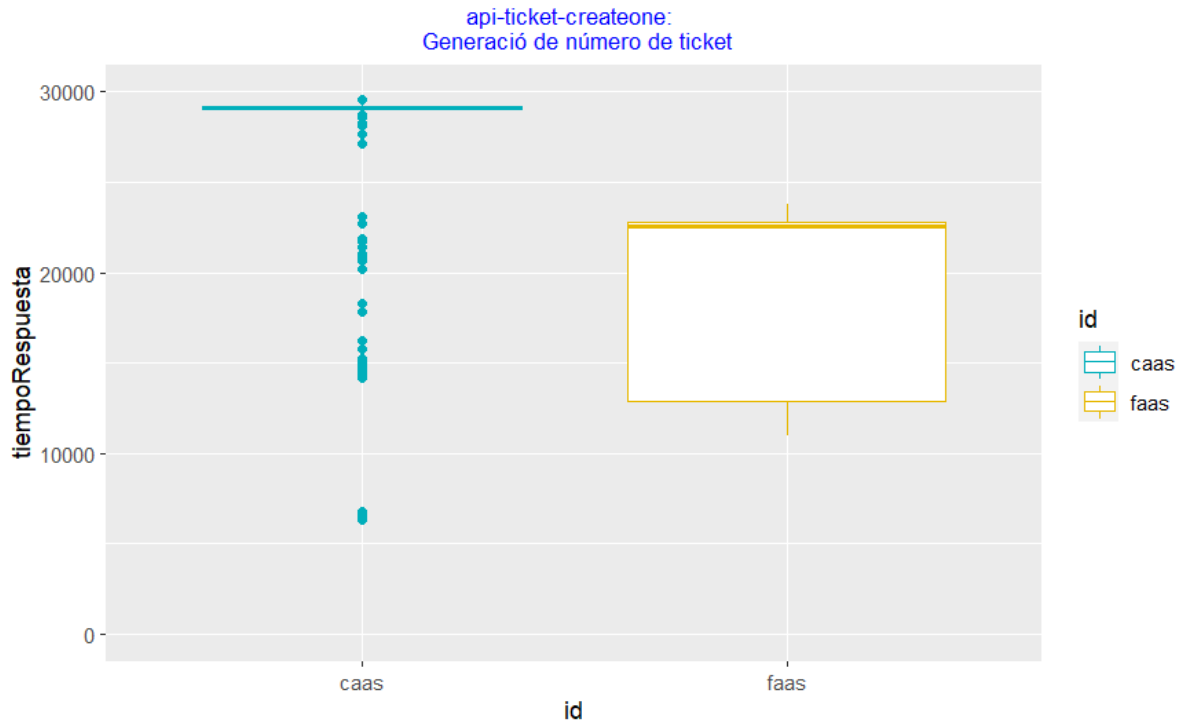

Figura 7.29: Gráfica boxplot de response time del API ticket create one para la prueba oficial 1 de FaaS vs CaaS

De acuerdo con los datos de la columna *pvalue-lvt* de la tabla 7.15, se acepta la hipótesis nula de igualdad de varianzas del Levene *test* para las 2 APIs de *flights read all* y *booking read all,* ya que los *p-value* son mayores a 0.05*,* sin embargo para las pruebas de Fligner-Killeen se rechaza la misma hipótesis nula para esas 2 APIs, ya que los valores correspondientes de la columna *pvalue-fkt* son menores a 0.05, lo cual indica una contradicción en la validación de igualdad de varianzas para las 2 APIs. Por lo tanto, se realizarán las pruebas no paramétricas tanto de Wilcoxon-Mann-Whitney como de Brunner-Munzel.

En el caso de las 3 2 APIs *flights read one*, y *booking write, ticket create one,* se rechaza la hipótesis nula de igualdad de varianzas tanto del Levene *test* como del Fligner-Killeen *test* ya que los valores de la columna *pvalue-lvt* y *pvalue-fkt* son menores a 0.05, por lo tanto, solo se usará la prueba de BM.

Tabla 7.15: Validación de comparación estadística de response time para la prueba oficial 2 FaaS vs CaaS

19-Nov	n1	n2	pvalue lvt	pvalue fkt	prueba valida
apiFlightsReadAll	676	770	0.705	0	WMW (BM)
apiFlightsReadOne	671	739	0.002	0	BM
apiBookingWrite	666	666	0.019	0	BM
apiBookingReadAll	620	660	0.288	0	WMW (BM)
apiTicketCreateOne	550	550	0	0	BM

De acuerdo con los datos de la columna *pvalue-wct* de la tabla 7.16, se debería rechazar las hipótesis nulas de igualdad de medias para las 2 APIs *flights read all* y *booking read all* debido a que los *p-value* son 0, pero como los valores de la magnitud del tamaño de efecto son "*small*", esta prueba se

considera inválida. Entonces, para esas APIs solo se tomará en cuenta el segundo tipo de prueba válida, para la cual, los valores de la columna *pvalue-bmt* también son 0, por lo tanto, los resultados de la comparación son “Diferentes”.

Para el caso de las 3 APIs *flights read one*, y *booking write* y *ticket create one,* de acuerdo con los valores de la columna *pvalue-bmt*, se rechaza la hipótesis nula de igualdad de medias debido a que el valor de *p-value* es 0, por lo tanto, los resultados de la comparación son “Diferentes”.

Tabla 7.16: Resultado de comparación estadística de response time para la prueba oficial 2 FaaS vs CaaS

19-Nov	pvalue wct	eff.size wct	magn. wct	pvalue bmt	prueba valida	Resultado Comparación
apiFlightsReadAll	**0**	0.183	small	**0**	WMW (BM)	Diferentes (**Diferentes**)
apiFlightsReadOne	**0**	0.213	small	**0**	BM	**Diferentes**
apiBookingWrite	**0**	0.247	small	**0**	BM	**Diferentes**
apiBookingReadAll	**0**	0.196	small	**0**	WMW (BM)	Diferentes (**Diferentes**)
apiTicketCreateOne	**0**	0.692	large	**0**	BM	**Diferentes**

7.2.4.5 Monitoreo de nuevas instancias

En las figuras 7.30 y 7.31 se muestra el monitoreo de las nuevas instancias generadas para FaaS y CaaS.

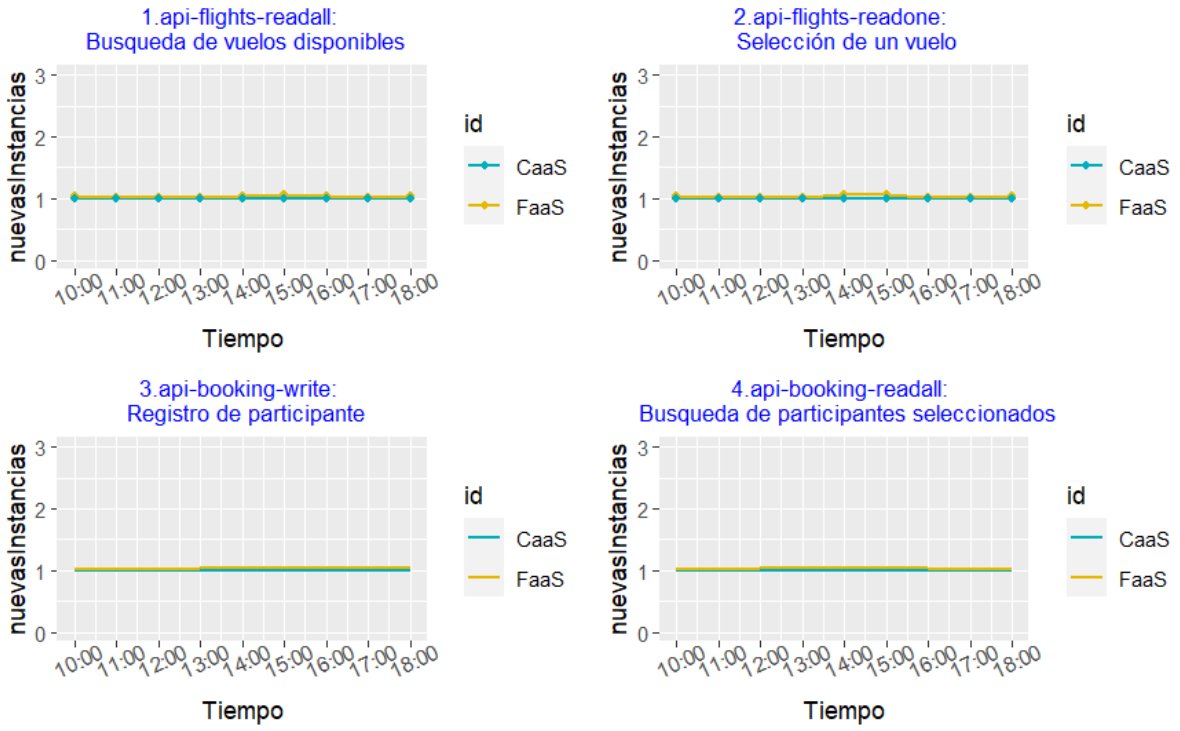

Figura 7.30: Gráfica time-series de las nuevas instancias generadas para las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

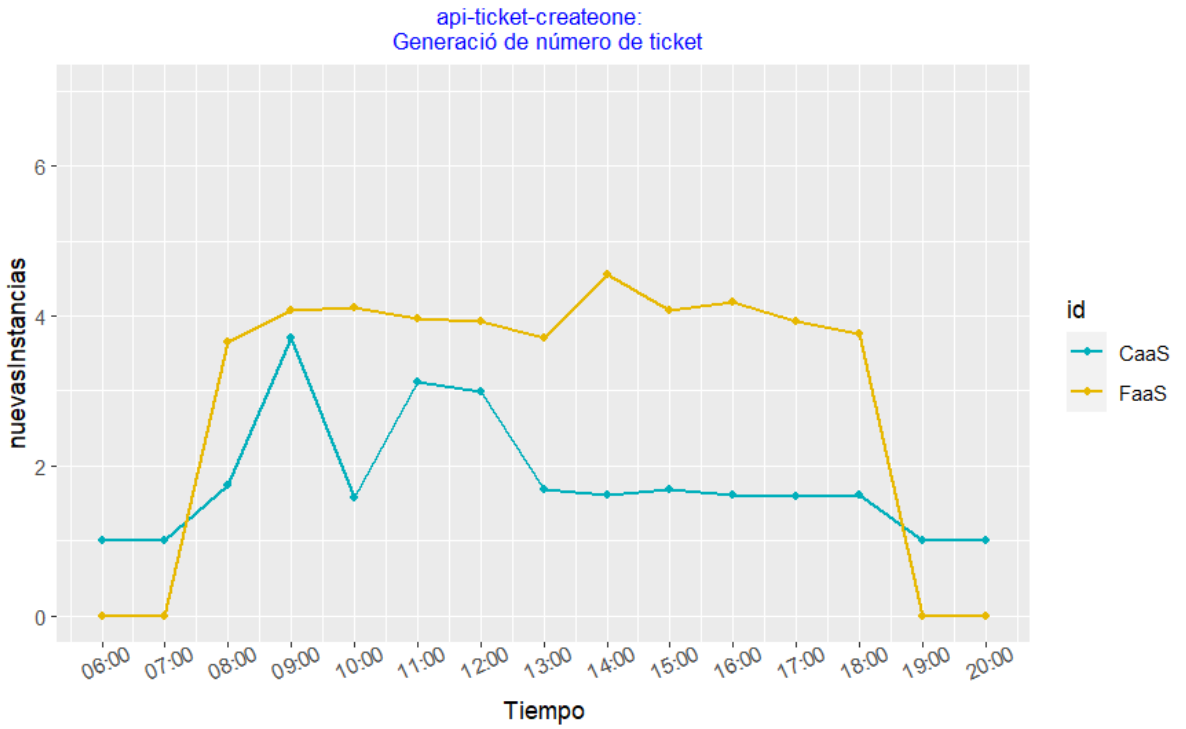

Figura 7.31: Gráfica time-series de las nuevas instancias generadas para el API ticket create one para la prueba oficial 1 de FaaS vs CaaS

7.2.4.6 Resultado de costos

En la figura 7.32 se muestra la suma de costos generados por los 5 APIs desplegados en FaaS con la configuración de memoria de 256MB y en CaaS con la configuración de CPU de los Fargate de CaaS con 0.25vCPU y memoria 1GB. Estos costos han sido calculados en 2 periodos, 10 y 60 minutos.

Se observa que en el periodo de 10 minutos el despliegue en CaaS cuesta alrededor de 16.71 centavos de dólar mientras que FaaS cuesta alrededor de 0.61 centavo de dólar.

Mientras que en el periodo de 60 minutos el despliegue en CaaS cuesta alrededor de 24.74 centavos de dólar mientras que FaaS mantiene su costo alrededor de 0.61 centavo de dólar.

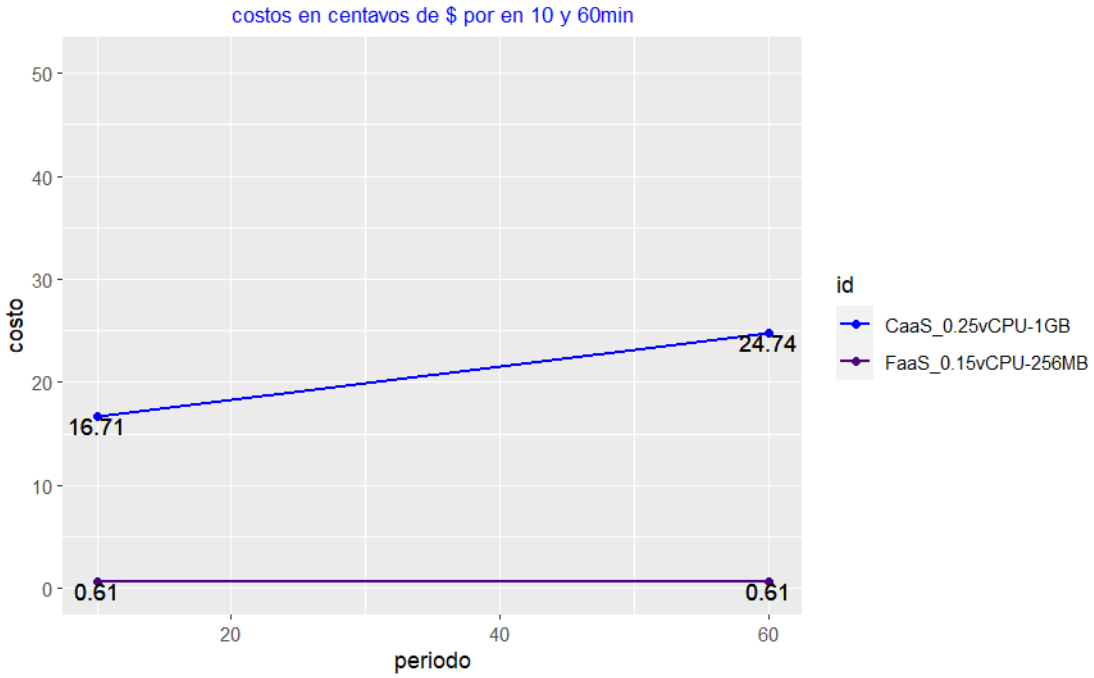

Figura 7.32: Gráfica de costos para todas las APIs para la prueba oficial 1 de FaaS vs CaaS

7.2.4.7 Discusión

- Los resultados de la tabla 7.16 indican que los valores medios de response time son significativamente diferentes para las 5 APIs. Revisando nuevamente la tabla 7.6 con esta información se deduce que los tiempos de respuesta de CaaS son efectivamente menores que FaaS para las 4 APIs búsqueda de *flights* y *booking*. Para el caso del API *ticket create one* FaaS presenta efectivamente tiempos menores de respuesta de CaaS. Hay que considerar que estos valores medios de *response time* no están considerando la degradación de servicio observado en la cantidad de solicitudes procesadas con errores por *timeout* del API *ticket create one*.
- Se generaron varios errores de *timeout* (462/550) en CaaS por lo cual se decidió ya no probar configuraciones de CPU y memoria más bajas y en lugar de eso la prueba oficial 3 fue con 1vCPU para CaaS y FaaS.
- Se observa que en las 4 APIs se presentan mejores resultados (menor tiempo de respuesta) en CaaS que en FaaS, estadísticamente validado por la prueba BM, por lo cual se deduce que cuando la configuración de memoria de un Lambda es menor o igual a 256MB, en el caso de los *requests* de pocos segundos de duración, su *performance* se degrada significativamente.
- Se observó que CaaS escala más lentamente que FaaS, lo cual también se comenta en [58].
- En costos, para el periodo de 10 minutos, se observa que el despliegue en FaaS es al menos 27 veces más económico que CaaS.

7.2.5 Prueba de rendimiento oficial 3: FaaS vs CaaS con la configuración de recursos 3

Las pruebas se realizaron en las siguientes condiciones:

- Ejecución de JMeter por CLI desde Google *Cloud.*
- Para las 4 APIs de tiempos de respuesta de promedio menor a 1 segundo:
    - El tamaño de memoria configurado para los Lambdas de FaaS fue de 128MB que corresponde aproximadamente a 0.07vCPU.
    - La configuración de CPU de los Fargate de CaaS es 0.25vCPU y memoria 512MB.

- Para el API ticket:
    - El tamaño de memoria configurado para los Lambdas de FaaS fue de 1769MB que corresponde a 1vCPU.
    - La configuración de CPU de los Fargate de CaaS es 1vCPU y memoria 2GB.

7.2.5.1 Monitoreo de consumo de recursos de la máquina virtual y los tipos de respuesta

En la figura 7.33 se observa que no hay incidentes en el uso de recursos de la VM de Google *Cloud*.

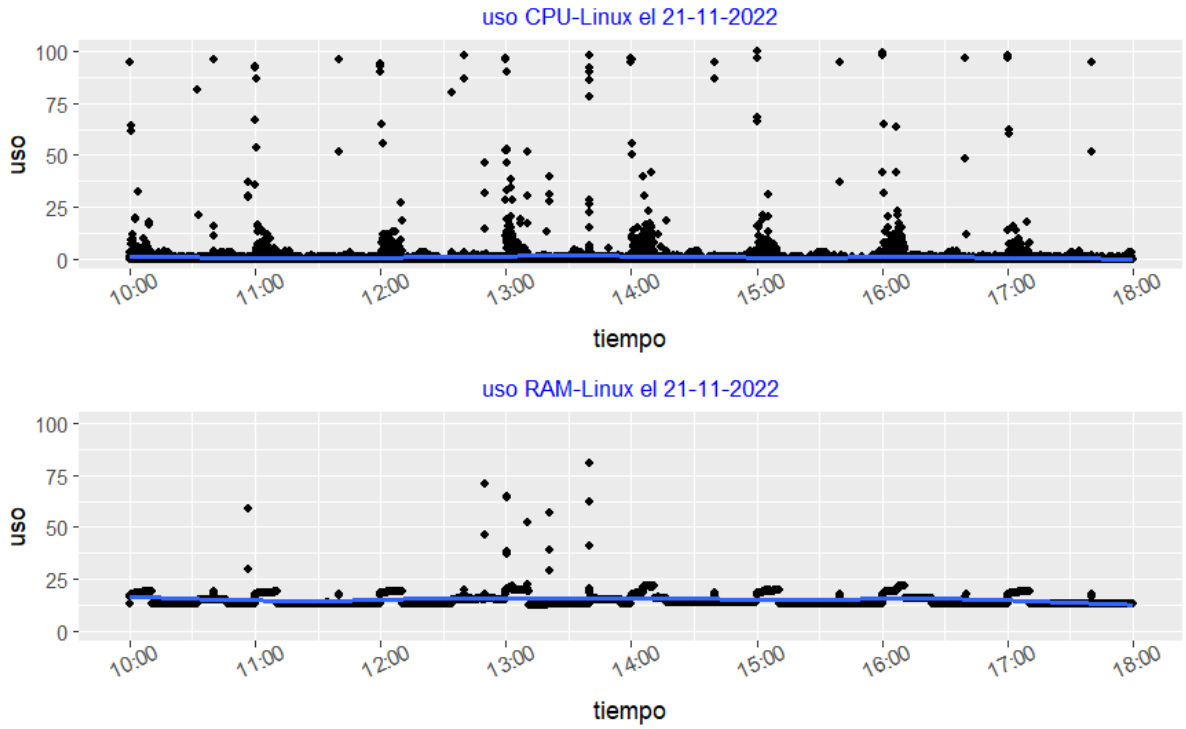

Figura 7.33: Consumo de recursos para la prueba oficial 1 de FaaS vs CaaS

7.2.5.2 Resultados de response code

En la tabla 7.17 se muestra el resumen de los *response code* por cada API, se observa que solo hubo 1 error de JMeter (jmeter.error como código de respuesta) en 880 respuestas HTTP para el API *flights read one*. Los errores 500, 504 tratan sobre limitaciones en el uso de recursos de la infraestructura de despliegue de las APIs y el 506 es un error personalizado de repetición de registro de usuario.

Las gráficas *time-series* del número de códigos de respuesta de las 5 APIs se muestran en las figuras 7.34 y 7.35, se observa que la cantidad de *response code* para FaaS y CaaS fueron similares y alrededor de 80 o 70 por hora, a excepción de las APIs *booking write* y *ticket create one* ya que ambas están desarrolladas para que solo acepten 50 solicitudes HTTP exitosas según lo indicado en la sección de modificaciones al software original.

Tabla 7.17: Resumen de response code

Nombre del API	Fecha	Total Req	Res code	Res 506	Error
apiFlightsReadAll-FaaS	Lu:21-11	880	200, 500	0	1
apiFlightsReadAll-CaaS	Lu:21-11	880	200	0	0
apiFlightsReadOne-FaaS	Lu:21-11	880	200, jmeter.error	0	1
apiFlightsReadOne-CaaS	Lu:21-11	880	200	0	0
apiBookingWrite-FaaS	Lu:21-11	781	200, 506	231	0
apiBookingWrite-CaaS	Lu:21-11	791	200, 506	241	0
apiBookingReadAll-FaaS	Lu:21-11	770	200	0	0
apiBookingReadAll-CaaS	Lu:21-11	770	200	0	0
apiTicketCreateOne-FaaS	Lu:21-11	550	200	0	0
apiTicketCreateOne-CaaS	Lu:21-11	550	200	0	0

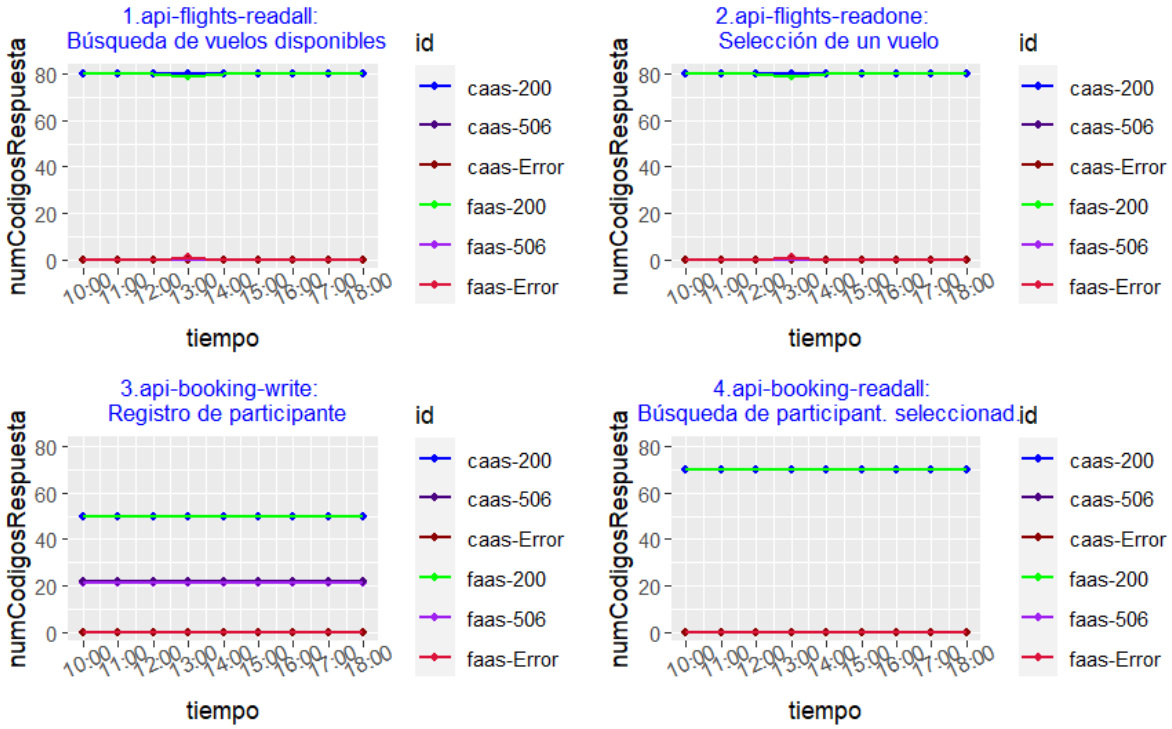

Figura 7.34: Gráfica time-series de response code de las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

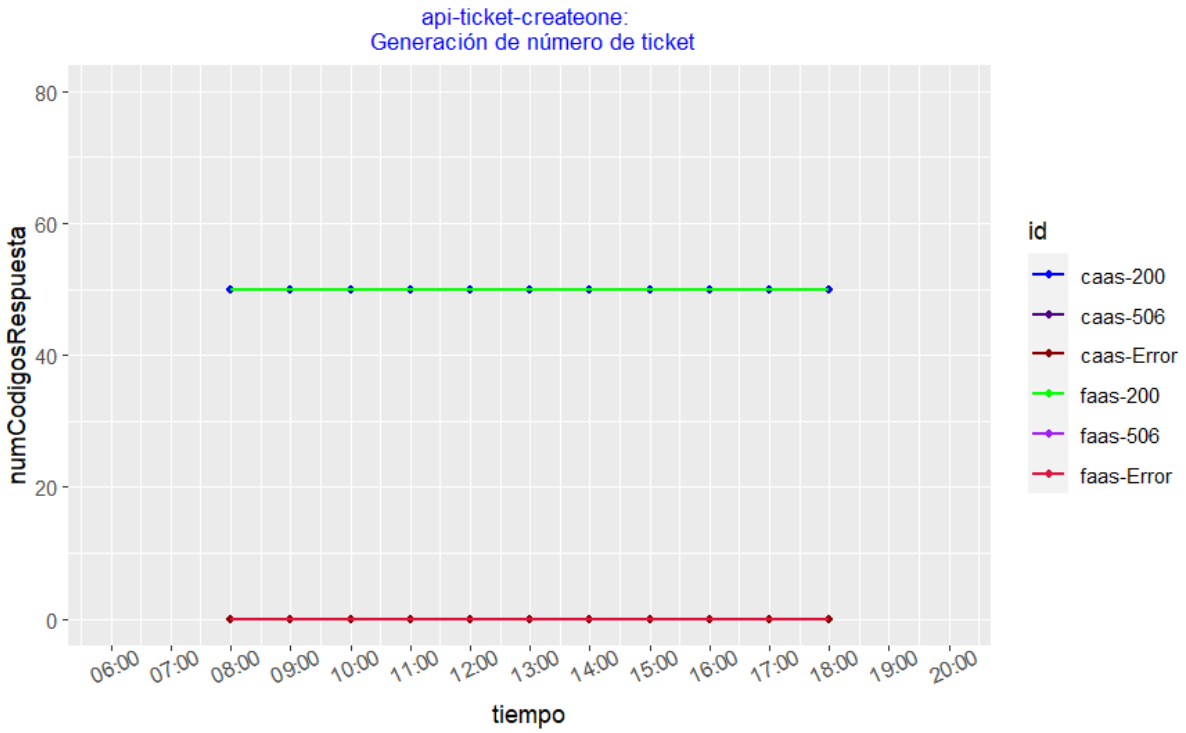

Figura 7.35: Gráfica time-series de response code del API ticket create one para la prueba oficial 1 de FaaS vs CaaS

7.2.5.3 Resultados de response time

En la tabla 7.18, se muestra el resumen estadístico de los *response time*.

En las figuras 7.16, 7.17, se observa que las gráficas de *time-series* tienen una línea que representa los valores promedio de los *response code* y estos valores coinciden aproximadamente con el *mean* de la tabla 7.10. Además, se observa que para las 4 APIs de *flights* y *booking,* el tiempo medio varía entre 176ms y 390ms, mientras que para el API *ticket create one,* los tiempos medios están entre 3.2 seg. y 4.1 seg.

Tabla 7.18: Resumen estadístico de response time para la prueba oficial 2 FaaS vs CaaS

21-Nov	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
apiFlightsReadAll_FaaS	164	211	260	362.0236	350	3053
apiFlightsReadAll_CaaS	125	147	168.5	212.3236	276.25	875
apiFlightsReadOne_FaaS	0	182	218	318.2472	318	2461
apiFlightsReadOne_CaaS	109	121	129	175.9236	251	1250
apiBookingWrite_FaaS	137	242.5	299	415.8372	396	3224
apiBookingWrite_CaaS	112	146	161	199.8549	276	850
apiBookingReadAll_FaaS	126	202.25	290	390.3508	351	2303
apiBookingReadAll_CaaS	110	122	131	176.0048	251	1155
apiTicketCreateOne_FaaS	1745	3322.25	3369.5	3225.6756	3475	3895
apiTicketCreateOne_CaaS	3569	3669.5	3744.5	4167.5289	3980	7407

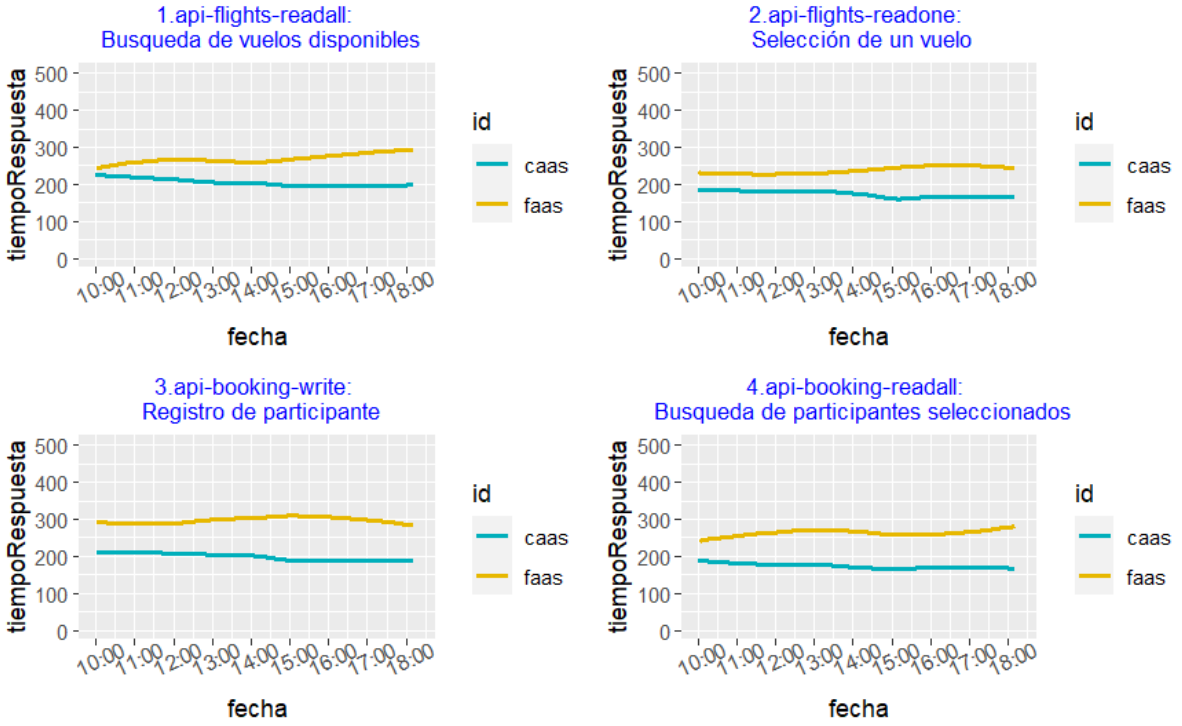

Figura 7.36: Gráfica time-series de response time de las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

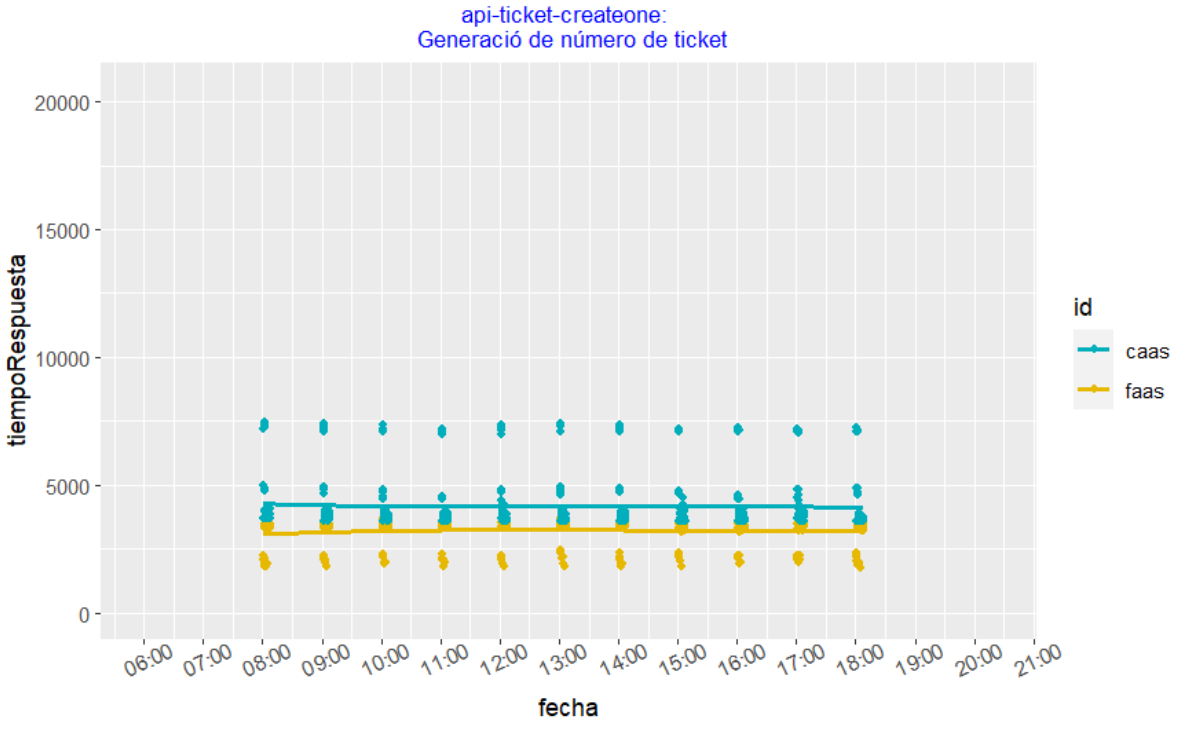

Figura 7.37: Gráfica time-series de response time del API ticket create one para la prueba oficial 1 de FaaS vs CaaS

7.2.5.4 Evaluación de diferencias estadísticamente significativa en los tiempos de respuesta

En las gráficas de *boxplot* de las figuras 7.38 y 7.39 se observa que es difícil apreciar la diferencia de medias en las 5 APIs. El resultado final de la diferencia o igualdad de medias se determinará con los resultados de las tablas 7.19 y 7.20.

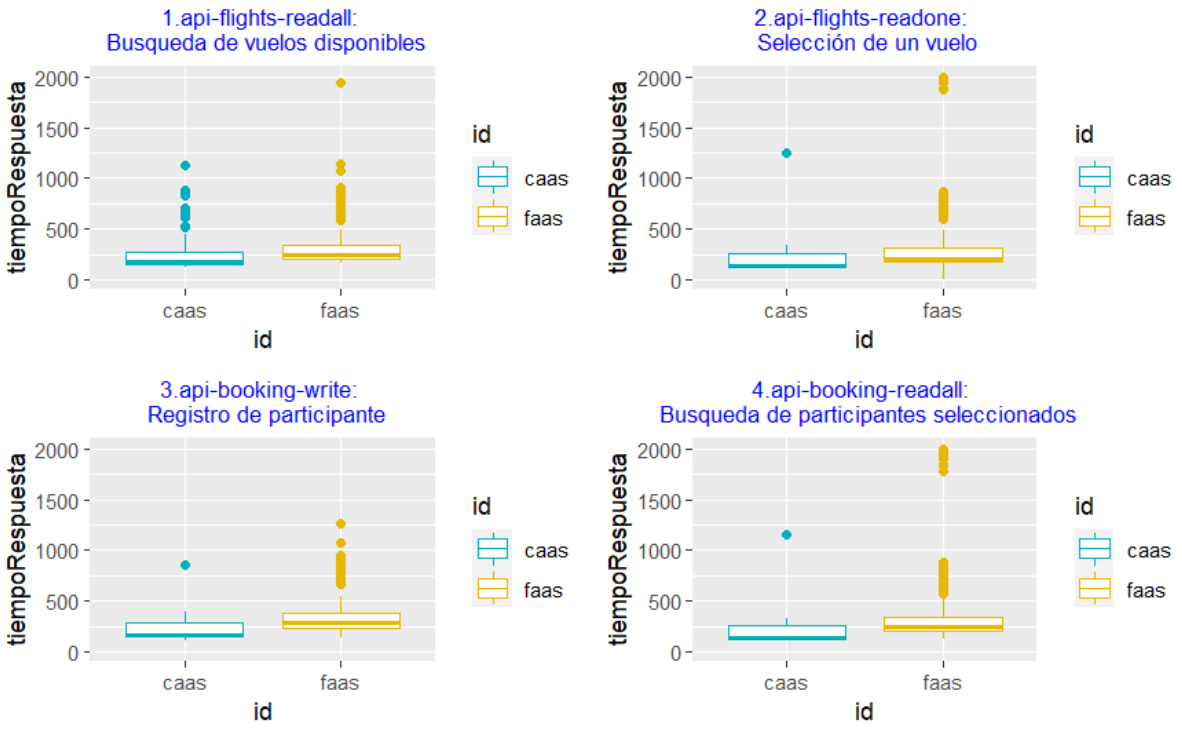

Figura 7.38: Gráfica boxplot de response time de las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

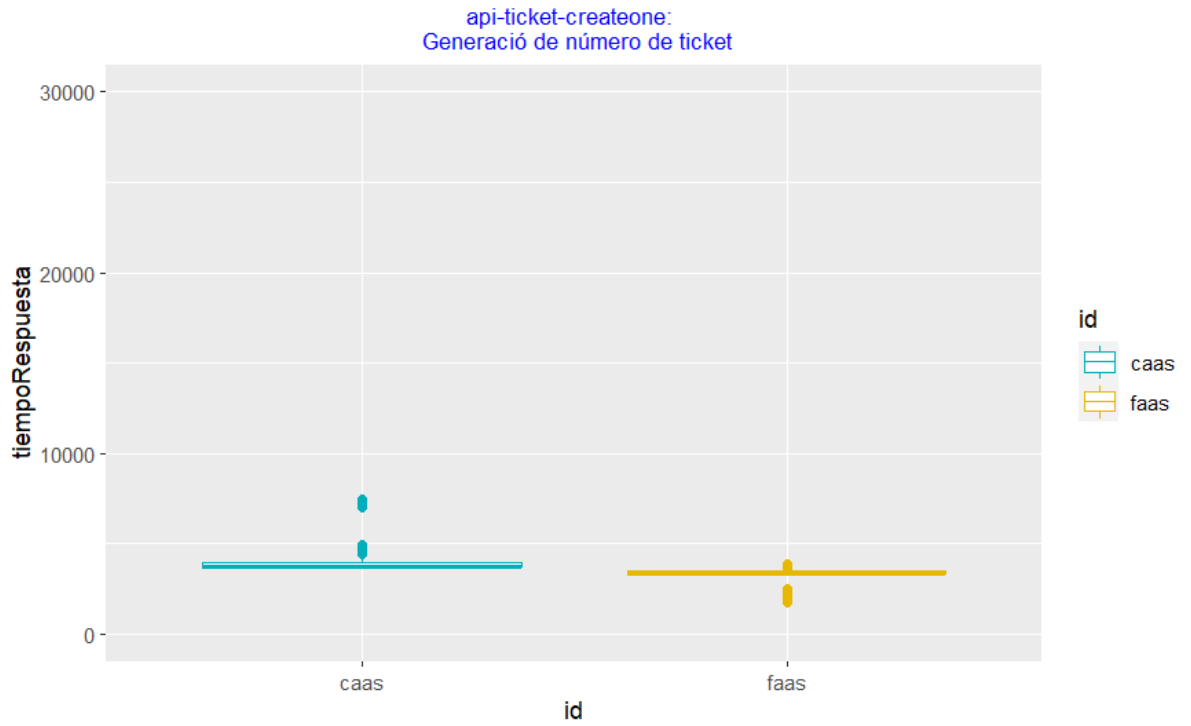

Figura 7.39: Gráfica boxplot de response time del API ticket create one para la prueba oficial 1 de FaaS vs CaaS

De acuerdo con los datos de la columna *pvalue-lvt* y *pvalue-fkt* de la tabla 7.19, se rechaza la hipótesis nula de igualdad de varianzas tanto de LVT como de FKT para las 5 APIs, por lo tanto, solo se usará las pruebas de Brunner-Munzel.

Tabla 7.19: Validación de comparación estadística de response time para la prueba oficial 2 FaaS vs CaaS

21-Nov	n1	n2	pvalue lvt	pvalue fkt	prueba valida
apiFlightsReadAll	880	880	0	0	BM
apiFlightsReadOne	880	880	0	0	BM
apiBookingWrite	791	781	0	0	BM
apiBookingReadAll	770	770	0	0	BM
apiTicketCreateOne	550	550	0	0	BM

De acuerdo con los datos de la columna pvalue-bmt de la tabla 7.20, se rechaza la hipótesis nula de igualdad de medias para las 5 APIs debido a que los *p-value* son 0, por lo tanto los resultados de la comparación son “Diferentes”.

Tabla 7.20: Resultado de comparación estadística de response time para la prueba oficial 2 FaaS vs CaaS

21-Nov	pvalue wct	eff.size wct	magn. wct	pvalue bmt	prueba valida	Resultado Comparación
apiFlightsReadAll	**0**	0.393	moder.	**0**	BM	**Diferentes**
apiFlightsReadOne	**0**	0.431	moder.	**0**	BM	**Diferentes**
apiBookingWrite	**0**	0.528	large	**0**	BM	**Diferentes**
apiBookingReadAll	**0**	0.494	moder.	**0**	BM	**Diferentes**
apiTicketCreateOne	**0**	0.85	large	**0**	BM	**Diferentes**

7.2.5.5 Monitoreo de nuevas instancias

En las figuras 7.40 y 7.41 se muestra el monitoreo de las nuevas instancias generadas para FaaS y CaaS.

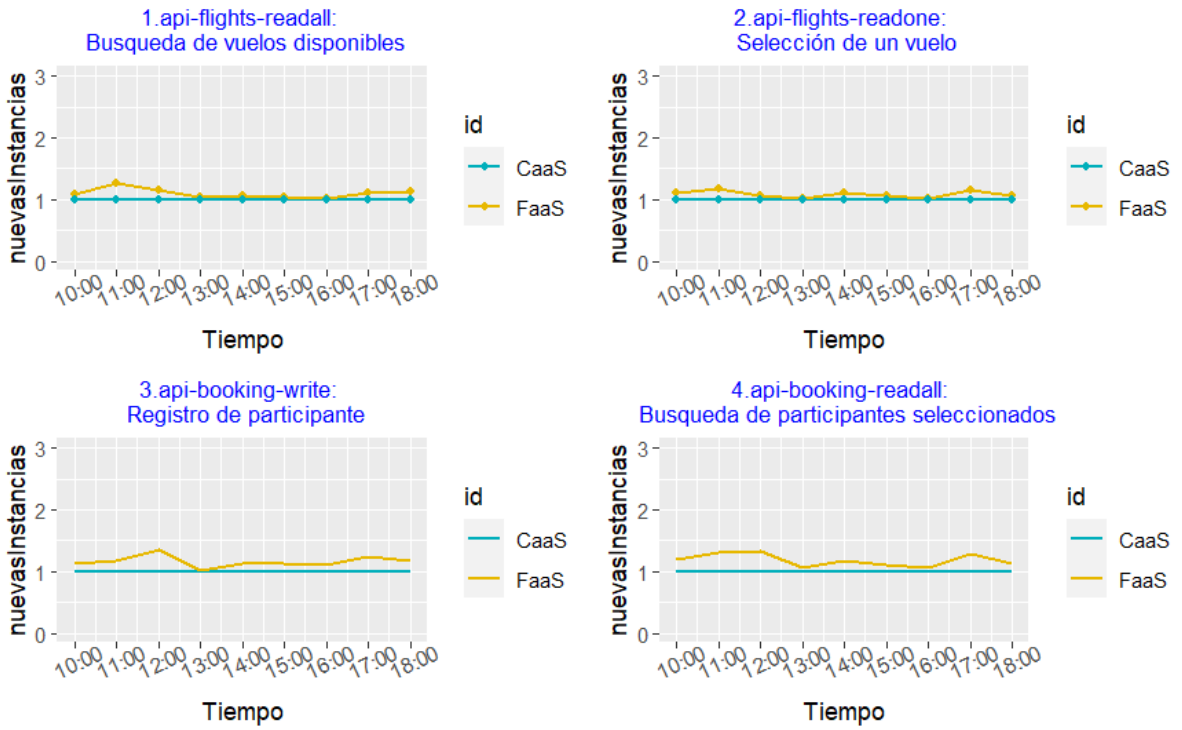

Figura 7.40: Gráfica time-series de las nuevas instancias generadas para las 4 APIs para la prueba oficial 1 de FaaS vs CaaS

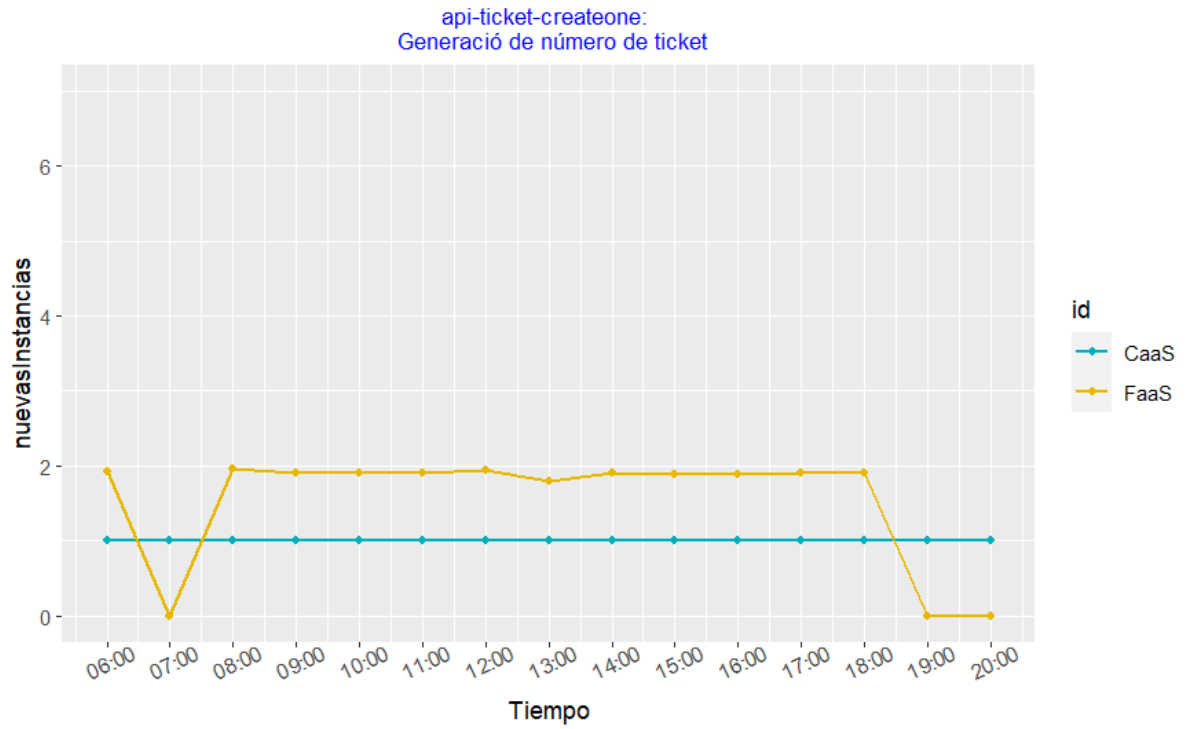

Figura 7.41: Gráfica time-series de las nuevas instancias generadas para el API ticket create one para la prueba oficial 1 de FaaS vs CaaS

7.2.5.6 Resultados de costos

En la figura 7.42 se muestra que la suma de costos generados por los 4 APIs de tiempos de respuesta de promedio menor a 1 segundo desplegados en FaaS con la configuración de memoria de 512MB y en CaaS con la configuración de CPU de los Fargate de CaaS con 0.5vCPU y memoria 1GB. Los cuales han sido sumados con los costos generados por el API ticket desplegado en FaaS con la configuración de memoria de 1769MB, que corresponde a 1vCPU y en CaaS con la configuración de CPU de los Fargate de CaaS con 1vCPU y memoria 2GB.

Estos costos han sido calculados en 2 periodos, 10 y 60 minutos.

Se observa que en el periodo de 10 minutos el despliegue en CaaS cuesta alrededor de 16.6 centavos de dólar mientras que FaaS cuesta alrededor de 1.03 centavo de dólar.

Mientras que en el periodo de 60 minutos el despliegue en CaaS cuesta alrededor de 27.85 centavos de dólar mientras que FaaS mantiene su costo alrededor de 1.03 centavo de dólar.

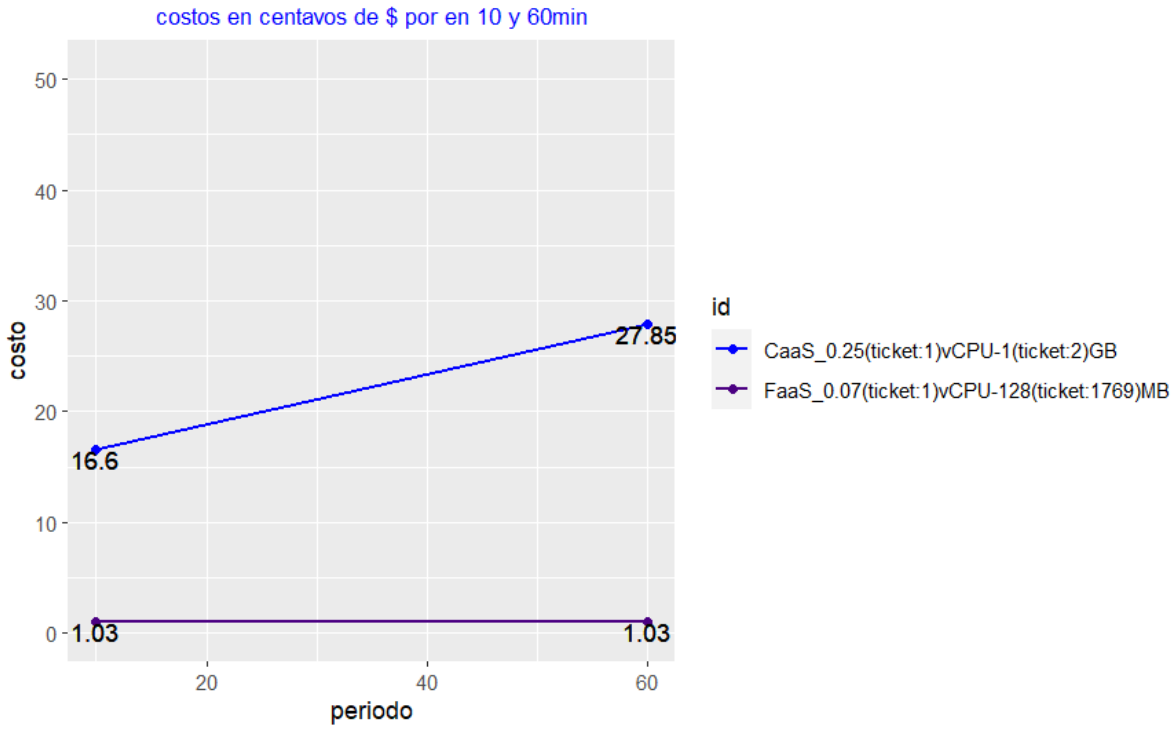

Figura 7.42: Gráfica de costos para todas las APIs para la prueba oficial 1 de FaaS vs CaaS

7.2.5.7 Discusión

- Los resultados de la tabla 7.20 indican que los valores medios de *response time* son significativamente diferentes para loas 5 APIs. Revisando nuevamente la tabla 7.18 con esta información se deduce que los tiempos de respuesta de CaaS son efectivamente menores que FaaS para las 4 APIs búsqueda de *flights* y *booking*. Para el caso del API *ticket create one,* FaaS presenta efectivamente tiempos menores de respuesta de CaaS.
- Al observar la figura 7.41 de monitoreo de nuevas instancias para el API *ticket create one,* se observa que las nuevas instancias para FaaS aumentaron automáticamente en el tiempo, lo cual le da ventaja de procesamiento sobre CaaS para cargas de trabajo que duran varios segundos.
- En costos, para el periodo de 10 minutos, se observa que el despliegue en FaaS es al menos 16 veces más económico que CaaS.

7.2.6 Evaluación de costos de los 3 días

En la figura 7.43 se muestra 4 gráficas, las generadas los 3 días y una gráfica ficticia considerando que se desplegaran 5 APIs del tipo ticket sobre un servicio FaaS con una configuración de memoria de 1769MB y en CaaS con una configuración de CPU de los Fargate de CaaS con 1vCPU y memoria 2GB.

Se observa en el periodo de 60min que los costos de CaaS son mayores que FaaS en todos los casos y en forma considerable respecto a FaaS.

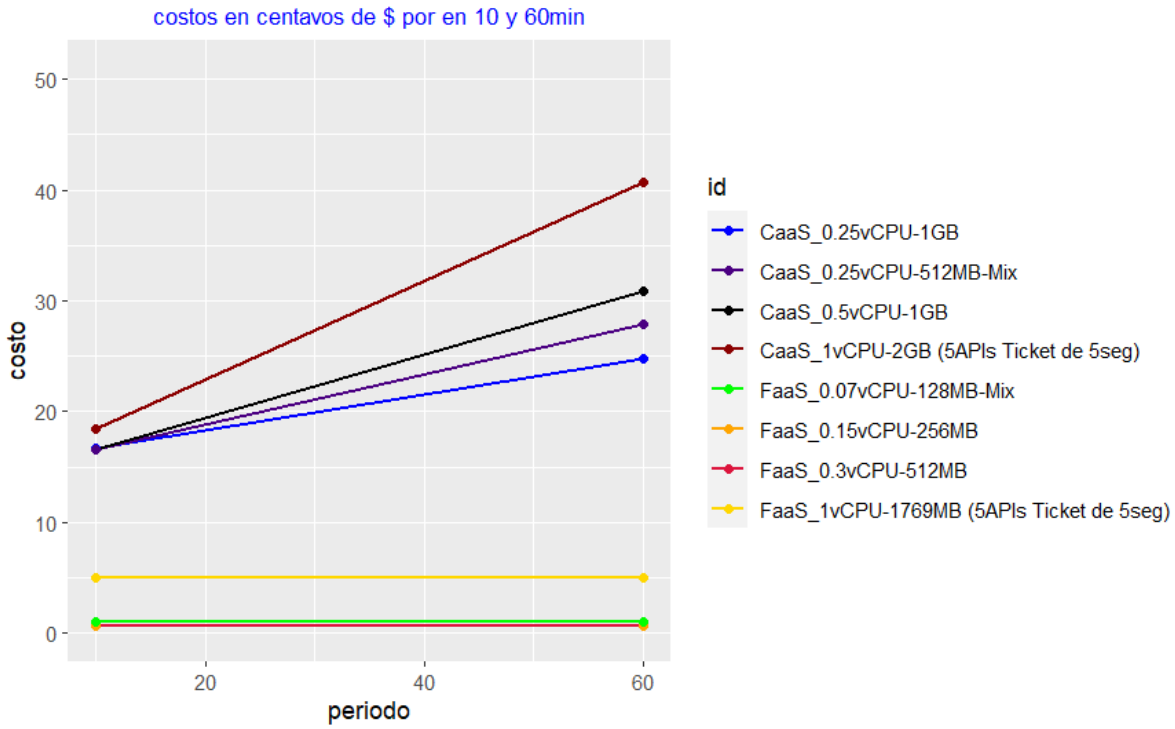

Figura 7.43: Gráfica de costos de las 3 pruebas de rendimiento

Además, se ha generado la gráfica de la figura 7.44 para otro caso ficticio, en el cual desplegarían 5 APIs del tipo *ticket* sobre un servicio FaaS con una configuración de memoria de 256MB y en CaaS con una configuración de CPU de los Fargate de CaaS con 0.25vCPU y memoria 1GB. Ambas configuraciones son las que normalmente se usan al empezar a probar ambos tipos de servicios

Se observa que los costos de CaaS empiezan a ser competitivos comparados con los de FaaS para este tipo de configuraciones de recursos cuando son utilizados para desplegar APIs con tiempos de duración de varios segundos. Esta mejora en los costos para CaaS se acentúa (en una proporción de 18 a 5 de Caas y FaaS) cuando el periodo evaluación es de 10 minutos, es decir cuanto más se acerca al tiempo de ejecución.

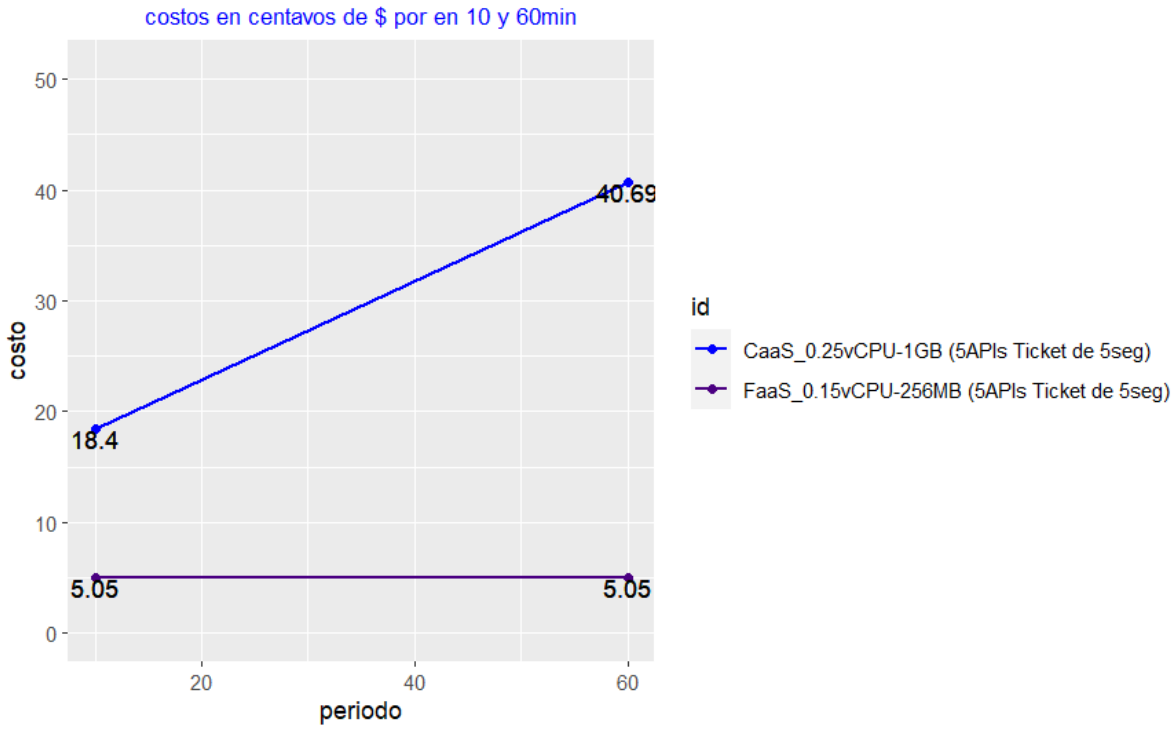

Figura 7.44: Gráfica de costos de un caso ficticio considerando 5 APIs con el mismo tiempo de duración que el API ticket

7.3 Discusión final de la evaluación

- Se comprueba que los despliegues en arquitecturas FaaS con una configuración de memoria de 256, 512, 1769 MB para aplicaciones que requieran tiempos de respuesta menores a 30 segundos presentan mejor *performance* y eficiencia en costos que CaaS en periodos de 10 y 60minutos para los 3 grupos de configuraciones: 0.25 vCPU con 1GB, 0.5 vCPU con 1GB y 1 vCPU con 2GB.
- Si bien los tiempos de respuesta de CaaS, para las APIs que requieran menos 1 segundo de ejecución fueron menores que FaaS la mayor parte del tiempo de evaluación, estas diferencias no son tan altas como para justificar las amplias diferencias de costos.
- Los costos generados por el servicio de CaaS son más competitivos comparados con los de FaaS en la medida que el tiempo de ejecución del API sea cercano al periodo de evaluación. Esto se debe a que el uso de recursos de la instancia de CaaS no ha pasado a cero a pesar de que la ejecución del API ha terminado y se sigue cobrando como si la instancia siguiera en uso.

8 CONCLUSIONES, RECOMENDACIONES Y TRABAJO FUTURO

8.1 Conclusiones del método y evaluación experimental

En la sección 3 se ha mostrado como se ha adaptado las metodologías identificadas en la revisión sistemática hacia un método que pueda ser usado por desarrolladores y arquitectos en escenarios cercanos a la realidad, como la modificación periódica de una app que no cumple con los objetivos de las pruebas de performance, para lo cual se usaron herramientas de desarrollo, despliegue y pruebas comúnmente usadas en la industria.

Se verificó que las pruebas de experimentación en las etapas oficiales se realizaron exitosamente respecto a la efectividad de ejecución de las pruebas de JMeter, ya que en la primera

prueba se presentó 1 error de JMeter entre 980 solicitudes HTTP, en la prueba 2 no hubo errores y en la prueba 3 se presentó 1 error entre 880. Estos resultados se deben al despliegue del ambiente de pruebas en una infraestructura con recursos suficientes de CPU y memoria, los cuales que fueron monitoreados por *scripts*. En el monitoreo no se detectaron incidentes con los recursos asignados.

En cuanto al análisis estadístico, se observó que las pruebas Brunner-Munzel (BM) siempre coincidieron con las pruebas de Wilcoxon-Mann-Whitney (WMV) cuando ambas eran válidas. Mientras que, cuando no se cumplía con los supuestos de WMW, las pruebas de BM presentaron resultados coherentes con lo esperado, lo que sugiere que este tipo de prueba, que es menos usada que la WMW, es confiable.

También, se observó que los *p-value* de las 4 pruebas de estadísticas LVT, FKT, WCT (con su *effect size*) y BMT son coherentes con los valores de *response time* de las tablas, a pesar de que no se ha depurado *outliers*.

Respecto a la optimización de costos, se lograron disminuir considerablemente, ya que se redujeron al menos en 4 veces como se explica en la sección 5.2.3.

Además, se determinó que es recomendable eliminar la infraestructura bajo pruebas, la cual se completa en pocos minutos mediante una sola línea de comando, cuando se tiene que realizar cambios en el *benchmark (*app de pruebas junto con su infraestructura de despliegue*)* ya que AWS tiene servicios que son cobrados por existir por cada hora dentro de la arquitectura CaaS. Esta operación no implica un retraso debido a tener que desplegar nuevamente ya que la arquitectura se puede volver a desplegar rápidamente mediante CDK al momento de recibir la solución por parte de desarrolladores y arquitectos.

Se consideró como hipótesis que FaaS tendría mejores resultados en costos que CaaS, lo cual se comprobó en los experimentos al observar que los servicios de FaaS fueran mucho más eficientes que CaaS, aunque no se pudo validar una configuración de parámetros de CaaS que permita un comportamiento de escalabilidad similar a FaaS.

Finalmente se observa que el *framework* CDK puede facilitar la comunicación entre analistas de pruebas de software, arquitectos y desarrolladores en las etapas de desarrollo ya que se puede explicar los cambios en la arquitectura en base a lenguajes de programación conocidos, como son JavaScript y Python.

8.2 Recomendaciones

Aunque la experimentación se enfocó en apps transaccionales sobre AWS, se puede realizar experimentos con otros tipos de apps en otras plataformas de nube utilizando mismo método, debido a que fue diseñado en base a documentación que no se limita a ningún tipo particular app o de infraestructura en la nube.

Para la ejecución de los experimentos se puede utilizar el *benchmark* que se ha desarrollado para esta investigación, el cual se ha publicado en GitHub. Aunque, esta aplicación se actualizará con un nuevo *framework* de *software*, se mantendrá el tipo de arquitectura basada en APIs REST.

Para el procesamiento estadístico de los CSVs generados en AWS se recomienda elaborar scripts personalizados en RStudio de acuerdo con las necesidades que requiera el proyecto de *testing*.

Verificar los parámetros configurados en CaaS, buscar que el resultado ofrezca un comportamiento de escalabilidad similar a FaaS. Se puede tomar de referencia el código fuente del script de CDK publicado en GitHub.

8.3 Trabajo futuro

Se utilizará el método propuesto para experimentar en otras plataformas de nube y posteriormente realizar una evaluación de *performance* de servicios FaaS y CaaS entre plataformas de nube, que es el tipo de evaluación más común en los *papers* revisados.

El procesamiento de los datos obtenido en CSV de AWS consumió bastante tiempo debido a la necesidad de personalizar scripts en RStudio. Por lo tanto, en un trabajo a futuro se organizará el código fuente de los scripts de tal forma que solo se necesite actualizar determinados parámetros para nuevos experimentos.

Los servicios de FaaS y CaaS son presentados por AWS como alternativas serverless, por lo tanto, debería existir configuraciones y escenarios en el cual CaaS es más eficiente en costos que FaaS. En consecuencia, en una investigación futura se identificará esos escenarios mediante experimentación.

REFERENCIAS BIBLIOGRÁFICAS

[1] A. Goli, O. Hajihassani, H. Khazaei, O. Ardakanian, M. Rashidi, and T. Dauphinee, “Migrating from monolithic to serverless: A fintech case study,” in *ICPE 2020 - Companion of the ACM/SPEC International Conference on Performance Engineering*, Association for Computing Machinery, Inc, Apr. 2020, pp. 20–25. doi: 10.1145/3375555.3384380.

[2] K. Burkat *et al.*, “Serverless containers - Rising viable approach to scientific workflows,” in *Proceedings - IEEE 17th International Conference on eScience, eScience 2021*, Institute of Electrical and Electronics Engineers Inc., Sep. 2021, pp. 40–49. doi: 10.1109/eScience51609.2021.00014.

[3] M. Copik, E. Zürich Switzerland Grzegorz Kwaśniewski ETH Zürich Switzerland Maciej Besta ETH Zürich Switzerland Michał Podstawski, and E. Zürich Switzerland, “SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing,” 2021.

[4] S. He, G. Manns, J. Saunders, W. Wang, L. Pollock, and M. Lou Soffa, “A statistics-based performance testing methodology for cloud applications,” *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, vol. 19, pp. 188–199, Aug. 2019, doi: 10.1145/3338906.3338912.

[5] M. Baruwal Chhetri, S. Chichin, Q. B. Vo, and R. Kowalczyk, “Smart CloudBench—A framework for evaluating cloud infrastructure performance,” *Information Systems Frontiers*, vol. 18, no. 3, pp. 413–428, Jun. 2016, doi: 10.1007/S10796-015-9557-2.

[6] N. Somu, N. Daw, U. Bellur, and P. Kulkarni, “PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications,” *2020 International Conference on COMmunication Systems and NETworkS, COMSNETS 2020*, pp. 144–151, Jan. 2020, doi: 10.1109/COMSNETS48256.2020.9027346.

[7] Z. Li, L. Obrien, and H. Zhang, “CEEM: A practical methodology for cloud services evaluation,” in *Proceedings - 2013 IEEE 9th World Congress on Services, SERVICES 2013*, 2013, pp. 44–51. doi: 10.1109/SERVICES.2013.73.

[8] A. Palade, A. Kazmi, and S. Clarke, “An evaluation of open source serverless computing frameworks support at the Edge,” *Proceedings - 2019 IEEE World Congress on Services, SERVICES 2019*, pp. 206–211, Jul. 2019, doi: 10.1109/SERVICES.2019.00057.

[9] AWS: Introducing AWS Fargate, “Introducing AWS Fargate.” [Online]. Available: https://aws.amazon.com/es/about-aws/whats-new/2017/11/introducing-aws-fargate-a-technology-to-run-containers-without-managing-infrastructure/

[10] S. Miller, T. Siems, and V. Debroy, “Kubernetes for Cloud Container Orchestration Versus Containers as a Service (CaaS): Practical Insights,” in *Proceedings - 2021 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW 2021*, Institute of Electrical and Electronics Engineers Inc., 2021, pp. 407–408. doi: 10.1109/ISSREW53611.2021.00110.

[11] AWS: Serverless on AWS, “Serverless on AWS: Build and run applications without thinking about servers.” [Online]. Available: https://aws.amazon.com/es/serverless/

[12] Docker: What is a container, “Use containers to Build, Share and Run your applications.” [Online]. Available: https://www.docker.com/resources/what-container/

[13] AWS: Serverless on AWS, “Serverless on AWS: Build and run applications without thinking about servers.” [Online]. Available: https://aws.amazon.com/es/serverless/

[14] AWS: CDK preguntas frecuentes, “Preguntas frecuentes sobre el kit de desarrollo de la nube de AWS,” 2022.

[15] J. Z. Gao, H.-S. J. Tsao, and Y. Wu, “Testing and Quality Assurance for Component-Based Software,” 2003.

[16] B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering, Technical Report EBSE-2007-01,” 2007.

[17] A. Ahmad, P. Jamshidi, and C. Pahl, “Protocol for Systematic Literature Review,” 2012.

[18] A. V. Papadopoulos *et al.*, “Methodological Principles for Reproducible Performance Evaluation in Cloud Computing,” *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1528–1543, Aug. 2021, doi: 10.1109/TSE.2019.2927908.

[19] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach, “BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms,” in *Proceedings - 2021 IEEE International Conference on Cloud Engineering, IC2E 2021*, Institute of Electrical and Electronics Engineers Inc., 2021, pp. 71–82. doi: 10.1109/IC2E52221.2021.00014.

[20] I. Pelle, J. Czentye, J. Doka, A. Kern, B. P. Gero, and B. Sonkoly, “Operating Latency Sensitive Applications on Public Serverless Edge Cloud Platforms,” *IEEE Internet Things J*, vol. 8, no. 10, pp. 7954–7972, May 2021, doi: 10.1109/JIOT.2020.3042428.

[21] I. Pelle, J. Czentye, J. Doka, and B. Sonkoly, “Towards Latency Sensitive Cloud Native Applications: A Performance Study on AWS,” in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, IEEE, Jul. 2019, pp. 272–280. doi: 10.1109/CLOUD.2019.00054.

[22] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, “Serverless computing: An investigation of factors influencing microservice performance,” *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, pp. 159–169, May 2018, doi: 10.1109/IC2E.2018.00039.

[23] J. Czentye, I. Pelle, A. Kern, B. P. Gero, L. Toka, and B. Sonkoly, “Optimizing latency sensitive applications for amazon’s public cloud platform,” *2019 IEEE Global Communications Conference, GLOBECOM 2019 - Proceedings*, Dec. 2019, doi: 10.1109/GLOBECOM38437.2019.9013988.

[24] S. Werner, R. Girke, and J. Kuhlenkamp, “An Evaluation of Serverless Data Processing Frameworks,” *WOSC 2020 - Proceedings of the 2020 6th International Workshop on Serverless Computing, Part of Middleware 2020*, pp. 19–24, Dec. 2020, doi: 10.1145/3429880.3430095.

[25] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, “FaaSdom: A benchmark suite for serverless computing,” *DEBS 2020 - Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, pp. 73–84, Jul. 2020, doi: 10.1145/3401025.3401738.

[26] S. Eismann *et al.*, “A case study on the stability of performance tests for serverless applications,” *Journal of Systems and Software*, vol. 189, Jul. 2022, doi: 10.1016/j.jss.2022.111294.

[27] Y. Gan *et al.*, “An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems,” *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pp. 3–18, Apr. 2019, doi: 10.1145/3297858.3304013.

[28] M. Villamizar *et al.*, “Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures,” *Service Oriented Computing and Applications*, vol. 11, no. 2, pp. 233–247, Jun. 2017, doi: 10.1007/s11761-017-0208-y.

[29] M. Aly, F. Khomh, and S. Yacout, “On the Performance Implications of Deploying IoT Apps as FaaS,” *Communications in Computer and Information Science*, vol. 1534 CCIS, pp. 25–47, 2022, doi: 10.1007/978-3-030-96040-7_3/COVER.

[30] L. Muller, C. Chrysoulas, N. Pitropakis, and P. J. Barclay, “A traffic analysis on serverless computing based on the example of a file upload stream on aws lambda,” *Big Data and Cognitive Computing*, vol. 4, no. 4, pp. 1–29, Dec. 2020, doi: 10.3390/bdcc4040038.

[31] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, “Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions,” *Future Generation Computer Systems*, vol. 110, pp. 502–514, Sep. 2020, doi: 10.1016/J.FUTURE.2017.10.029.

[32] C. Tsigkanos, M. Garriga, L. Baresi, and C. Ghezzi, “Cloud Deployment Tradeoffs for the Analysis of Spatially Distributed Internet of Things Systems,” *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, p. 17, Apr. 2020, doi: 10.1145/3381452.

[33] C. F. Fan, A. Jindal, and M. Gerndt, “Microservices vs serverless: A performance comparison on a cloud-native web application,” in *CLOSER 2020 - Proceedings of the 10th International Conference on Cloud Computing and Services Science*, SciTePress, 2020, pp. 204–215. doi: 10.5220/0009792702040215.

[34] A. Pogiatzis and G. Samakovitis, “An Event-Driven Serverless ETL Pipeline on AWS,” *Applied Sciences 2021, Vol. 11, Page 191*, vol. 11, no. 1, p. 191, Dec. 2020, doi: 10.3390/APP11010191.

[35] A. Vittorio Papadopoulos *et al.*, “PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications,” *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 1, no. 4, Aug. 2016, doi: 10.1145/2930659.

[36] D. Kallergis, J. Tsantilis, and C. Douligeris, “Performance evaluation of cloud systems: A behavioural approach,” *2015 IEEE International Symposium on Signal Processing and Information Technology, ISSPIT 2015*, pp. 409–414, Jan. 2016, doi: 10.1109/ISSPIT.2015.7394370.

[37] G. Kousiouris *et al.*, “A Multi-Cloud Framework for Measuring and Describing Performance Aspects of Cloud Services Across Different Application Types.,” *CLOSER 2014 - Proceedings of the 4th International Conference on Cloud Computing and Services Science*, pp. 714–721, 2014, doi: 10.5220/0004975407140721.

[38] Z. Li, L. O’Brien, H. Zhang, and R. Cai, “A factor framework for experimental design for performance evaluation of commercial cloud services,” *CloudCom 2012 - Proceedings: 2012 4th IEEE International Conference on Cloud Computing Technology and Science*, pp. 169–176, 2012, doi: 10.1109/CLOUDCOM.2012.6427525.

[39] G. R. Garay, A. Tchernykh, and A. Y. Drozdov, “An approach for the performance evaluation of multi-tier cloud applications,” *Proceedings - 2nd International Conference on Engineering and Telecommunication, En and T 2015*, pp. 63–66, Feb. 2016, doi: 10.1109/ENT.2015.19.

[40] K. Sreenivasan, “On a framework for analyzing cloud system performance,” pp. 11–11, May 2012, doi: 10.1109/NCCISP.2012.6189677.

[41] A. Jacob and C. Raj, “Testing Methodologies for Cloud Performance,” *International Journal of Innovative Technology and Exploring Engineering*, 2019.

[42] B. L. Muhammad-Bello and M. Aritsugi, “A transparent approach to performance analysis and comparison of infrastructure as a service providers,” *Computers & Electrical Engineering*, vol. 69, pp. 317–333, Jul. 2018, doi: 10.1016/J.COMPELECENG.2017.12.034.

[43] J. Wan, X. Yang, Z. Ren, and Z. Ye, “Performance Evaluation and Modeling Method Research Based on IaaS Cloud Platform,” *International Journal of Grid and Distributed Computing*, vol. 9, no. 10, pp. 141–152, 2016, doi: 10.14257/ijgdc.2016.9.10.13.

[44] H. Jeon, Y. G. Min, and K. K. Seo, “A performance measurement framework of cloud storage services,” *Indian J Sci Technol*, vol. 8, pp. 105–111, 2015, doi: 10.17485/IJST/2015/V8IS8/64230.

[45] G. Ataş and V. C. Gungor, “Performance evaluation of cloud computing platforms using statistical methods,” *Computers & Electrical Engineering*, vol. 40, no. 5, pp. 1636–1649, Jul. 2014, doi: 10.1016/J.COMPELECENG.2014.03.017.

[46] H. Jeon, Y. G. Min, and K. K. Seo, “A framework of performance measurement monitoring of cloud service infrastructure system for service activation,” *International Journal of Software Engineering and its Applications*, vol. 8, no. 5, pp. 127–138, 2014, doi: 10.14257/IJSEIA.2014.8.5.11.

[47] C. Liu, M. Yu, and Y. Zhang, “A Nonparametric Multivariate Method for Performance Analysis of Virtual Machines in Cloud Computing Systems,” *Modelling, Identification and Control*, pp. 95–98, Mar. 2014, doi: 10.2316/P.2014.809-048.

[48] Y. Qin, “A Evaluation Method of System Performance Based on Cloud Theory,” *Applied Mechanics and Materials*, vol. 530–531, pp. 496–501, 2014, doi: 10.4028/WWW.SCIENTIFIC.NET/AMM.530-531.496.

[49] Y.-Q. You, J., Dong, X.-L., Luo, J.-B., Sun, “Performance monitoring and evaluation methods of computing system under cloud mode,” *Xitong Fangzhen Xuebao / Journal of System Simulation*, 2013.

[50] Q. He, Z. Li, L. Wang, H. Wang, and J. Sun, “Performance Measurement Technique of Cloud storage system,” 2013, doi: 10.2991/ICCSEE.2013.721.

[51] A. Avritzer, M. Camilli, A. Janes, and B. Russo, “PPTAMλ : What, Where, and How of Cross-domain Scalability Assessment,” 2021.

[52] AWS: Lambda Pricing, “AWS Lambda Pricing.”

[53] AWS: Fargate pricing, “AWS Fargate Pricing.”

[54] AWS: PrivateLink pricing, “AWS PrivateLink pricing,” 2022.

[55] AWS: Amazon VPC pricing, “Amazon VPC pricing.”

[56] AWS: Architect an Airline Booking Application, “Architect an Airline Booking Application, End-to-End.” Accessed: Aug. 29, 2022. [Online]. Available: https://pages.awscloud.com/GLOBAL-devstrategy-OE-BuildOnServerless-2019-reg-event.html

[57] M. W. Fagerland, “t-tests, non-parametric tests, and large studies—a paradox of statistical practice?,” 2012.

[58] V. Ionescu, “Scaling containers on AWS in 2022,” 2022.

ANEXOS

A. Código CDK

A.1. Código de FaaS

En las siguientes figuras se muestra el código CDK FaaS.

```
const mem=128 // 256, 512
const memTicket=1769
const v_deploy='vdep02' //2022-11-21
```

Figura 1: Código para definir la cantidad de memoria configurada en los lambdas

```
const tableFlights = dynamodb.Table.fromTableArn(this, `${v_deploy}-
TableFlightsFaaS`, 'arn:aws:dynamodb:us-east-
1:612828869890:table/SabFaasAwsBuildStack-vdep01TableFlightsA9A384D7-
1SKYWG1AQGSPC');
    const tableBooking = dynamodb.Table.fromTableArn(this,
`${v_deploy}-TableBookingFaaS`, 'arn:aws:dynamodb:us-east-
1:612828869890:table/SabFaasAwsBuildStack-vdep01TableBooking36C4AF35-
69JM3QWVCIQF');


    const tableFlightsCaaS = dynamodb.Table.fromTableArn(this,
`${v_deploy}-TableFlightsCaaS`, 'arn:aws:dynamodb:us-east-
1:612828869890:table/SabCaasAwsBuildStack-vdep01TableFlightsA9A384D7-
12HP76U3H5XGA');
    const tableBookingCaaS = dynamodb.Table.fromTableArn(this,
`${v_deploy}-TableBookingCaaS`, 'arn:aws:dynamodb:us-east-
1:612828869890:table/SabCaasAwsBuildStack-vdep01TableBooking36C4AF35-
GR9Q8R8HS9MI');
```

Figura 2: Código para importar la tabla de dynamodb que se ha creado manualmente

```
// 3.1.Funciones
    const fnLambdaFlightsReadAll = new lambda.Function(this,
`${v_deploy}-sab-FlightsReadAll-Fn`, {
      runtime: lambda.Runtime.NODEJS_16_X,
      handler: 'index.handler',
      memorySize: mem,
      timeout: cdk.Duration.minutes(14),
      code: lambda.Code.fromAsset(path.join(__dirname,
'/../../2.SAB_api-docker-zip-files/1.api-flights-readall.zip')),
    });

const fnLambdaFlightsReadOne = new lambda.Function(this, `${v_deploy}-
sab-FlightsReadOne-Fn`, {
      runtime: lambda.Runtime.NODEJS_16_X,
      handler: 'index.handler',
      memorySize: mem,
      timeout: cdk.Duration.minutes(10),
      code: lambda.Code.fromAsset(path.join(__dirname,
'/../../2.SAB_api-docker-zip-files/2.api-flights-readone.zip')),
    });
```

```
    const fnLambdaFlightsSpare = new lambda.Function(this,
`${v_deploy}-sab-FlightsSpare-Fn`, {
      runtime: lambda.Runtime.NODEJS_16_X,
      handler: 'index.handler',
      memorySize: mem,
      timeout: cdk.Duration.minutes(10),
      code: lambda.Code.fromAsset(path.join(__dirname,
'/../../2.SAB_api-docker-zip-files/3.api-flights-spare.zip')),});
    const fnLambdaTicketCreateOne = new lambda.Function(this,
`${v_deploy}-sab-TicketCreateOne-Fn`, {
      runtime: lambda.Runtime.NODEJS_16_X,
      handler: 'index.handler',
      memorySize: memTicket,
      timeout: cdk.Duration.minutes(14),
      code: lambda.Code.fromAsset(path.join(__dirname,
'/../../2.SAB_api-docker-zip-files/4.api-ticket-createone.zip')),});
    const fnLambdaBookingReadAll = new lambda.Function(this,
`${v_deploy}-sab-BookingReadAll-Fn`, {
      runtime: lambda.Runtime.NODEJS_16_X,
      handler: 'index.handler',
      memorySize: mem,
      timeout: cdk.Duration.minutes(14),
      code: lambda.Code.fromAsset(path.join(__dirname,
'/../../2.SAB_api-docker-zip-files/5.api-booking-readall.zip')),});
    const fnLambdaBookingWrite = new lambda.Function(this,
`${v_deploy}-sab-BookingWrite-Fn`, {
      runtime: lambda.Runtime.NODEJS_16_X,
      handler: 'index.handler',
      memorySize: mem,
      timeout: cdk.Duration.minutes(10),
      code: lambda.Code.fromAsset(path.join(__dirname,
'/../../2.SAB_api-docker-zip-files/6.api-booking-write.zip')),});
    const fnLambdaBookingSpare = new lambda.Function(this,
`${v_deploy}-sab-BookingSpare-Fn`, {
      runtime: lambda.Runtime.NODEJS_16_X,
      handler: 'index.handler',
      memorySize: mem,
      timeout: cdk.Duration.minutes(10),
      code: lambda.Code.fromAsset(path.join(__dirname,
'/../../2.SAB_api-docker-zip-files/7.api-booking-spare.zip')),});
```

Figura 3: Código para crear las funciones lambda y subir el código empaquetado en un zip

```
//3.2.Permisos
    tableFlights.grantReadData(fnLambdaFlightsReadAll);
    tableFlights.grantReadData(fnLambdaFlightsReadOne);
    tableFlights.grantFullAccess(fnLambdaFlightsSpare);
    tableFlightsCaaS.grantFullAccess(fnLambdaFlightsSpare);
    tableBooking.grantReadData(fnLambdaBookingReadAll);
    tableBooking.grantReadWriteData(fnLambdaBookingWrite);
    tableBooking.grantFullAccess(fnLambdaBookingSpare);
    tableBookingCaaS.grantFullAccess(fnLambdaBookingSpare);
```

Figura 4: Código para asignar permisos de lectura y escritura

```
//create the API gateway with one method and path
    const api=new apigw.RestApi(this,"gw-faas-sab",{
      cloudWatchRole: true,
      deployOptions: {
        loggingLevel: apigw.MethodLoggingLevel.INFO,
        dataTraceEnabled: true
      },
    });

    new cdk.CfnOutput(this,"HTTP API URL",{
      value: api.url ?? "Something went wrong with the deploy",
    });
```

Figura 5: Código para crear el api gateway

```
apiFlightsReadAll.addMethod('GET',new
apigw.LambdaIntegration(fnLambdaFlightsReadAll));
    apiFlightsReadOne.addMethod('ANY',new
apigw.LambdaIntegration(fnLambdaFlightsReadOne));
    apiFlightsReadOne.addProxy({defaultIntegration: new
apigw.LambdaIntegration(fnLambdaFlightsReadOne),anyMethod: true });
    apiFlightsSpare.addMethod('ANY',new
apigw.LambdaIntegration(fnLambdaFlightsSpare));
    apiFlightsSpare.addProxy({defaultIntegration: new
apigw.LambdaIntegration(fnLambdaFlightsSpare),anyMethod: true });
    apiTicketCreateOne.addMethod('ANY',new
apigw.LambdaIntegration(fnLambdaTicketCreateOne));
    apiTicketCreateOne.addProxy({defaultIntegration: new
apigw.LambdaIntegration(fnLambdaTicketCreateOne),anyMethod: true });
    apiBookingReadAll.addMethod('GET',new
apigw.LambdaIntegration(fnLambdaBookingReadAll));
    apiBookingWrite.addMethod('POST',new
apigw.LambdaIntegration(fnLambdaBookingWrite));
    apiBookingSpare.addMethod('ANY',new
apigw.LambdaIntegration(fnLambdaBookingSpare));
    apiBookingSpare.addProxy({defaultIntegration: new
apigw.LambdaIntegration(fnLambdaBookingSpare),anyMethod: true });
```

Figura 6: Código para crear los métodos e integrarlos con las funciones lambda

A.2. Código CaaS

Código de creación de recursos generales usados por las APIs

En las siguientes figuras se muestra el código CDK CaaS.

```
const v_deploy='vdep02'
const mem=512
const memTicket=2048 //1024
const cpu=256
const cpuTicket=1024 //512
```

Figura 7: Código para definir la cantidad de memoria y cpu configurado en los fargate

```
const tableFlights = dynamodb.Table.fromTableArn(this, `${v_deploy}-TableFlights`, 'arn:aws:dynamodb:us-east-1:612828869890:table/SabCaasAwsBuildStack-vdep01TableFlightsA9A384D7-12HP76U3H5XGA');
const tableBooking = dynamodb.Table.fromTableArn(this, `${v_deploy}-TableBooking`, 'arn:aws:dynamodb:us-east-1:612828869890:table/SabCaasAwsBuildStack-vdep01TableBooking36C4AF35-GR9Q8R8HS9MI');
```

Figura 8: Código para importar la tabla de dynamodb que se ha creado en la consola de AWS

```
const vpc = new ec2.Vpc(this, 'caasVPC', {
          maxAzs: 2,
          natGateways: 0,
          gatewayEndpoints: {
              S3: { service: ec2.GatewayVpcEndpointAwsService.S3 },
              DYNAMODB: { service: ec2.GatewayVpcEndpointAwsService.DYNAMODB }
          }
        })

        vpc.addInterfaceEndpoint('EcrDockerEndpoint', {
          service: ec2.InterfaceVpcEndpointAwsService.ECR_DOCKER
        })
        vpc.addInterfaceEndpoint('EcrApiEndpoint', {
            service: ec2.InterfaceVpcEndpointAwsService.ECR
        })

        vpc.addInterfaceEndpoint('CloudWatch', {
          service: ec2.InterfaceVpcEndpointAwsService.CLOUDWATCH
        })

        vpc.addInterfaceEndpoint('CloudWatchLogs', {
          service: ec2.InterfaceVpcEndpointAwsService.CLOUDWATCH_LOGS
        })
```

Figura 9: Código para crear la VPC y sus endpoints para S3, dynamoDB, ECR y cloudwatch

```
const cluster = new ecs.Cluster(this, 'Cluster', {
            vpc: vpc,
            });
```

Figura 10: Código para un ecs cluster para ambos servicios principal y secundario

```
const nlbGeneral = new elbv2.NetworkLoadBalancer(this, 'LBGeneral', {
            vpc,
            crossZoneEnabled: true,
            internetFacing: true,
            });
```

Figura 11: Código para crear un nlb general para ser usado por todas las APIs

Código del API fligts read all

En las siguientes figuras se muestra el código para la creación de recursos del API flights read all.

```
const flightsReadAllServiceTaskDefinition = new
ecs.FargateTaskDefinition(this, `${v_deploy}-ECS-flightsReadAll-
SrvcTaskDef`, {
      memoryLimitMiB: mem,
      cpu: cpu,
    });
```

Figura 12: Código para crear una definición de tarea configurando la memoria y CPU

```
const flightsReadAllServiceSecGrp = new ec2.SecurityGroup(this,
"flightsReadAllServiceSecurityGroup", {
      allowAllOutbound: true,
      securityGroupName: 'flightsReadAllServiceSecurityGroup',
      vpc: vpc
    });
    flightsReadAllServiceSecGrp.connections.allowFromAnyIpv4(ec2.Port.
tcp(3010));
```

Figura 13: Código para crear un security group

```
const caasflightsReadAllServiceRepov1 =
ecr.Repository.fromRepositoryName(this,
      "caasSabflightsReadAllContainer",
      "caas-sab-flights-readall-container",
    );
    const
flightsReadAllServiceContainer=flightsReadAllServiceTaskDefinition.ad
dContainer('ECR-flightsRA', {
      image:
ecs.ContainerImage.fromEcrRepository(caasflightsReadAllServiceRepov1)
,
      logging: ecs.LogDrivers.awsLogs({ streamPrefix: 'ECR-
EventsFlightsReadAll' }),
    });
```

Figura 14: Código para configurar la subida de la imagen del container hacia el ecr creado por consola

```
flightsReadAllServiceContainer.addPortMappings({
      containerPort: 3010
    });
```

Figura 15: Código para crear puerto de viiiea doneviii del servicio ejecutador por el servicio de ecs

```
const flightsReadAllEcsService = new ecs.FargateService(this,
`${v_deploy}-flightsReadAllSrv`, {
      cluster: cluster,
      taskDefinition: flightsReadAllServiceTaskDefinition,
      desiredCount: 1,
      assignPublicIp: false,
      securityGroups: [flightsReadAllServiceSecGrp],
    });

    const scalableTargetFlightsRA =
flightsReadAllEcsService.autoScaleTaskCount({
      minCapacity: 1,
      maxCapacity: 3,
    });
```

Figura 16: Código para asociar una instancia de servicio fargate con el ecs y configurando el autoescalamiento

```
tableFlights.grantReadWriteData(flightsReadAllEcsService.taskDefinitio
n.taskRole);
```

Figura 17: Código para brindar permisos de lectura y escritura del servicio fargate sobre el recurso dynamodb importado

```
const listenerflightsReadAll =
nlbGeneral.addListener('flightsReadAllListener', {
      port: 3010,
    });

listenerflightsReadAll.addTargets('flightsReadAllTarget', {
      port: 3010,
      targets: [flightsReadAllEcsService],

    });
```

Figura 18: Código para crear un listener y un puerto target al balanceador para el servicio principal

Código del API flights read one

En las siguientes figuras se muestra el código para la creación de recursos del API flights read one.

```
const flightsReadOneServiceTaskDefinition = new
ecs.FargateTaskDefinition(this, `${v_deploy}-ECS-flightsReadOne-
SrvcTaskDef`, {
      memoryLimitMiB: mem,
      cpu: cpu,
    });
```

Figura 19: Código para crear una definición de tarea configurando la memoria y CPU

```
const flightsReadOneServiceSecGrp = new ec2.SecurityGroup(this,
"flightsReadOneServiceSecurityGroup", {
      allowAllOutbound: true,
      securityGroupName: 'flightsReadOneServiceSecurityGroup',
      vpc: vpc
    });
flightsReadOneServiceSecGrp.connections.allowFromAnyIpv4(ec2.Port.tcp(
3011));
```

Figura 20: Código para crear un security group

```
const caasflightsReadOneServiceRepov1 =
ecr.Repository.fromRepositoryName(this,
      "caasSabflightsReadOneContainer",
      "caas-sab-flights-readone-container",
    );
    const
flightsReadOneServiceContainer=flightsReadOneServiceTaskDefinition.ad
dContainer('ECR-flightsRO', {
      image:
ecs.ContainerImage.fromEcrRepository(caasflightsReadOneServiceRepov1)
,
      logging: ecs.LogDrivers.awsLogs({ streamPrefix: 'ECR-
EventsFlightsReadOne' }),


    });
```

Figura 21: Código para configurar la subida de la imagen del container hacia el ecr creado por consola

```
flightsReadOneServiceContainer.addPortMappings({
      containerPort: 3011
    });
```

Figura 22: Código para crear puerto de recepcion del servicio ejecutador por el servicio de ecs

```
const flightsReadOneEcsService = new ecs.FargateService(this,
`${v_deploy}-flightsReadOneSrv`, {
      cluster: cluster,
      taskDefinition: flightsReadOneServiceTaskDefinition,
      desiredCount: 1,
      assignPublicIp: false,
      securityGroups: [flightsReadOneServiceSecGrp],
    });

    const scalableTargetFlightsRO =
flightsReadOneEcsService.autoScaleTaskCount({
      minCapacity: 1,
      maxCapacity: 3,
    });
```

Figura 23: Código para asociar una instancia de servicio fargate con el ecs y configurando el autoescalamiento

```
tableFlights.grantReadWriteData(flightsReadOneEcsService.taskDefinitio
n.taskRole);
```

Figura 24: Código para brindar permisos de lectura y escritura del servicio fargate sobre el recurso dynamodb importado

```
const listenerflightsReadOne =
nlbGeneral.addListener('flightsReadOneListener', {
      port: 3011,
    });

listenerflightsReadOne.addTargets('flightsReadOneTarget', {
      port: 3011,
      targets: [flightsReadOneEcsService],

    });
```

Figura 25: Código para crear un listener y un puerto target al balanceador para el servicio principal

Código del API ticket create one

En las siguientes figuras se muestra el código para la creación de recursos del API ticket create one.

```
const ticketCreateOneServiceTaskDefinition = new
ecs.FargateTaskDefinition(this, `${v_deploy}-ECS-ticketCreateOne-
SrvcTaskDef`, {
      memoryLimitMiB: memTicket,
      cpu: cpuTicket,
    });
```

Figura 26: Código para crear una definición de tarea configurando la memoria y CPU

```
const ticketCreateOneServiceSecGrp = new ec2.SecurityGroup(this,
"ticketCreateOneServiceSecurityGroup", {
      allowAllOutbound: true,
      securityGroupName: 'ticketCreateOneServiceSecurityGroup',
      vpc: vpc
    });
ticketCreateOneServiceSecGrp.connections.allowFromAnyIpv4(ec2.Port.tcp
(8010));
```

Figura 27: Código para crear un security group

```
const caasticketCreateOneServiceRepo =
ecr.Repository.fromRepositoryName(this,
       "caasSabticketCreateOneContainer",
       "caas-sab-ticket-createone-container",
     );
    const
ticketCreateOneServiceContainer=ticketCreateOneServiceTaskDefinition.
addContainer('ECR-ticketWO', {
      image:
ecs.ContainerImage.fromEcrRepository(caasticketCreateOneServiceRepo),
      logging: ecs.LogDrivers.awsLogs({ streamPrefix: 'ECR-
EventsTicketCreateOne' }),
    });
```

Figura 28: Código para configurar la subida de la imagen del container hacia el ecr creado por consola

```
ticketCreateOneServiceContainer.addPortMappings({
      containerPort: 8010
    });
```

Figura 29: Código para crear puerto de xiiecepción del servicio ejecutador por el servicio de ecs

```
const ticketCreateOneEcsService = new ecs.FargateService(this,
`${v_deploy}-ticketCreateOneEcsSrv`, {
      cluster: cluster,
      taskDefinition: ticketCreateOneServiceTaskDefinition,
      desiredCount: 2,
      assignPublicIp: false,
      securityGroups: [ticketCreateOneServiceSecGrp],
    });

    const scalableTargetticketCO =
ticketCreateOneEcsService.autoScaleTaskCount({
      minCapacity: 1,
      maxCapacity: 20,
    });
```

```
scalableTargetticketCO.scaleOnCpuUtilization('CpuScaling', {
      targetUtilizationPercent: 75,
      scaleInCooldown: cdk.Duration.seconds(10),
      scaleOutCooldown: cdk.Duration.seconds(10),
    });

    scalableTargetticketCO.scaleOnMemoryUtilization('MemoryScaling', {
      targetUtilizationPercent: 50,
    });
```

Figura 30: Código para asociar una instancia de servicio fargate con el ecs y configurando el autoescalamiento

```
const listenerNLBticketCreateOne =
nlbGeneral.addListener('ListenerNLBticketCreateOne', {
      port: 8010,
    });

listenerNLBticketCreateOne.addTargets('TargetNLBticketCreateOne', {
      port: 8010,
      targets: [ticketCreateOneEcsService],
    });
```

Figura 31: Código para crear un listener y un puerto target al balanceador para el servicio principal

Código del API booking read all

En las siguientes figuras se muestran el código para la creación de recursos del API booking read all.

```
const bookingReadAllServiceTaskDefinition = new
ecs.FargateTaskDefinition(this, `${v_deploy}-ECS-bookingReadAll-
SrvcTaskDef`, {
      memoryLimitMiB: mem,
      cpu: cpu,
    });
```

Figura 32: Código para crear una definición de tarea configurando la memoria y CPU

```
const bookingReadAllServiceSecGrp = new ec2.SecurityGroup(this,
"bookingReadAllServiceSecurityGroup", {
      allowAllOutbound: true,
      securityGroupName: 'bookingReadAllServiceSecurityGroup',
      vpc: vpc
    });
    bookingReadAllServiceSecGrp.connections.allowFromAnyIpv4(ec2.Port.
tcp(3050));
```

Figura 33: Código para crear un security group

```
const caasbookingReadAllServiceRepo =
ecr.Repository.fromRepositoryName(this,
       "caasSabbookingReadAllContainer",
       "caas-sab-booking-readall-container",
     );

    const
bookingReadAllServiceContainer=bookingReadAllServiceTaskDefinition.ad
dContainer('ECR-bookingRA', {
      image:
ecs.ContainerImage.fromEcrRepository(caasbookingReadAllServiceRepo),
      logging: ecs.LogDrivers.awsLogs({ streamPrefix: 'ECR-
EventsBookingReadAll' }),
    });
```

Figura 34: Código para configurar la subida de la imagen del container hacia el ecr creado por consola

```
bookingReadAllServiceContainer.addPortMappings({
      containerPort: 3050
    });
```

Figura 35: Código para crear puerto de recepcion del servicio ejecutador por el servicio de ecs

```
const bookingReadAllEcsService = new ecs.FargateService(this,
`${v_deploy}-bookingReadAllECSSrv`, {
      cluster: cluster,
      taskDefinition: bookingReadAllServiceTaskDefinition,
      desiredCount: 1,
      assignPublicIp: false,
      securityGroups: [bookingReadAllServiceSecGrp],
    });

    const scalableTargetBookingRA =
bookingReadAllEcsService.autoScaleTaskCount({
      minCapacity: 1,
      maxCapacity: 3,
    });
```

Figura 36: Código para asociar una instancia de servicio fargate con el ecs y configurando el autoescalamiento

```
tableBooking.grantReadData(bookingReadAllEcsService.taskDefinition.tas
kRole);
```

Figura 37: Código para brindar permisos de lectura y escritura del servicio fargate sobre el recurso dynamodb importado

```
const listenerNLBbookingReadAll =
nlbGeneral.addListener('ListenerNLBbookingReadAll', {
      port: 3050,
    });

listenerNLBbookingReadAll.addTargets('TargetNLBbookingReadAll', {
      port: 3050,
      targets: [bookingReadAllEcsService],
    });
```

Figura 38: Código para crear un listener y un puerto target al balanceador para el servicio principal

Código del API booking write

En las siguientes figuras se muestran el código para la creación de recursos del API booking write.

```
const bookingWriteServiceTaskDefinition = new
ecs.FargateTaskDefinition(this, `${v_deploy}-ECS-bookingWrite-
SrvcTaskDef`, {
      memoryLimitMiB: mem,
      cpu: cpu,
    });
```

Figura 39: Código para crear una definición de tarea configurando la memoria y CPU

```
const bookingWriteServiceSecGrp = new ec2.SecurityGroup(this,
"bookingWriteServiceSecurityGroup", {
      allowAllOutbound: true,
      securityGroupName: 'bookingWriteServiceSecurityGroup',
      vpc: vpc
    });

    bookingWriteServiceSecGrp.connections.allowFromAnyIpv4(ec2.Port.tc
p(3051));
```

Figura 40: Código para crear un security group

```
const caasbookingWriteServiceRepo =
ecr.Repository.fromRepositoryName(this,
       "caasSabbookingWriteContainer",
       "caas-sab-booking-write-container",
     );

    const
bookingWriteServiceContainer=bookingWriteServiceTaskDefinition.addCon
tainer('ECR-bookingW', {
      image:
ecs.ContainerImage.fromEcrRepository(caasbookingWriteServiceRepo),
      logging: ecs.LogDrivers.awsLogs({ streamPrefix: 'ECR-
EventsBookingWrite' }),
    });
```

Figura 41: Código para configurar la subida de la imagen del container hacia el ecr creado por consola

```
bookingWriteServiceContainer.addPortMappings({
      containerPort: 3051
    });
```

Figura 42: Código para crear puerto de recepcion del servicio ejecutador por el servicio de ecs

```
const bookingWriteEcsService = new ecs.FargateService(this,
`${v_deploy}-bookingWriteEcsSrv`, {
      cluster: cluster,
      taskDefinition: bookingWriteServiceTaskDefinition,
      desiredCount: 1,
      assignPublicIp: false,
      securityGroups: [bookingWriteServiceSecGrp],
    });

    const scalableTargetBookingW =
bookingWriteEcsService.autoScaleTaskCount({
      minCapacity: 1,
      maxCapacity: 3,
    });
```

Figura 43: Código para asociar una instancia de servicio fargate con el ecs y configurando el autoescalamiento

```
tableBooking.grantReadWriteData(bookingWriteEcsService.taskDefinition.
taskRole);
```

Figura 44: Código para brindar permisos de lectura y escritura del servicio fargate sobre el recurso dynamodb importado

```
const listenerNLBbookingWrite =
nlbGeneral.addListener('ListenerNLBbookingWrite', {
      port: 3051,
    });

listenerNLBbookingWrite.addTargets('TargetNLBbookingWrite', {
      port: 3051,
      targets: [bookingWriteEcsService],
    });
```

Figura 45: Código para crear un listener y un puerto target al balanceador para el servicio principal

Código del api Gateway

```
const api=new apigw.RestApi(this,"gw-caas-sab",{
      cloudWatchRole: true,
      deployOptions: {
        loggingLevel: apigw.MethodLoggingLevel.INFO,
        dataTraceEnabled: true
      },
    });
```

Figura 46: Código para crear el recurso api Gateway habilitando el registro de logs

```
const linkGeneral = new apigw.VpcLink(this, 'linkGeneral', {
      targets: [nlbGeneral],
    });
```

Figura 47: Código para para crear el vpclink para realizar la integración privada entre el apigateway y los servicios en ejecución en el ecs

```
const sabGWFlightsReadAll=api.root.addResource('api-flights-readall');
const sabGWFlightsReadOne=api.root.addResource('api-flights-readone');
const sabGWTicketCreateone=api.root.addResource('api-ticket-
createone');
const sabGWBookingReadAll=api.root.addResource('api-booking-readall');
const sabGWBookingWrite=api.root.addResource('api-booking-write');
```

Figura 48: Código para agregar al apigateway los recursos para el consumo de los 5 APIs

```
sabGWFlightsReadAll.addMethod(
      'ANY',
      new apigw.Integration({
        type: apigw.IntegrationType.HTTP_PROXY,
        integrationHttpMethod:'ANY',
        uri:'http://sabca-lbgen-ihk96w3yvvbz-5f3a0b8816acda45.elb.us-east-
1.amazonaws.com:3010/api-flights-readall',
        options: {
          connectionType: apigw.ConnectionType.VPC_LINK,
          vpcLink: linkGeneral,
        },
      })
    );
```

Figura 49: Código para crear un método any para el API flights read all y se le asocia el vpclink

```
sabGWFlightsReadOne.addMethod(
      'ANY',
      new apigw.Integration({
        type: apigw.IntegrationType.HTTP_PROXY,
        integrationHttpMethod:'ANY',
        uri:'http://sabca-lbgen-ihk96w3yvvbz-5f3a0b8816acda45.elb.us-
east-1.amazonaws.com:3011/api-flights-readone',
        options: {
          connectionType: apigw.ConnectionType.VPC_LINK,
          vpcLink: linkGeneral,
        },
      }),
    );
```

```
const itemFlightsReadOne = sabGWFlightsReadOne.addResource('{item}');
    itemFlightsReadOne.addMethod(
      'ANY',
      new apigw.Integration({
        type: apigw.IntegrationType.HTTP_PROXY,
        integrationHttpMethod:'ANY',
        uri:'http://sabca-lbgen-ihk96w3yvvbz-5f3a0b8816acda45.elb.us-
east-1.amazonaws.com:3011/api-flights-readone/{item}',
        options: {
          connectionType: apigw.ConnectionType.VPC_LINK,
          vpcLink: linkGeneral,
          requestParameters:
{'integration.request.path.item':'method.request.path.item'},
        },
      }),
      {requestParameters: {'method.request.path.item': true}}
    );
```

Figura 50: Código para crear un método any para el API flights read one y se le asocia el vpclink

```
sabGWTicketCreateone.addMethod(
      'ANY',
      new apigw.Integration({
        type: apigw.IntegrationType.HTTP_PROXY,
        integrationHttpMethod:'ANY',
        uri:'http://sabca-lbgen-ihk96w3yvvbz-5f3a0b8816acda45.elb.us-
east-1.amazonaws.com:8010/api-ticket-createone',
        options: {
          connectionType: apigw.ConnectionType.VPC_LINK,
          vpcLink: linkGeneral,
        },
      })
    );
```

```
const itemTicketCreateone =
sabGWTicketCreateone.addResource('{item}');
    itemTicketCreateone.addMethod(
      'ANY',
      new apigw.Integration({
        type: apigw.IntegrationType.HTTP_PROXY,
        integrationHttpMethod:'ANY',
        uri:'http://sabca-lbgen-ihk96w3yvvbz-5f3a0b8816acda45.elb.us-
east-1.amazonaws.com:8010/api-ticket-createone/{item}',
        options: {
          connectionType: apigw.ConnectionType.VPC_LINK,
          vpcLink: linkGeneral,
          requestParameters:
{'integration.request.path.item':'method.request.path.item'},
        },
      }),
      {requestParameters: {'method.request.path.item': true}}
    );
```

Figura 51: Código para crear un método any para el API ticket create one y se le asocia el vpclink

```
sabGWBookingReadAll.addMethod(
      'ANY',
      new apigw.Integration({
        type: apigw.IntegrationType.HTTP_PROXY,
        integrationHttpMethod:'ANY',
        uri:'http://sabca-lbgen-ihk96w3yvvbz-5f3a0b8816acda45.elb.us-
east-1.amazonaws.com:3050/api-booking-readall',
        options: {
          connectionType: apigw.ConnectionType.VPC_LINK,
          vpcLink: linkGeneral,
        },
      })
    );
```

Figura 52: Código para crear un método any para el API booking read all y se le asocia el vpclink

```
sabGWBookingWrite.addMethod(
      'ANY',
      new apigw.Integration({
        type: apigw.IntegrationType.HTTP_PROXY,
        integrationHttpMethod:'ANY',
        uri:'http://sabca-lbgen-ihk96w3yvvbz-5f3a0b8816acda45.elb.us-
east-1.amazonaws.com:3051/api-booking-write',
        options: {
          connectionType: apigw.ConnectionType.VPC_LINK,
          vpcLink: linkGeneral,
        },
      })
    );
```

Figura 53: Código para crear un método any para el API booking write y se le asocia el vpclink

B. Uso de servicios de Cloudwatch de AWS

En las gráficas se muestra las interfaces del servicio de Cloudwatch de AWS

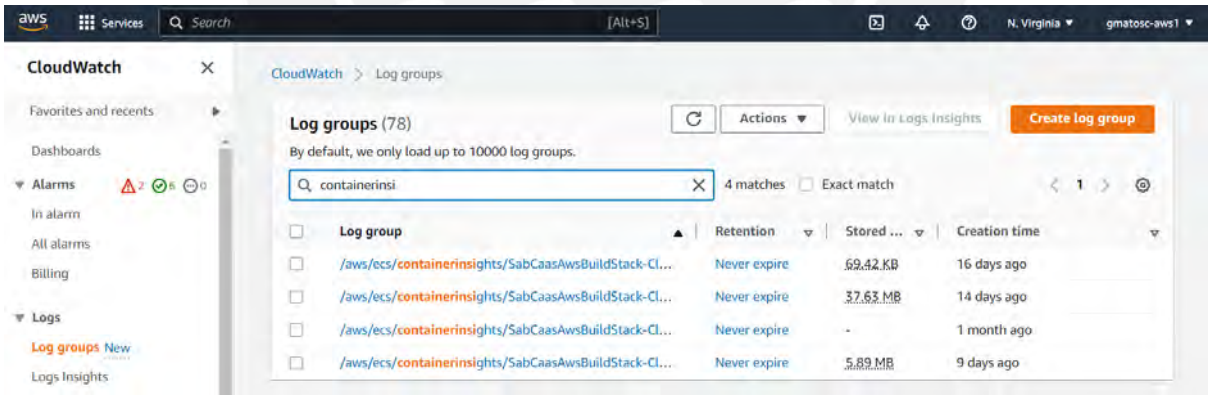

Figura 54: Log groups que contienen los logs streams de métricas de los Fargate

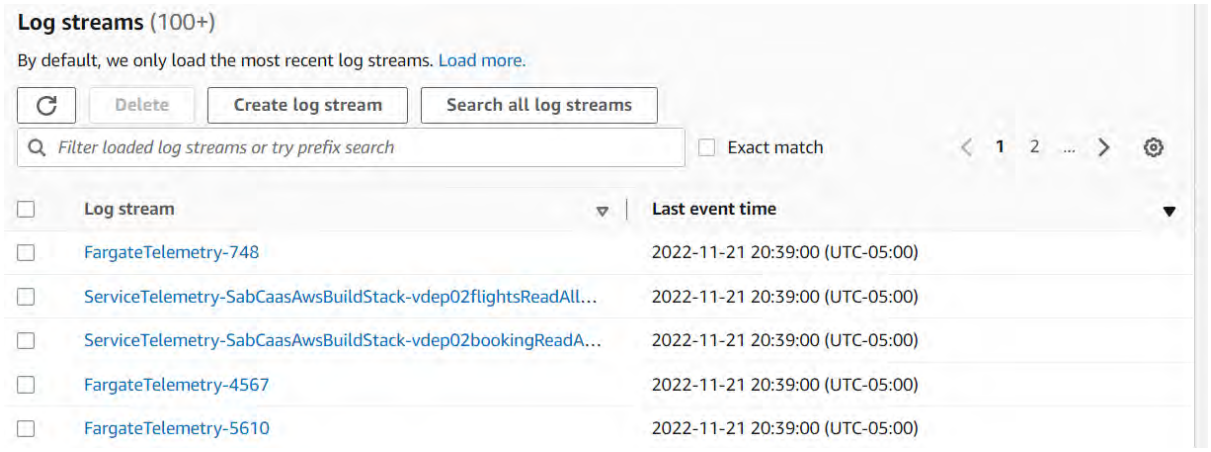

Figura 55: Log streams que contienen los registros de métricas de los Fargate

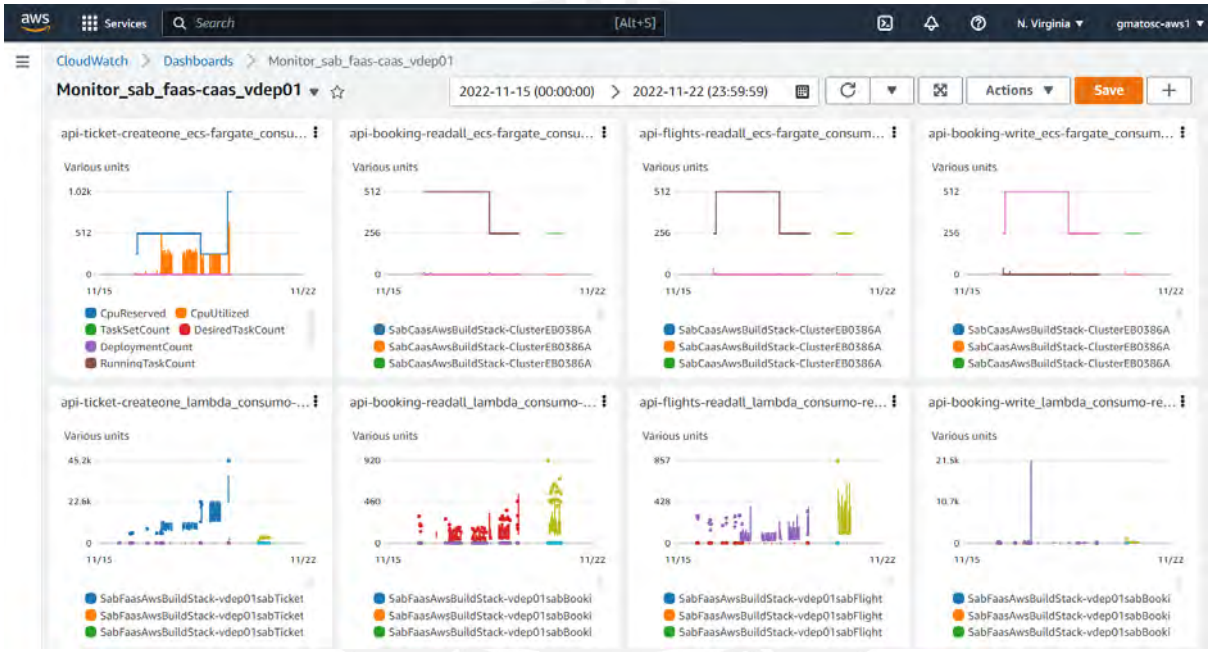

Figura 56: Servicio de dashboard de AWS

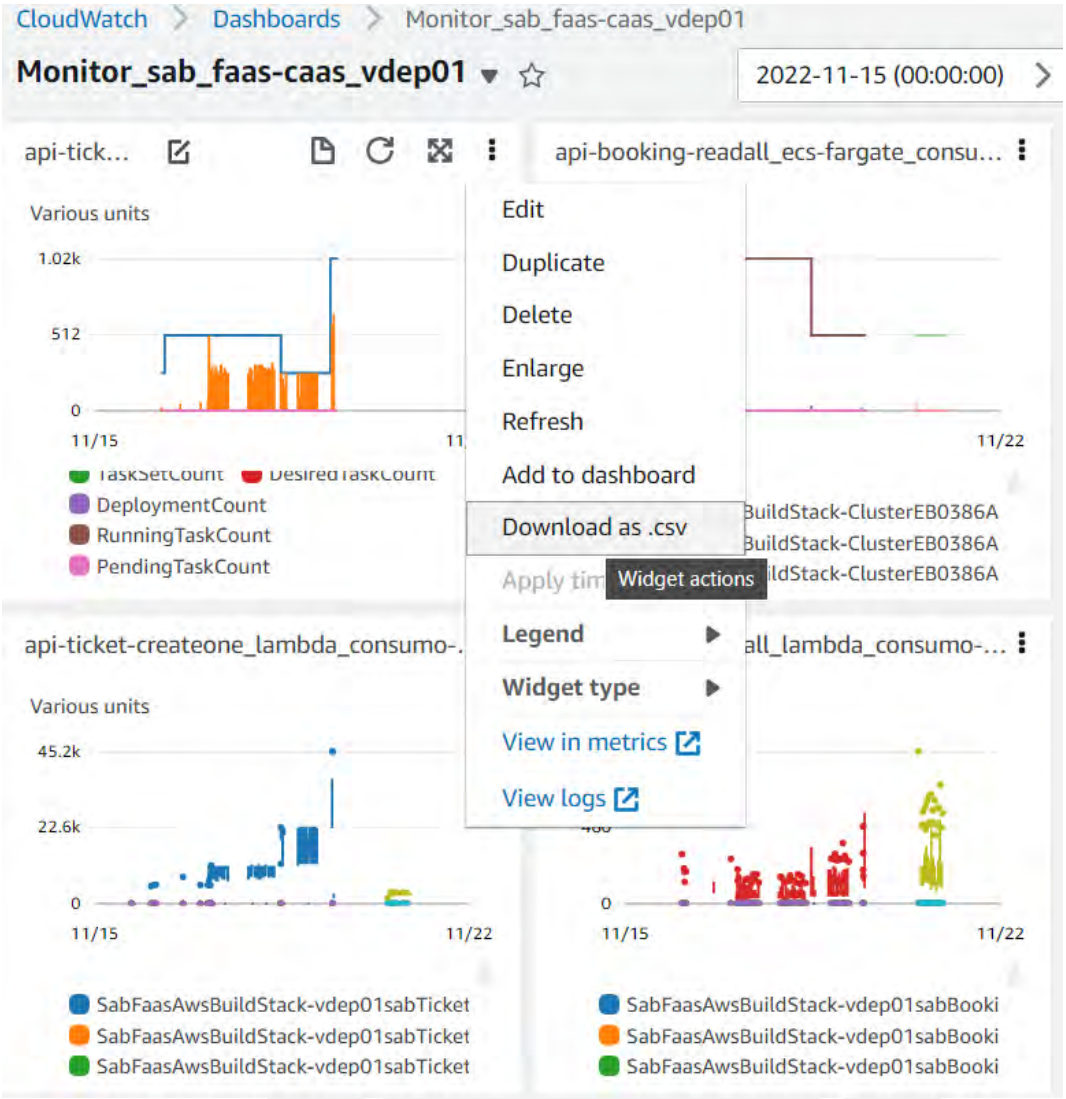

Figura 57: Plugin de métricas del dashboard que permiten la descarga como CSVs