ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

Creación de un microservicio en la nube para la resolución de problemas de optimización matemática que permitan autoconfigurar sistemas de almacenamiento distribuido

PROYECTO INTEGRADOR

Previo la obtención del Título de:

Ingeniero en Computación

Presentado por:
Jorge Luis Cedeño Arteaga
Oscar Daniel Moreno Abad

GUAYAQUIL - ECUADOR
Año: 2018

DEDICATORIA

El presente proyecto de investigación va dedicado a mi familia, pero principalmente a mis padres Luis Cedeño y Docty Arteaga que me han brindado siempre su apoyo, buenos ejemplos y amor. A mi hermana: Docty Cedeño que fue un soporte durante todos estos años de estudio y a mi hermano Abraham ya que no he estado presente en su vida como me hubiera gustado estar. A mis tíos que siempre serán como unos segundos padres por todo el cariño que me brindan, y a Ricardo Vélez, que alguna vez fue quien me dijo "estudia ingeniería".

Jorge Cedeño

El presente proyecto lo dedico a mis padres por haberme acompañado durante todo mi recorrido estudiantil.

A mi tía por ser una segunda madre para mí.

Oscar Moreno

AGRADECIMIENTOS

A la ESPOL, a los docentes por preocuparse siempre de brindarnos la mejor educación, en especial a la PhD. Cristina Abad por su gran ayuda y colaboración en cada momento de consulta en el presente proyecto, y al PhD. Boris Vintimilla por la retroalimentación durante la escritura del presente documento, y a mi compañero de proyecto por siempre mantener una buena relación de trabajo

Jorge Cedeño

Mis más sinceros agradecimientos a mis padres por todo su apoyo para formar mi educación.

A la PhD Cristina Abad por brindarnos las herramientas necesarias para desarrollar nuestro trabajo y su valiosa guía.

Al PhD Boris Vintimilla por la retroalimentación en la elaboración del presente documento.

A mi compañero de proyecto por su apoyo.

Oscar Moreno

DECLARACIÓN EXPRESA

“Los derechos de titularidad y explotación, me(nos) corresponde conforme al reglamento de propiedad intelectual de la institución; *Jorge Cedeño, Oscar Moreno* y doy(damos) mi(nuestro) consentimiento para que la ESPOL realice la comunicación pública de la obra por cualquier medio con el fin de promover la consulta, difusión y uso público de la producción intelectual"

Jorge Cedeño

Oscar Moreno

RESUMEN

Los proveedores de servicios en la nube y otras empresas usan cachés para reducir latencia, brindar un servicio más rápido y generar mayor satisfacción de los usuarios. Estas cachés son softwares simples y están diseñadas para atender requerimientos a gran velocidad con mínima latencia, y usualmente, más allá de sus políticas desalojo, como LRU, no cuentan con inteligencia adicional para adaptarse a los cambios en las cargas de trabajo o los requerimientos de las aplicaciones. Este trabajo busca ofrecer un servicio que calcule automáticamente la repartición de la memoria caché entre distintas aplicaciones, de tal manera que se maximice el rendimiento del sistema. Para implementar la solución propuesta, hemos decidido resolverlo con tres módulos: módulo 1, Función de AWS Lambda que resuelve el problema de optimización usando un algoritmo de hill climbing, módulo 2, función de AWS Lambda que resuelve el problema de optimización usando un algoritmo evolutivo propuesto en [2] y módulo 3: Aquí se integraron los dos módulos previos y se realizaron pruebas de funcionalidad y comparación para ambos algoritmos. Dentro de los resultados se obtuvo un análisis comparativo y de tiempo entre ambos solvers. Dónde el algoritmo de hill climbing obtenía mejores resultados para una repartición de memoria de 6 ó más aplicaciones y para menos aplicaciones el algoritmo evolutivo resultó la mejor opción. A partir de esos resultados se implementó el microservicio, de tal manera que cuando se lo invoque use uno de los 2 algoritmos según el número de aplicaciones.

Palabras Clave: AWS Lambda, FaaS, Hill Climbing, Caché, Optimización.

ABSTRACT

Cloud service providers and other companies use caches to reduce latency, provide faster service and generate greater user satisfaction. These caches are simple software and are designed to meet high-speed requirements with minimum latency, beyond their eviction policies, such as LRU; frequently, these caches do not have any additional intelligence to adapt to changes in workloads or application requirements. This project seeks to offer a service that automatically configures the distribution of the cache between different applications, to maximize the system's performance. To implement the service solution, we have implemented three modules: module 1, AWS Lambda function that solves the optimization problem using a hill climbing algorithm, module 2, AWS Lambda function that solves the optimization problem using an evolutionary algorithm proposed in [2] and module 3, here the two previous modules were integrated, and tests of functionality and comparison were carried out for both algorithms. Within the results, a comparative and time analysis between both solvers was obtained. Where the hill climbing algorithm obtained better results for a distribution of memory of 6 or more applications and for less applications the evolutionary algorithm was the best option. From these results, the microservice was implemented, so that when it is invoked, use one of the 2 modules according to the number of applications.

Keywords: Aws Lambda, FaaS, Hill Climbing, Cache, Optimization

ÍNDICE GENERAL

ABREVIATURAS

ESPOL	Escuela Superior Politécnica del Litoral
FaaS	Function as a Service
MRC	Miss Rate Curves
AWS	Amazon Web Services
EC2	Elastic Cloud Computing
MySQL	My Structured Query Language
SQL	Structured Query Language
S3	Simple Storage Service
HTML	Hyper Text Markup Language
LRU	Least Recently Used
HTTP	Hyper Text Transfer Protocol
JSON	Java Script Object Notation
API	Application Programming Interface

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1

1. INTRODUCCIÓN

El presente proyecto es desarrollado como parte de la materia integradora de la ESPOL en el año 2018, donde el cliente del mismo es el grupo de investigación de big data de la ESPOL.

1.1 Descripción del problema

El concepto de la computación en la nube nace desde 1960, cuando Joseph Carl Robnett Licklider expresó su idea de “Red de Computadoras Intergalácticas” [8] y en 1961 John McCarthy dijo que “Algún día la computación podrá ser organizada como un servicio público” [4]. Pero no fue hasta finales de la década de los 90 y durante la primera década del 2000 donde los primeros grandes hitos del cloud Computing empiezan a tomar lugar. En 1999 Salesforce.com fue pionera en la entrega de aplicaciones empresariales a través de una página web simple. Posteriormente, Amazon lanza Amazon Web Services AWS en el 2002 y en el 2006 lanza el Elastic Compute Cloud (EC2), servicio web que permite a empresas e individuos alquilar computadoras en la nube. Entre el 2006 y el 2007, Google lanza Google Docs., con lo que se masificó el uso de los servicios en la nube.

La “Nube”, es un término que se introdujo por grandes corporaciones como Google y Amazon, en el 2006 cuando lanzaron sus nuevos servicios. Desde estos primeros hitos, el Cloud Computing está creciendo y siendo adoptado a gran velocidad en el sector empresarial [9]. Este crecimiento se debe principalmente al rápido desarrollo de la tecnología del procesamiento y el almacenamiento de datos, además del éxito del Internet [10].

Estos constantes avances conllevaron al desarrollo de tecnologías como Memcached y Redis, que se han convertido en parte importante de la infraestructura en la nube [2]. Estos dos productos son cachés de software, implementados con bases de datos de clave-valor con almacenamiento en memoria. Se las usa en arquitecturas en la nube para cachear información, para evitar, en lo posible tener que consultar a la base de datos.

Los proveedores de servicios en la nube usan cachés para reducir la latencia y brindar un servicio más rápido, lo cual puede generar una mayor satisfacción del usuario y por ende un incremento en los ingresos [2]. Pequeñas mejoras en el hit rate de la caché, tiene un impacto significativo en el rendimiento percibido por el usuario en servicios web modernos, porque leer datos de una base de datos basada en discos duros (como MySQL) es órdenes de magnitud más lento que leer en memoria caché [3].

Pero las cachés de software son simples y están diseñadas para atender requerimientos a gran velocidad con mínima latencia; más allá de sus políticas de desalojo, como LRU, no tienen inteligencia adicional lo que no les permite adaptarse a los cambios en las cargas de trabajo o los requerimientos de las aplicaciones [1].

Se usará un ejemplo para ilustrar el problema que se quiere solucionar: Imagine un sitio web el cual tiene que almacenar en caché las siguientes cargas de trabajo: Resultados SQL, páginas HTML dinámicas, perfiles y avatares de usuarios. Tomando en cuenta este escenario el reto es repartir de la manera más eficiente el recurso limitado de memoria. Esto se hace actualmente de manera estática, basándose en la experiencia humana empírica y el análisis de la carga de trabajo offline.

Lo anterior nos lleva a plantearnos la siguiente pregunta.

¿Cómo se puede repartir de manera dinámica, el recurso de memoria de una caché de software, entre distintas aplicaciones?

1.2 Justificación del problema

Muchos sistemas de almacenamiento distribuidos se beneficiarían de poder auto configurarse, por ejemplo, en cuanto a particionamiento de espacio entre aplicaciones. Actualmente esto se lo hace de manera estática [1], basándose en la experiencia humana empírica y el análisis de la carga de trabajo fuera de línea.

También se conoce que mientras menos interactivo sea un sitio, más probable será que los usuarios hagan clic y realicen otra actividad, por lo que se busca reducir latencia de

los sitios web. Por ejemplo, Amazon encontró que cada 100 ms de latencia les costó 1% en ventas, mientras que Google reporta que un extra de 0.5 segundos en el tiempo de generación de la página de búsqueda, disminuye el tráfico en 20%. Un corredor podría perder $ 4 millones en ingresos por milisegundo si su plataforma de negociación electrónica está 5 milisegundos detrás de la competencia [7].

1.3 Objetivos

1.3.1 Objetivo general

Crear un microservicio en la nube para la resolución de problemas de optimización matemática que permitan autoconfigurar sistemas de almacenamiento distribuido.

1.3.2 Objetivos específicos

- Analizar las posibles alternativas para resolver el problema de optimización.
- Establecer la plataforma en la que se va a implementar el microservicio.
- Implementar la solución en la plataforma seleccionada.
- Evaluar la solución implementada.

1.4 Marco teórico

1.4.1 Conceptos

Memcached [5] es una base de datos de clave-valor que guarda sus datos en memoria y no en almacenamiento persistente. Memcached es un producto de código abierto y libre, de alto rendimiento, pero destinado para agilizar las aplicaciones web dinámicas aliviando la carga de la base de datos.

Redis [6] es un almacén de estructura de datos en memoria, y también es de código abierto. Redis se utiliza como base de datos, caché y agente de mensajes. Admite estructuras de datos tales como cadenas, hashes, listas, conjuntos, conjuntos ordenados con consultas de rango, mapas de bits, hiperlogálogos e índices geoespaciales con consultas radiales. Redis tiene una replicación incorporada, secuencias de comandos Lua, desalojo LRU, transacciones y

diferentes niveles de persistencia en disco, y proporciona alta disponibilidad a través de Redis Sentinel y particiones automáticas con Redis Clúster.

1.4.2 Trabajos relacionados

En [2] se estudia el problema de la partición dinámica de memoria en cachés de nube y lo modelan como un problema de optimización. Proponen un modelo que se pueda usar para configurar dinámicamente las asignaciones de memoria en función de las cargas de trabajo observadas, de modo que se maximice la utilidad general del sistema. También consideran el efecto de tener backends con diferentes perfiles de rendimiento, así como el costo de volver a particionar la memoria. En [2] también se propone un solver como servicio que pueda ser utilizado por muchos clientes para calcular periódicamente su partición óptima.

En [3] demuestran que al usar una caché web se puede mejorar de manera significativa ajustando su comportamiento para ajustarse dinámicamente a los requisitos de diferentes aplicaciones. Se demuestra que el rendimiento de ciertas aplicaciones se puede mejorar significativamente mejorando la asignación de los bloques de memoria dentro de los servidores de Memcached, sin interferir con la ruta de datos de la memoria caché.

En [11] presentan una arquitectura de almacenamiento en caché llamada Moirai, definida por software que permite el control de las memorias cachés en un centro de datos de múltiples inquilinos, sin requerir ninguna entrada o sugerencias de inquilinos. Moirai puede ayudar a facilitar la administración de la infraestructura de caché distribuida y permite al proveedor alcanzar una serie de objetivos diferentes.

CAPÍTULO 2

2. METODOLOGÍA

Para resolver el problema de la repartición óptima de la memoria entre distintas cargas de trabajo o aplicaciones, se desea optimizar la función de utilidad propuesta en [2]. La cual es la planteada en la ecuación 2.1. Teniendo como condición la expresión 2.2 que indica que la suma de las memorias repartidas no sea mayor a la memoria total a repartir, además, como se indica en 2.3 cada una debe ser mayor o igual a el mínimo de memoria que se le puede asignar a cada aplicación.

$$F(m) = \sum_{i=1}^{n} w_i \times U_i(m_i) \tag{2.1}$$

$$sujeto\ a: \quad \sum_{i=1}^{n} m_i \leq M \tag{2.2}$$

$$m_i \geq \underline{m_i}, \quad i = 1, \dots, n, \tag{2.3}$$

De las expresiones anteriores se tiene que n es el número de aplicaciones para el cuál se desea repartir la memoria total M, w_i es el peso o importancia asignada a la aplicación i, m_i es como se representa la memoria asignada a la aplicación i, $\underline{m_i}$ es la memoria mínima que se le puede asignar a la aplicación i, U_i es una función que me indica la utilidad individual de la aplicación i cuando se le asigna cierta cantidad de memoria.
U_i Se define como en 2.4, donde se expresa que es el producto del negativo de la frecuencia de una aplicación multiplicado por el tiempo de acceso estimado cuando tiene una cantidad de memoria asignada, el cual se define en 2.5.

$$U_i(m_i) = -f_i \times EAT\ (m_i) \tag{2.4}$$

$$EAT\ (m_i) = h_i(m_i) \times cd_i + [1 - h_i(m_i)] \times bd_i \tag{2.5}$$

Se tiene que $h_i(m_i)$ es el hit rate medido en la aplicación i cuando se le asigna m_i de memoria, cd_i es el tiempo de acceso a un objeto de la aplicación i que se encuentre en la caché, y bd_i es el tiempo de acceso a un objeto de la aplicación i que no se encuentre en la caché, es decir que esté en el backend.

Para crear el microservicio, tenemos tres plataformas de servicios en la nube como opciones: AWS Lambda [14], Azure Functions [15], Google Cloud Functions [16].
Las tres permiten implementar Funciones como Servicios (FaaS). Para el presente proyecto, optamos por AWS Lambda debido a las siguientes razones:

- Soporta más lenguajes de programación que los demás, incluyendo Java, Python, nodejs, C# y Go.
- Es un producto más maduro y con más documentación disponible.
- Es la plataforma preferida por el cliente.

Por tanto, se va a implementar una solución usando funciones como servicio en AWS Lambda, que contenga la implementación de los dos solvers, según el diseño mostrado en la Figura 2.1.

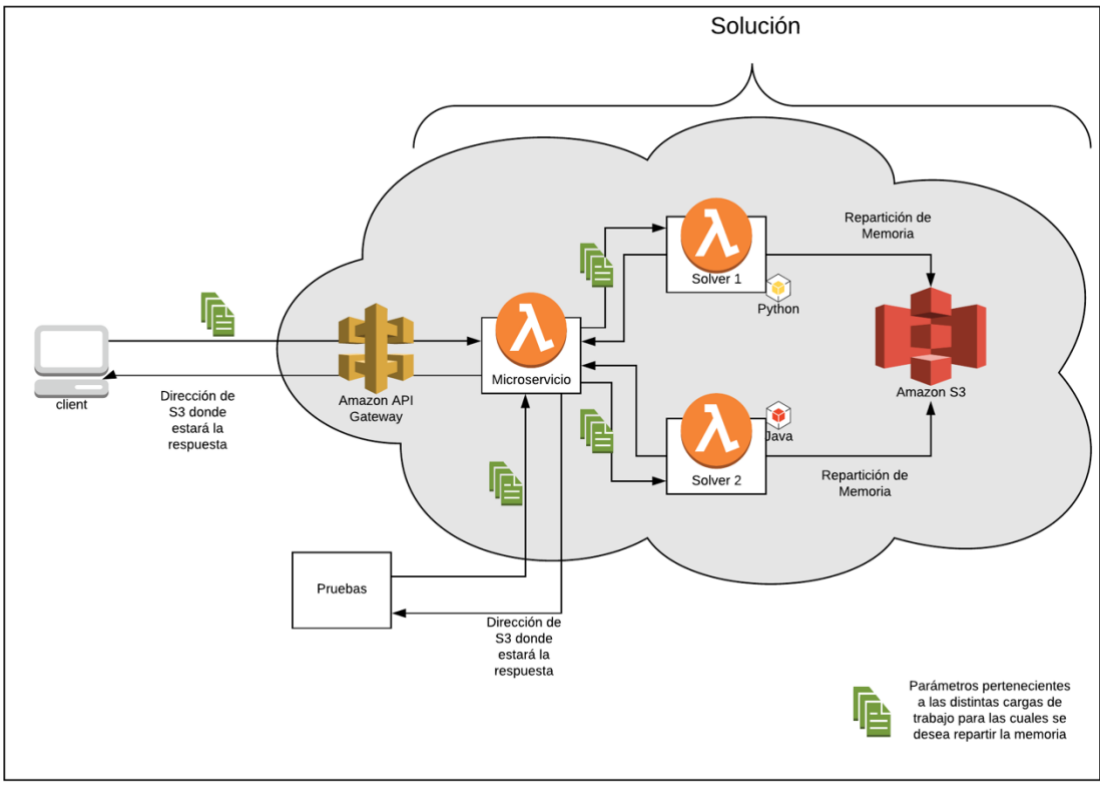


Figura 2.1 Diseño de la solución

2.1 Datos de entrada

Para cada aplicación que está compartiendo la memoria caché, necesitamos tener una representación del comportamiento de cada aplicación en la ventana de tiempo más reciente.

Los datos para resolver el problema son:

- Memoria total a repartir
- Tiempo de acceso a un objeto cacheado (por cada aplicación)
- Tiempo de acceso a un objeto en backend (por cada aplicación)
- Frecuencia de requerimientos (por cada aplicación)
- Mínimo de memoria que se le puede asignar (por cada aplicación)
- Peso asignado a cada aplicación (importancia de la aplicación)
- Curva de tasa de fallos (MRC), para cada aplicación

2.2 Herramientas de desarrollo

Dentro de las herramientas a utilizar se encuentran: AWS Lambda, API Gateway y Amazon S3.

2.2.1 AWS Lambda

AWS Lambda es un servicio de computación sin servidores (serverless). Usa un modelo de función como servicio (FaaS), en el cual se ejecuta el código cargado en la nube cada vez que se hace una invocación; AWS cobra por cada millón de invocaciones. AWS Lambda puede ejecutar código automáticamente en respuesta a varios eventos, como solicitudes HTTP a través de Amazon API Gateway, modificaciones a objetos en buckets de Amazon S3, actualizaciones de tablas en Amazon DynamoDB y transiciones de estado en AWS Step Functions. En este caso usaremos Amazon API Gateway.

2.2.2 Amazon API Gateway

Amazon API Gateway es un servicio completamente administrado que facilita la creación, mantenimiento, publicación, monitorización y la protección de las API. Usando la consola de administración de AWS se puede crear una API que hace las veces de “puerta delantera” para que las aplicaciones obtengan acceso a datos, lógica de negocio o funcionalidades desde sus servicios de backend, como cargas de trabajo ejecutadas en Amazon Elastic Cloud (Amazon EC2), código ejecutado en AWS Lambda o cualquier aplicación web. En este caso lo usaremos para ejecutar código en AWS Lambda.

2.2.3 Amazon S3

Amazon S3 es un servicio de almacenamiento de objetos en la nube, provisto por Amazon en el cual se pueden almacenar y recuperar grandes volúmenes de datos desde cualquier ubicación, se integra muy bien con gran cantidad de aplicaciones de terceros, En este caso lo utilizaremos para guardar las soluciones que nos da el microservicio implementado en AWS Lambda, y la función Lambda enviará como respuesta

2.3 Plan de implementación

Para la implementación de la solución hemos decidido resolverlo en tres módulos.

2.3.1 Módulo de solver 1

Este módulo será una función de AWS Lambda, la cual será invocada a través de API Gateway. Resolverá el problema de optimización usando el algoritmo de hill climbing, el cual será implementado en Python, basado en una implementación propuesta por el cliente, al cual se le realizarán las mejoras necesarias para las necesidades del proyecto.

El módulo recibe los datos de entrada mencionados anteriormente en la seccion 2.1 con respecto a las cargas de trabajo, además de la dirección de S3 donde

guardará la solución arrojada por el solver en forma de un vector ordenado que contendrá los tamaños a asignarse a cada aplicación.

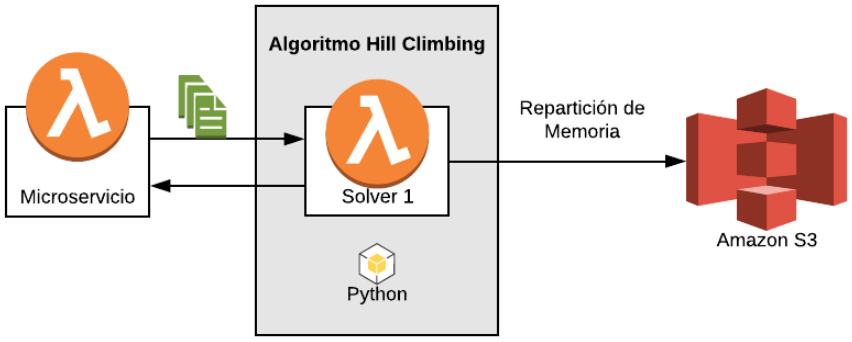


Figura 2.2 Diseño del módulo de solver 1

2.3.2 Módulo de solver 2

Este módulo será una función de AWS Lambda, la cual será invocada a través de API Gateway. Resolverá el problema de optimización usando el algoritmo evolutivo propuesto en [2], el cual está implementado en Java.

El módulo recibe los datos de entrada mencionados anteriormente en la sección 2.1 con respecto a los workloads además de la dirección de S3 donde guardará la solución arrojada por el solver en forma de un vector ordenado que contendrá los tamaños a asignarse a cada aplicación.

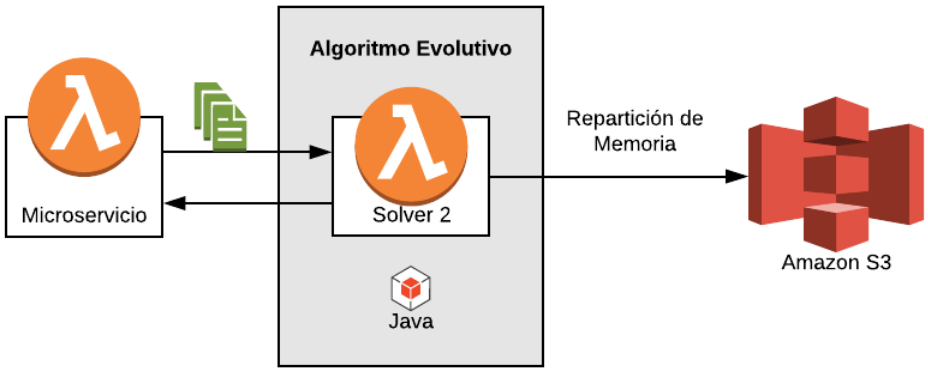


Figura 2.3 Diseño del módulo de solver 2

2.3.3 Módulo de integración y pruebas

Este módulo será una función de AWS Lambda, la cual será invocada a través de API Gateway. Se encargará de decidir cuál solver utilizar en base a la entrada de datos obtenida, una vez haya decidido, invocará la función Lambda que considere mejor. Con respecto a los datos de entrada recibirá los mencionados en la sección 2.1 los cuales a su vez los pasará como parámetros a la función al solver seleccionado además de la dirección de S3 donde guardar la respuesta, como salida le devolverá al cliente la dirección de S3 donde dentro de máximo 5 minutos podrá encontrar la respuesta de cómo repartir la memoria entre sus distintas aplicaciones.

Además, se realizarán pruebas de funcionalidad para validar los distintos casos de uso para comprobar si la salida del sistema respecto a los casos es lo que se espera.

2.4 Plan de recolección de datos

De los datos mencionados en la sección de datos de entrada (Sección 2.1), el grupo de investigación nos proporcionará las curvas de miss rate. Para las pruebas, junto con el cliente se definirán distintas configuraciones para probar (distintos: tamaños de memoria, MRC, tiempos, pesos, cantidad de workloads, etc). Los algoritmos a implementar en los solvers también son proporcionados por el grupo de investigación.

2.5 Fiabilidad de los datos

Las MRC (miss rate curves) que proporcionará el grupo de investigación se han obtenido en base a trazas de cargas de trabajo de aplicaciones reales (Yahoo, YouTube, Dreamworks). A estas trazas se las dividió en archivos con 1 millón de líneas o registros, y a cada uno de estos archivos se les corrió el algoritmo SHARDS (implementado por el grupo de investigación) para sacar las MRC.

Se tienen más de 100 MRC para crear configuraciones de pruebas.

CAPÍTULO 3

3. IMPLEMENTACIÓN Y ANÁLISIS DE RESULTADOS

3.1 Implementación

3.1.1 Módulo 1, solver que implementa hill climbing

El algoritmo de hill climbing también llamado algoritmo de escalada simple o ascenso de colina, consiste en un método de optimización matemática el cuál sirve para hallar máximos locales de alguna función.

En este caso se realizó esta implementación en Python para encontrar los valores de memoria para maximizar la función de utilidad introducida en el capítulo 2, Para tratar de encontrar el máximo local de la función y no un máximo local, al momento de encontrar un máximo, se realiza un salto más grande para tratar de encontrar un mejor valor de la función, esto se lo realiza varias veces con el afán de evitar que se estanque en un máximo local y obtener una mala respuesta.

Para tener este algoritmo dentro de una función de AWS Lambda, se debe crear una función handler la cuál entre sus parámetros recibe un parámetro llamado evento. En este caso la función Lambda se invoca mediante un post http usando API Gateway en la cual el evento es la carga del post, que en este caso es un archivo JSON el cual contiene los datos de entrada que necesita el Algoritmo y la función de utilidad, para que ésta pueda ser optimizada.

3.1.2 Módulo 2, Solver que implementa algoritmo evolutivo.

Solver con implementación del algoritmo evolutivo propuesto en [2], consiste en la optimización de un problema en base a una función de utilidad, para poder mejorar el resultado de esa función se recorre un lazo con un valor límite de iteraciones que se encargará de reemplazar la mejor respuesta encontrada en cada iteración por la anterior. La implementación de este algoritmo se realizó en Java.

Para subir este algoritmo como función dentro de AWS Lambda, se debe crear una aplicación de java con una clase principal llamada LambdaHandler. Para

poder invocar la función usamos API Gateway donde enviamos un requerimiento POST con un archivo JSON que contiene todos los datos de entrada para los parámetros del algoritmo, para que me devuelva el resultado de la optimización.

3.1.3 Módulo de Integración

Este último módulo va a ser una función en AWS Lambda en donde se encargará de manejar a que solver invocar dependiendo del número de aplicaciones de la configuración. Este módulo se implementó en Python, donde se evalúa la cantidad de aplicaciones para la cual repartir la memoria y en base a eso decidir que solver ejecutar.

3.2 Resultados

El siguiente análisis de resultados se lo hizo con el fin de contrastar los dos solvers implementados de tal manera de hacer los ajustes necesarios al microservicio para que funcione de la mejor manera posible.

Una vez implementado los 2 solvers, por cada uno, realizamos pruebas con distintas configuraciones con distintas cantidades de aplicaciones para las cuales hacer la repartición de memoria: 2, 3, 4, 5, 6, 7 y 8 aplicaciones. Para las configuraciones con menos de 7 aplicaciones, se efectuaron 20 pruebas con cada solver, y para las demás configuraciones se efectuaron 5 pruebas con cada solver. Uno de los propósitos de realizar estas pruebas es comprobar si el algoritmo logra estabilizarse, observando si las respuestas se parecen entre sí.

Debido a que los algoritmos son de optimización de la función de utilidad, el valor de la función utilidad más alto indica una mejor repartición de memoria, es por eso que de entre todas las pruebas por cada configuración, marcamos como “BEST” a la prueba que nos arrojó como resultado una mejor repartición de memoria.

La calidad de los resultados del algoritmo evolutivo [2], están ligados principalmente del valor *limit*, que es el número de iteraciones del lazo principal del algoritmo, para las configuraciones 2, 3, 4, 5 y 6 se usó el limit = 1000, en el caso de 6 se hizo una prueba adicional con limit = 10000 para ver qué impacto podía tener este cambio. Para las pruebas con 7 y 8 aplicaciones se usó limit = 5000.

Luego realizamos una comparación entre el BEST que nos arrojó cada algoritmo, el objetivo de esto fue para asegurarnos de que los dos algoritmos nos den respuestas similares, ya que están resolviendo el mismo problema.

3.2.1 Solver 1: Algoritmo hill climbing

Al ejecutar las pruebas con el Solver 1, encontramos que, con configuraciones de 2 y 3 aplicaciones, el algoritmo no era tan preciso con respecto a cuál era la mejor repartición de memoria, en las 20 pruebas nos dio respuestas muy diferentes, esto se lo puede observar en las figuras 3.1 y 3.2.

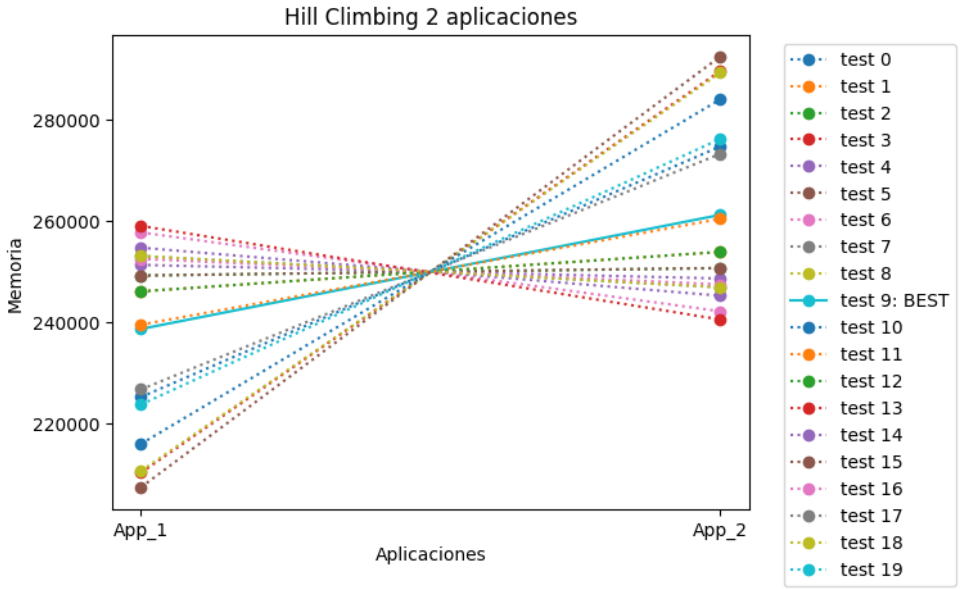


Figura 3.1 Repartición de 500 Mb memoria para 2 aplicaciones usando el algoritmo de hill climbing

Para las pruebas con más aplicaciones, el algoritmo se portó de manera un poco más precisa, es decir, las respuestas entregadas en las 20 pruebas eran más parecidas entre sí, esto se lo puede apreciar en las figuras 3.3, 3.4 y 3.5

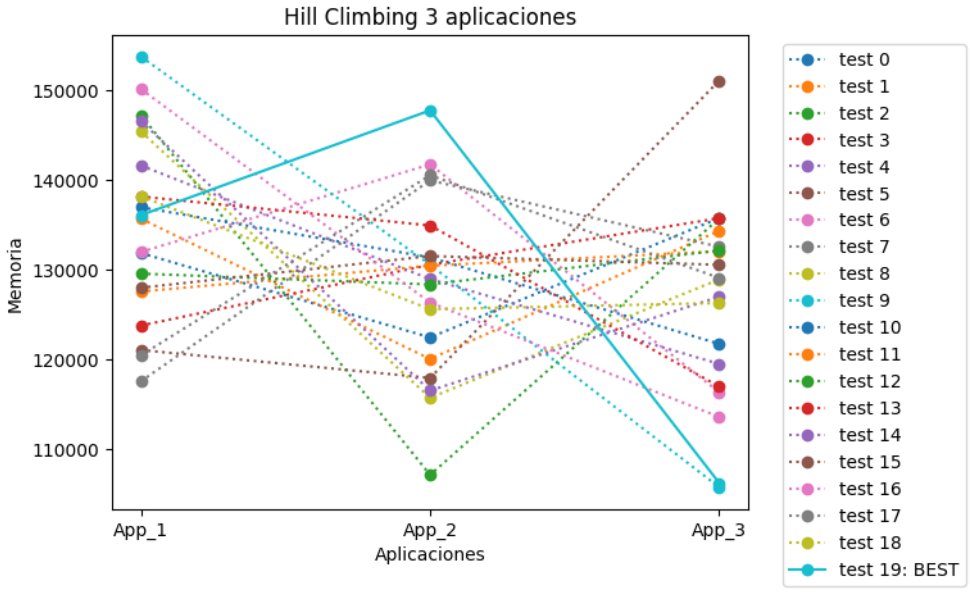


Figura 3.2 Repartición de 390 Mb de memoria para 3 aplicaciones usando el algoritmo de hill climbing

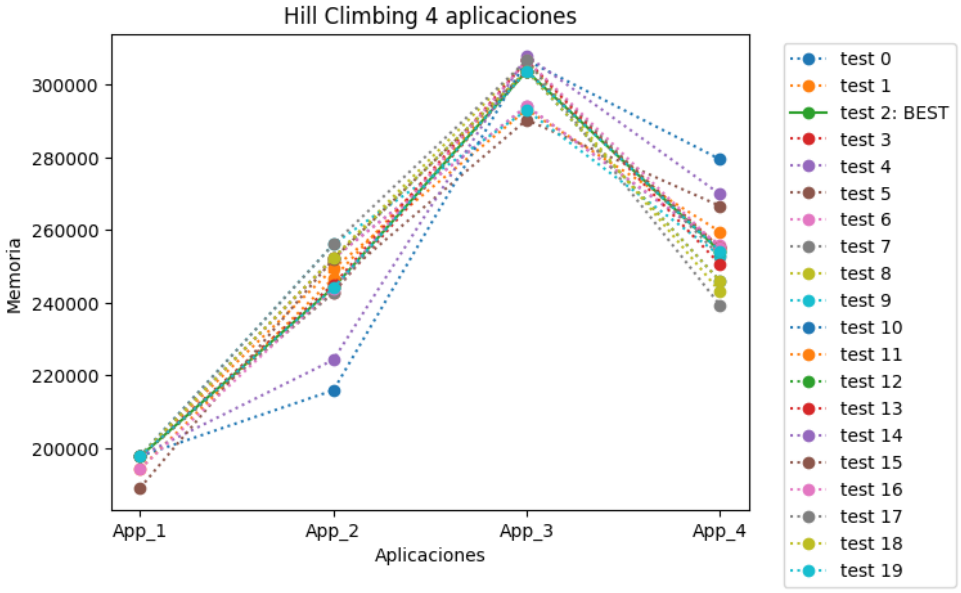


Figura 3.3 Repartición de 1000 MB de memoria para 4 aplicaciones usando el algoritmo de hill climbing

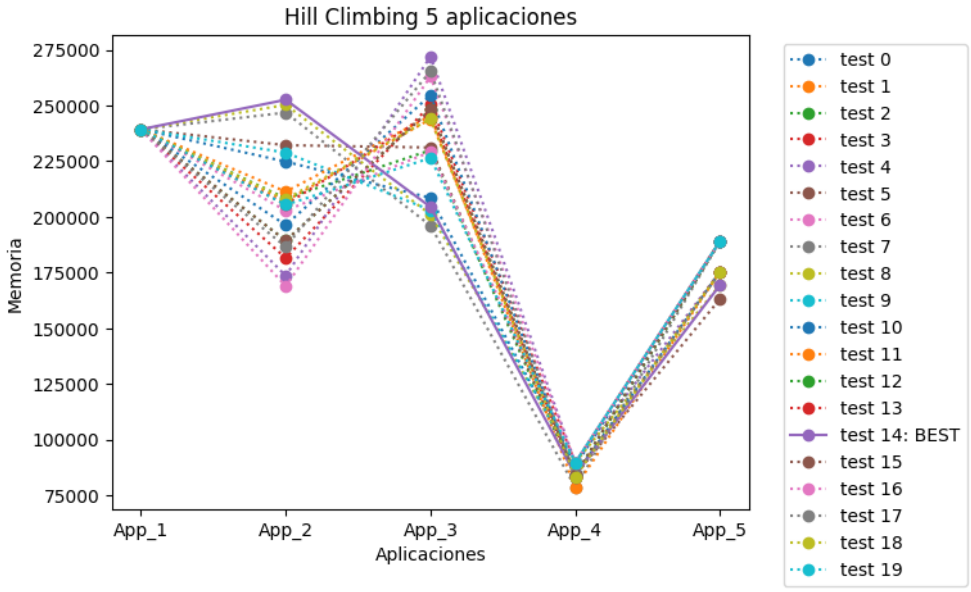


Figura 3.4 Repartición de 950 MB de memoria para 5 aplicaciones usando el algoritmo de hill climbing

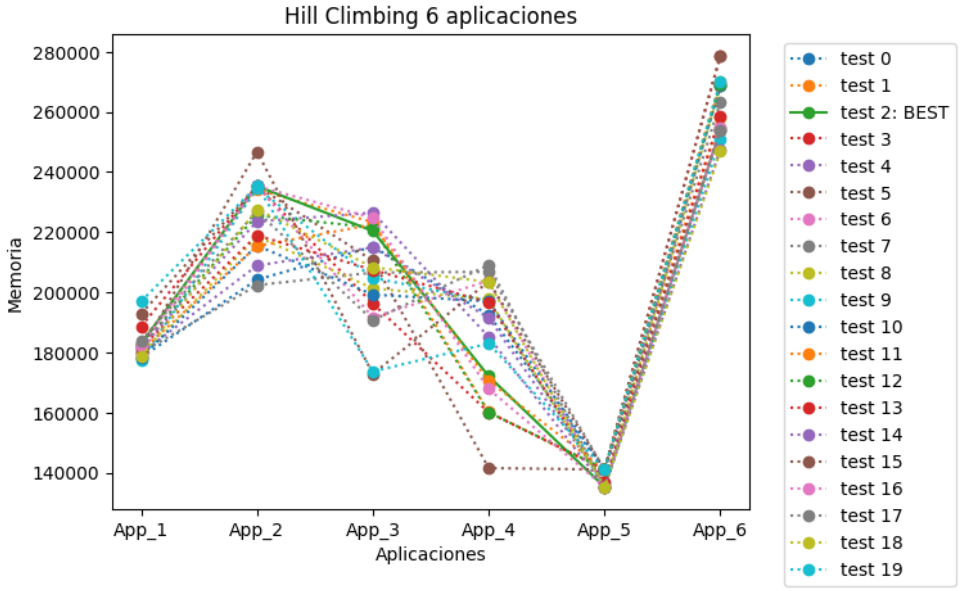


Figura 3.5 Repartición de 1200 MB memoria para 6 aplicaciones usando el algoritmo de hill climbing

Con configuraciones de 7 y 8 aplicaciones solo se realizaron 5 pruebas con cada configuración, pero son suficientes para apreciar que el algoritmo sigue arrojando resultados parecidos entre sí y cada vez se lo ve más estable (Figuras 3.6 y 3.7).

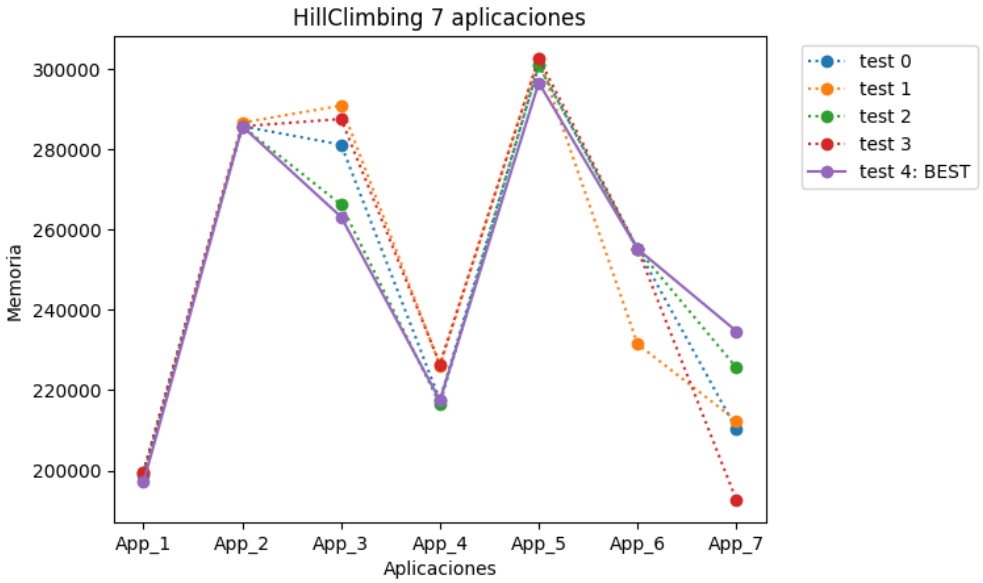


Figura 3.6 Repartición de 1750 MB memoria para 7 aplicaciones usando el algoritmo de hill climbing

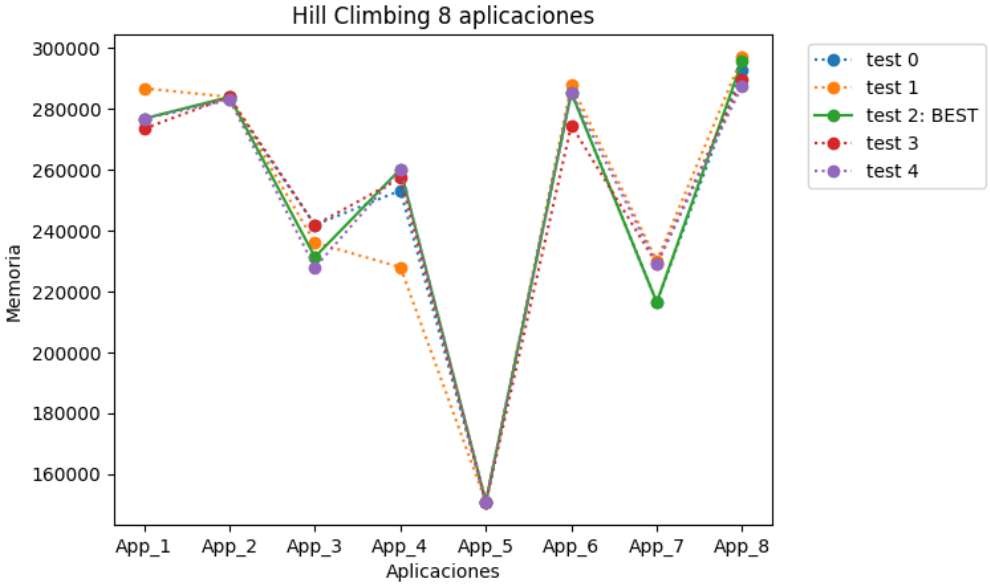


Figura 3.7 Repartición de 2000 MB memoria para 8 aplicaciones usando el algoritmo de hill climbing

3.2.2 Solver 2: Algoritmo evolutivo

En las configuraciones de 2 aplicaciones se encontró que a diferencia del algoritmo de hill climbing, este algoritmo es preciso, todas las 20 pruebas nos arrojaron prácticamente la misma respuesta, podemos ver una ilustración de esto en la figura 3.8.

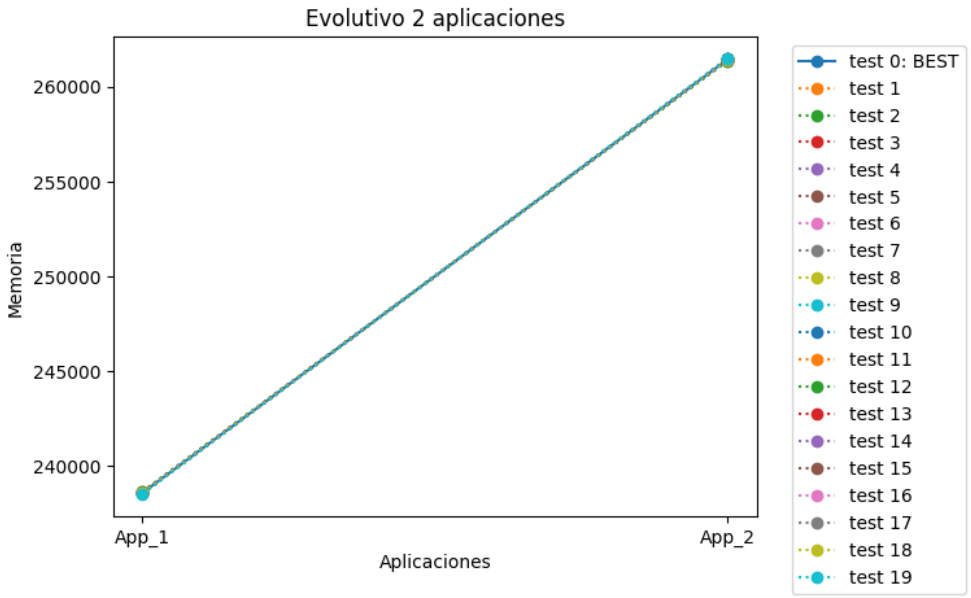


Figura 3.8 Repartición de 500 Mb memoria para 2 aplicaciones usando el algoritmo evolutivo con 1000 iteraciones

En las configuraciones de 3 aplicaciones, encontramos que, si bien los valores de repartición de memoria en las aplicaciones no logran estabilizarse por completo, la dispersión entre estos valores es pequeña, a diferencia de lo que se obtuvo con el algoritmo de hill climbing.

En las configuraciones de 4 y 5 aplicaciones, se puede ver una repartición y precisión más parecida a las obtenidas con hill climbing, pero aun así se puede apreciar que las obtenidas con este algoritmo son más parecidas entre sí (figuras 3.10 y 3.11).

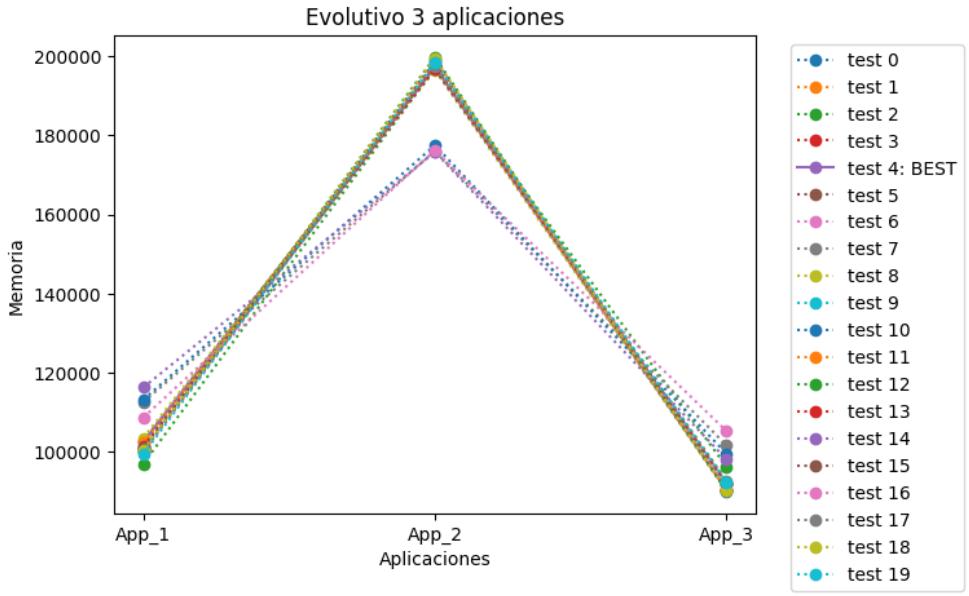


Figura 3.9 Repartición de 390 Mb memoria para 3 aplicaciones usando el algoritmo evolutivo con 1000 iteraciones

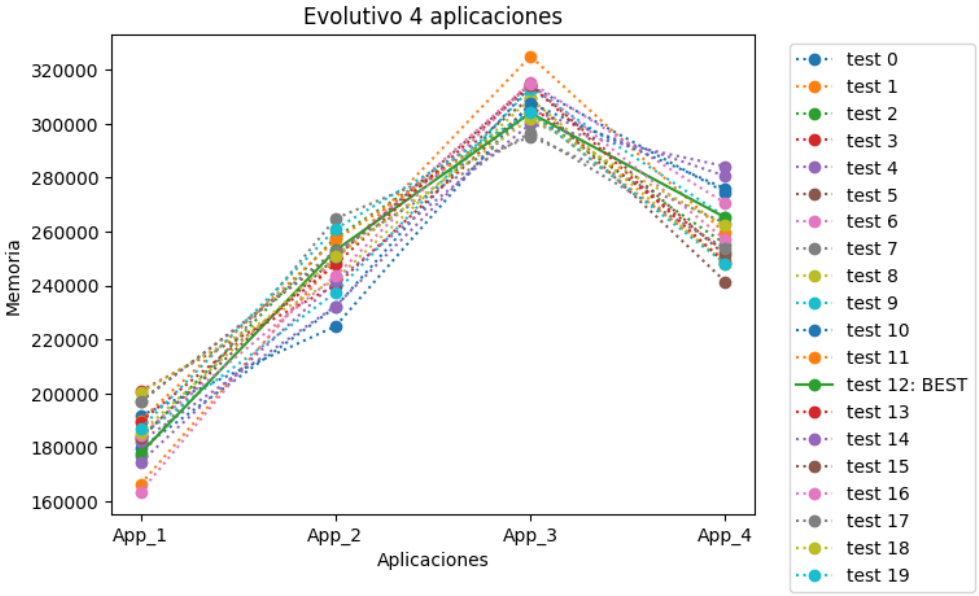


Figura 3.10 Repartición de 1000 Mb memoria para 4 aplicaciones usando el algoritmo evolutivo con 1000 iteraciones

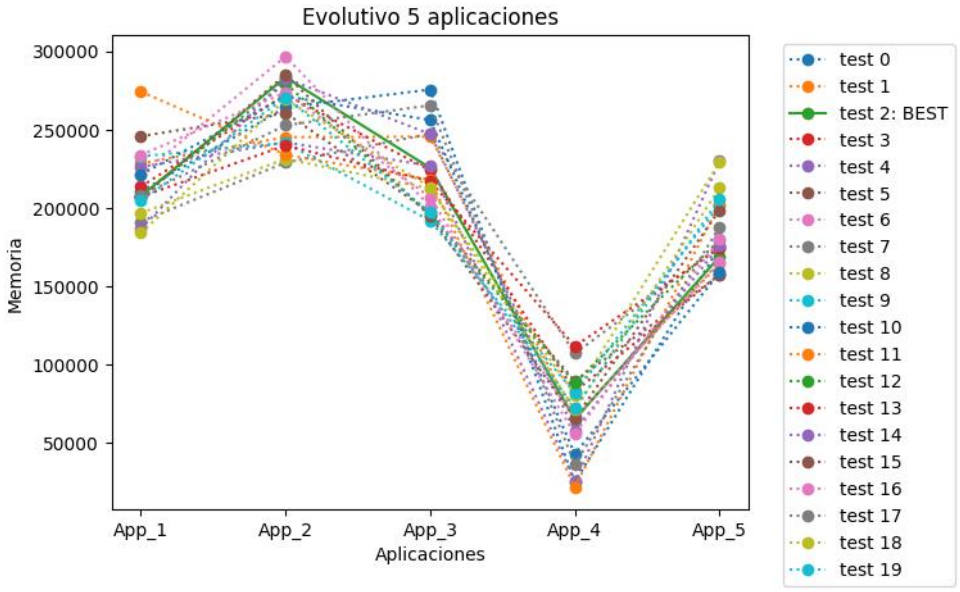


Figura 3.11 Repartición de 950 Mb memoria para 5 aplicaciones usando el algoritmo evolutivo con 1000 iteraciones

En las configuraciones de 6 aplicaciones con limit = 1000 pudimos apreciar que el algoritmo no nos arrojó respuestas precisas, estas difieren bastante entre sí (figura 3.12). Para obtener mejores resultados con las configuraciones de 6 aplicaciones se decidió cambiar el *limit a* 10000, donde se pudo apreciar que el algoritmo ganó precisión causando así que los distintos resultados sean más parecidos entre sí (figura 3.13), pero aun así el resultado no se compara al obtenido con hill climbing el cuál se ve más preciso (figura 3.5)

Para las pruebas con las configuraciones de 7 y 8 aplicaciones se decidió usar limit = 5000 ya que al ser más curvas el algoritmo se quedaba corto con limit = 1000. De igual manera en estos dos (figuras 3.14 y 3.15) casos se puede apreciar que en el algoritmo de hill climbing se obtienen soluciones más parecidas entre sí (Figuras 3.6 y 3.7).

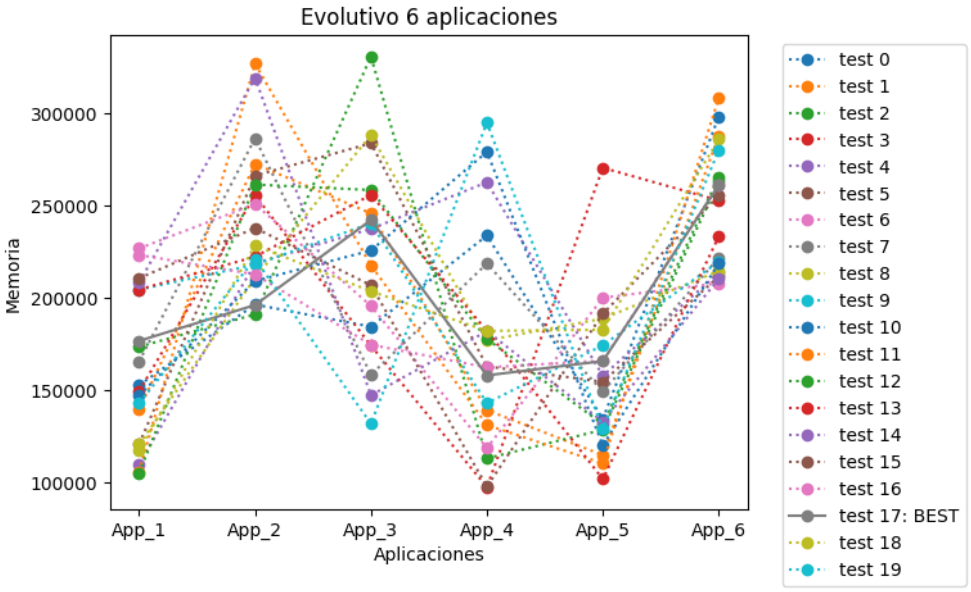


Figura 3.12 Repartición de 1200 MB memoria para 6 aplicaciones usando el algoritmo evolutivo con 1000 iteraciones

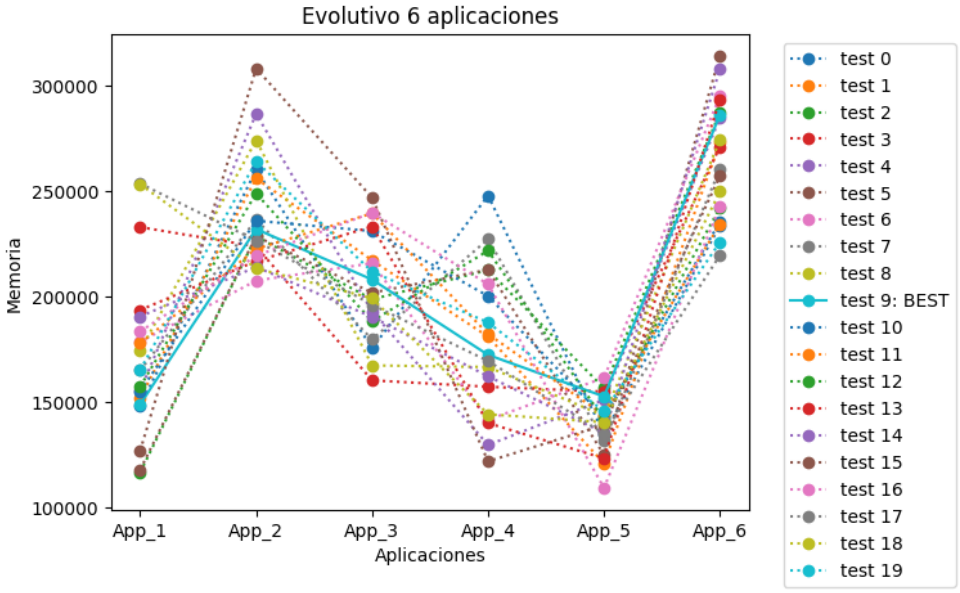


Figura 3.13 Repartición de 1200 MB memoria para 6 aplicaciones usando el algoritmo evolutivo con 10000 iteraciones

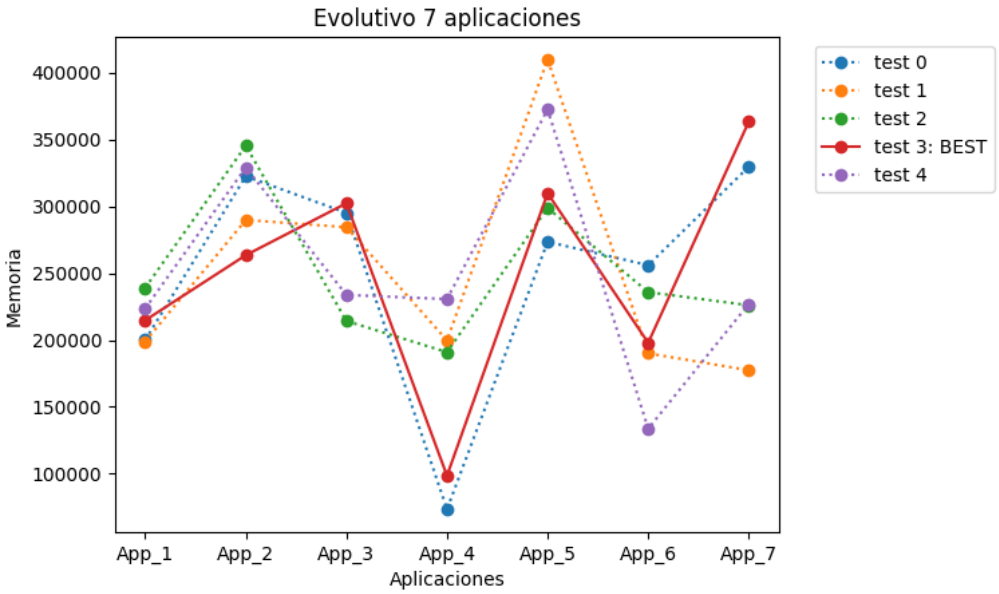


Figura 3.14 Repartición de 1750 MB memoria para 7 aplicaciones usando el algoritmo evolutivo con 5000 iteraciones

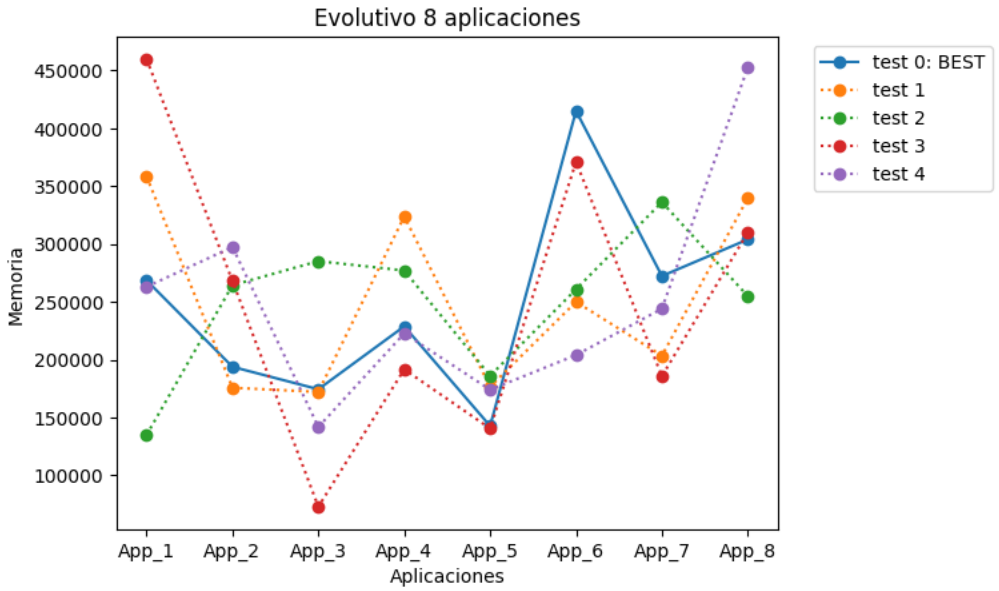


Figura 3.15 Repartición de 2000 MB memoria para 8 aplicaciones usando el algoritmo evolutivo con 5000 iteraciones

3.2.3 Comparación entre Solvers

Al ver las gráficas presentadas en la sección anterior, se puede apreciar a simple vista que el solver evolutivo tiene resultados mejores y más precisos para cuando se trata de pocas aplicaciones (2, 3, 4 hasta 5), a partir de 6 aplicaciones, se puede observar que el algoritmo empieza a ser menos preciso, esto se debe principalmente a el límite en el número de iteraciones o generaciones, se podría aumentar este número pero si se compara el costo en tiempo con la mejora que se obtiene, se llega a la conclusión de que no es conveniente aumentar tanto el número de generaciones. En su defecto, el algoritmo de hill climbing, es más preciso cuando se aumenta el número de aplicaciones.

Se decidió realizar una comparación de los valores de la función de utilidad de las soluciones “BEST” que se obtuvieron de las pruebas anteriores de cada configuración por cada algoritmo, además se agregó a la comparación el valor de la función de utilidad en el caso de que se repartiera la misma cantidad de memoria a cada aplicación (tabla 3.1).

Tabla 3.1 Comparación de la función de utilidad de reparticiones obtenidas mediante distintos métodos

Número de Aplicaciones. / Algoritmo Usado	2	3	4	5	6	7	8
Evolutivo	-13710	-12201	-11309	-31747	-19154	-24117	-42421
Hill Climbing	-13712	-12381	-11312	-31777	-19116	-24074	-42214
Repartición Equitativa	-13731	-12515	-11384	-32186	-19203	-24162	-42351

Se resaltó el método que se comportaba mejor por cada número de aplicaciones, comprobando así lo que se mencionó anteriormente, que el algoritmo evolutivo arrojó mejores resultados para configuraciones con menos de 6 aplicaciones.

Algo más que se puede apreciar de esta tabla que aunque los valores sean cantidades grandes, la diferencia entre una repartición óptima y una que no lo es, tal como repartir la memoria de manera equitativa, es de unas cuantas decenas de unidades, esto me indica que aunque aparentemente la diferencia parezca poca entre un algoritmo y otro, es verdaderamente significativa; véase el caso de la configuración con 7 aplicaciones, que el mayor valor es -24074, separándose 43 unidades de lo que se obtuvo con el algoritmo evolutivo, -24117, no parece mucho, pero si lo comparamos este segundo valor con el obtenido en el caso de repartirle a las 7 aplicaciones cantidades iguales de memoria, vemos que tienen 45 unidades de diferencia; Y al ver las gráficas 3.6 y 3.14 vemos que la solución brindada es bastante diferente a repartir equitativamente las memorias.

3.2.4 Comparación de tiempos entre Solvers.

Se realizaron varias comparaciones:

En la figura 3.16 se puede ver que el algoritmo de hill climbing se toma más tiempo que el evolutivo cuando limit es igual 1000, pero en el caso que limit es igual a 10000 (figura 3.17, solo para la configuración con 6 aplicaciones) el algoritmo evolutivo se toma mucho más tiempo, y como vimos en la sección anterior, los resultados aun así no superan a los obtenidos por hill climbing.

En la figura 3.18 se comparan los tiempos de los 2 algoritmos con todas las configuraciones que se realizaron, en el caso del algoritmo evolutivo con limit = 5000 para cuando se tenían 7 y 8 aplicaciones.

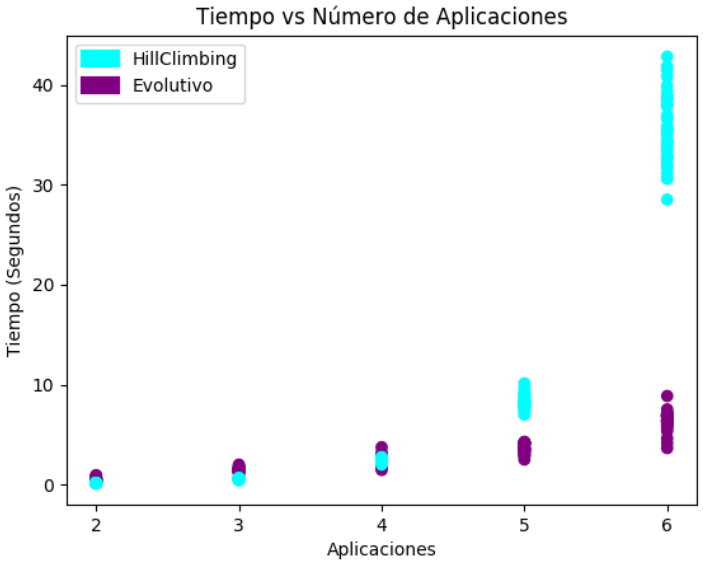


Figura 3.16 Tiempo vs Número de aplicaciones, de 2 a 6 aplicaciones con limit = 1000 para el solver evolutivo

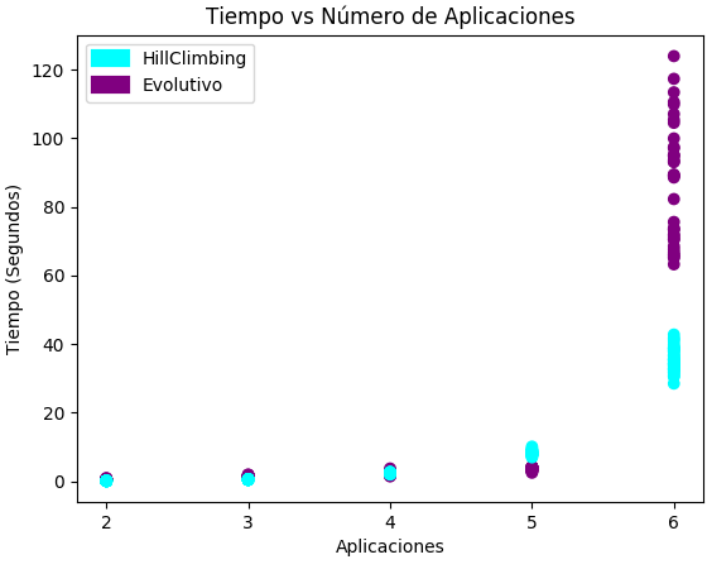


Figura 3.17 Tiempo vs Número de aplicaciones, de 2 a 5 aplicaciones con limit = 1000, y 6 con limit = 10000 para los casos del solver evolutivo

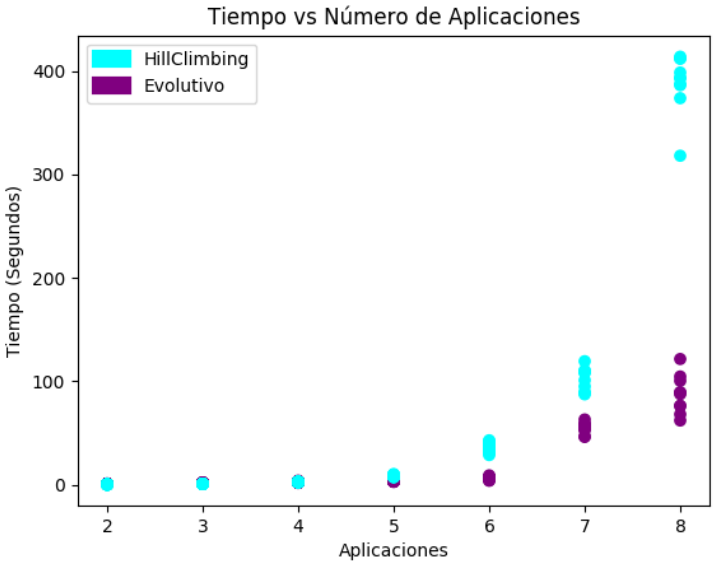


Figura 3.18 Tiempo vs Número de aplicaciones. Para el caso del algoritmo evolutivo se utilizó limit = 1000 en los casos de 2 a 6 aplicaciones, y limit = 5000 en los casos de 7 y 8

3.2.5 Resultados del microservicio

Analizando los resultados obtenidos a partir de cada solver se implementó el microservicio de tal manera que cuando se invoque el servicio, para una repartición de memoria de 5 aplicaciones o menos, el microservicio utilice el solver con el algoritmo evolutivo, caso contrario, que el solver utilizado sea el que implementa el algoritmo de hill climbing.

Se realizaron pruebas ya con el microservicio implementado para configuraciones con cantidades diferentes de aplicaciones, de 2 a 8, en AWS Lambda para estimar el costo que tendría mantener este servicio activo. Como se puede apreciar en la tabla 3.2 los precios no son altos, teniendo como máximo 670 dólares en el caso de que se invoque un millón de veces el servicio para una repartición de memoria entre 8 aplicaciones, esto sería menos de un centavo por cada invocación.

Para realizar estas pruebas se configuró las funciones Lambda con la máxima capacidad computacional posible, de esto también depende el costo por cada 100 ms, y por ende el tiempo de respuesta de la función Lambda.

Tabla 3.2 Precios por millón de invocaciones del microservicio.

Número de Aplicaciones	2	3	4	5	6	7	8
Tiempo facturado por invocación en milisegundos	600	900	1300	1500	1600	4800	13700
Costo aproximado por millón de invocaciones en USD	30	44	65	73	78	235	670

Se debe tener en cuenta que el microservicio es capaz de brindar respuesta para configuraciones con mayor número de aplicaciones, teniendo en cuenta que mientras se desee repartir memoria entre más cargas de trabajo el servicio se tomará más tiempo en dar una respuesta, lo cual implica más costo, pero sin embargo las pruebas realizada nos bastan para comprobar que los mismos no son altos, con respecto a los beneficios que se podrían obtener.

CAPÍTULO 4

4. CONCLUSIONES Y RECOMENDACIONES

4.1 Conclusiones

Con los dos algoritmos implementados se puede resolver del problema de optimización, aunque para cuando se necesita repartir memoria entre varias aplicaciones, más de 6, el algoritmo de hill climbing funciona mejor, que el evolutivo, debido a que los valores de repartición de memoria logran estabilizarse y consiguen un mejor valor de utilidad. Y para configuraciones con menos aplicaciones funciona mejor el evolutivo debido a lo mismo con el de hill climbing, los valores de repartición de memoria se estabilizan y el valor de utilidad es mejor.

Además, con respecto al tiempo cuando se realizaron comparaciones, se pudo comprobar que, a mayor número de aplicaciones, el tiempo de ejecución es mayor y para cada solver se comporta de manera diferente. Para el algoritmo de hill climbing se puede observar en la figura 3.18 toma un comportamiento exponencial.

Para el algoritmo evolutivo, el tiempo depende más por el valor *limit.* El valor del atributo limit del algoritmo evolutivo es directamente proporcional al tiempo de ejecución del algoritmo y a la precisión y exactitud de las respuestas, un valor de limit ideal dependerá de la cantidad de aplicaciones para las cuales se desee repartir la memoria.

Es factible llevar el servicio a comercialización. Teniendo en cuenta que las empresas podrían querer repartir memoria entre más de 8 aplicaciones, decidimos tomar un promedio de 2000 dólares por millón de invocaciones, es decir USD 0.20 por cada 100 invocaciones del servicio. Supongamos que una empresa utiliza nuestra solución 20 veces al día para ajustar la repartición de memoria de sus servidores de caché, esto serían 600 invocaciones al mes, teniendo en cuenta lo antes mencionado nos costaría USD 1.20 esas 600 invocaciones que realiza la empresa. Se puede realizar un modelo de suscripción mensual de USD 20.00 y daría resultados positivos tanto para nosotros como proveedores del servicio y para la empresa.

Una empresa podría tener varias aplicaciones en un servidor de caché, entre ellas una página de venta de ropa, pero al tener la memora repartida de una manera poco eficiente, la pagina sufre de problemas de alta latencia. Al usar nuestro servicio la empresa podría repartir esa memoria de una manera eficiente, así reduciendo su latencia y posiblemente mejorar sus ingresos, ya que la pagina dará una mejor experiencia de usuario a sus clientes.

4.2 Recomendaciones

Se recomienda probar con otros algoritmos para optimizar la función de utilidad y contrastarlos con los implementados en este trabajo para comparar velocidad y precisión.

Se recomienda hacer pruebas con configuraciones reales, y así determinar cuánto mejora la latencia promedio una vez reconfigurada la repartición de memoria en base a las respuestas brindadas por el microservicio.

Se recomienda hacer un estudio de mercado para determinar si las empresas que utilizan memoria caché en sus servidores contratarían este servicio, además cuánto estarían dispuesto a pagar.

BIBLIOGRAFÍA

[1] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In NSDI, 2016.

[2] C. Abad, A. Abad and L. Lucio, Dynamic Memory Partitioning for Cloud Caches with Heterogeneous Backends, In ICPE, 2017

[3] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15), Santa Clara, CA, July 2015. USENIX Association.

[4] K. Cruz, Historia del cloud computing, *RITS*, vol.1, no. 7, pp. 51, noviembre, 2012.

[5] Memcached - a distributed memory object caching system, *Memcached.org*, 2018. [Online]. Available: https://memcached.org/. [Accessed: 27- May- 2018].

[6] Redis, *Redis.io*, 2018. [Online]. Available : https://redis.io/. [Accessed: 27- May- 2018].

[7] N. Shalom, "Amazon found every 100ms of latency cost them 1% in sales. | GigaSpaces Blog", *The GigaSpaces Technologies Blog*, 2018. [Online]. Available: https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/. [Accessed: 27- May- 2018].

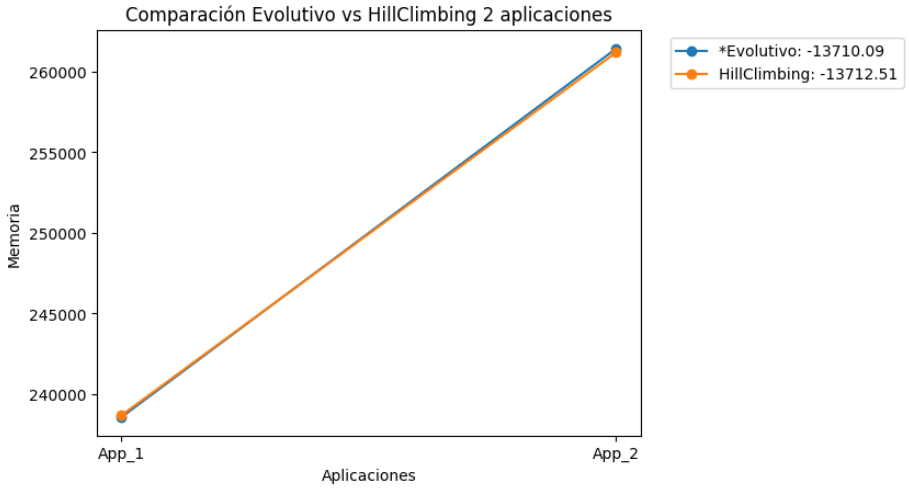
[8] "A history of cloud computing", *ComputerWeekly.com*, 2018. [Online]. Available : https://www.computerweekly.com/feature/A-history-of-cloud-computing. [Accessed: 20- Aug- 2018].

[9] S. Swamy, Cloud computing adoption journey within organizations, En *International Conference on Cross-Cultural Design*, Springer, Berlin, Heidelberg, 2013. p. 70-78.

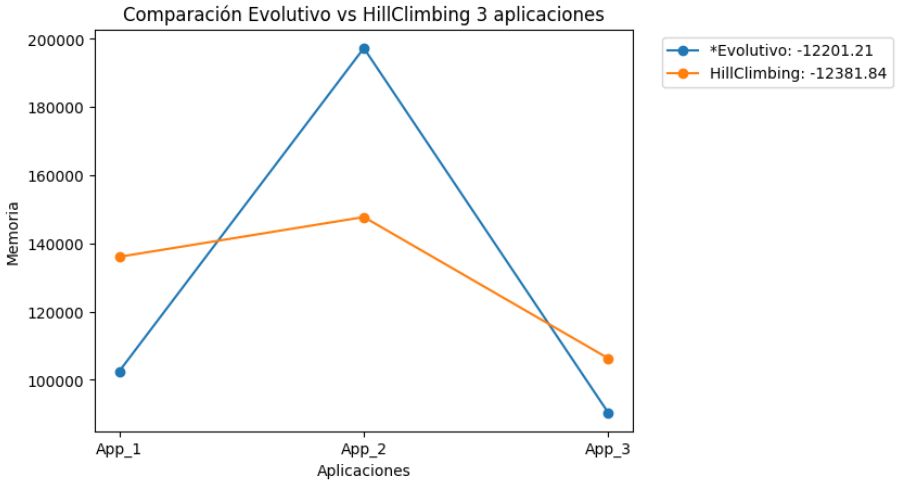
[10] Q. Zhang, L. Cheng, R. Boutaba, "Cloud computing: State-of-the-art and research challenges", *J. Internet Serv. Appl.*, vol. 1, no. 1, pp. 7-18, May 2010.

[11] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. Technical Report CSRG-626, Department of Computer Science, University of Toronto, 2015.

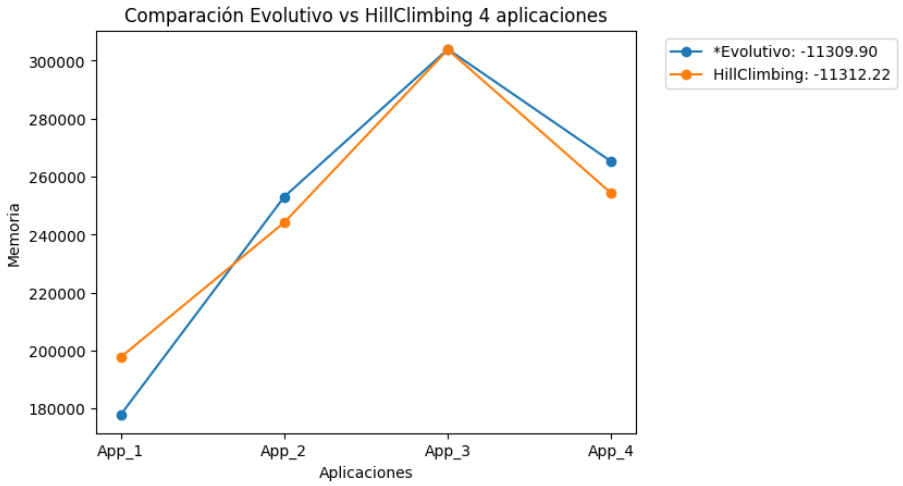
ANEXOS



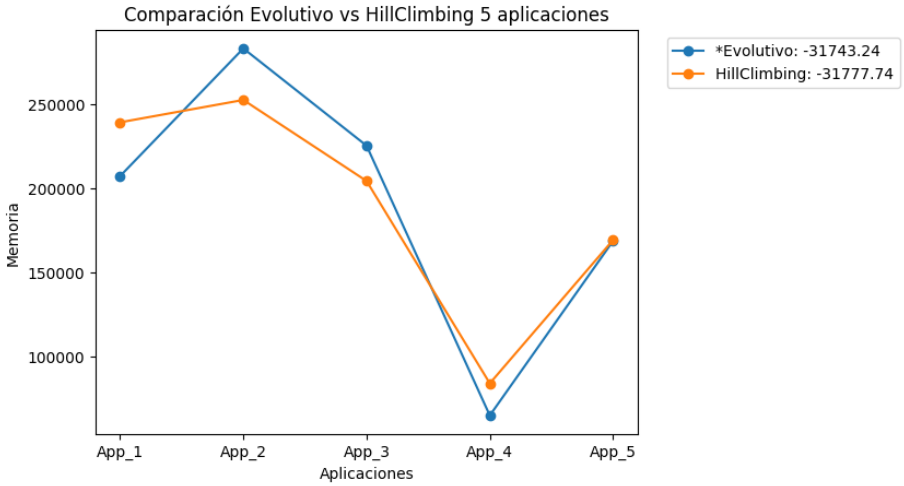
Anexo 1 Comparación de los algoritmos usando una configuración de 2 aplicaciones



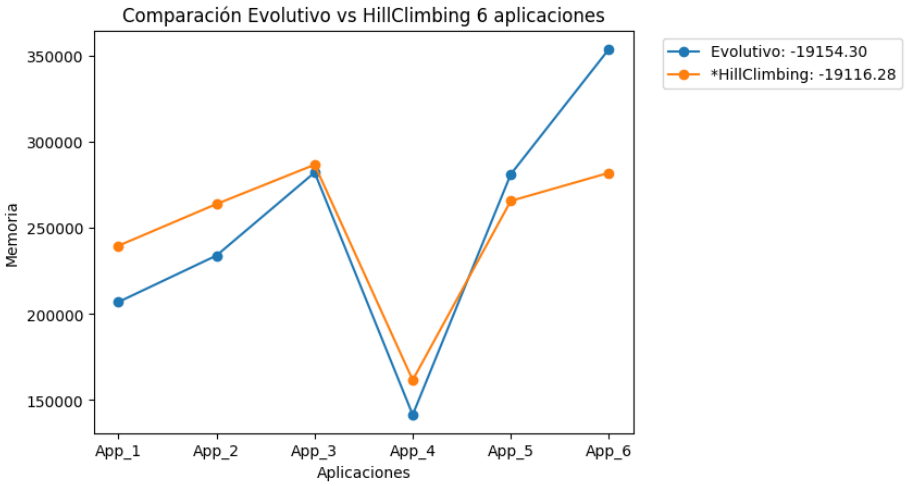
Anexo 2 Comparación de los algoritmos usando una configuración de 3 aplicaciones



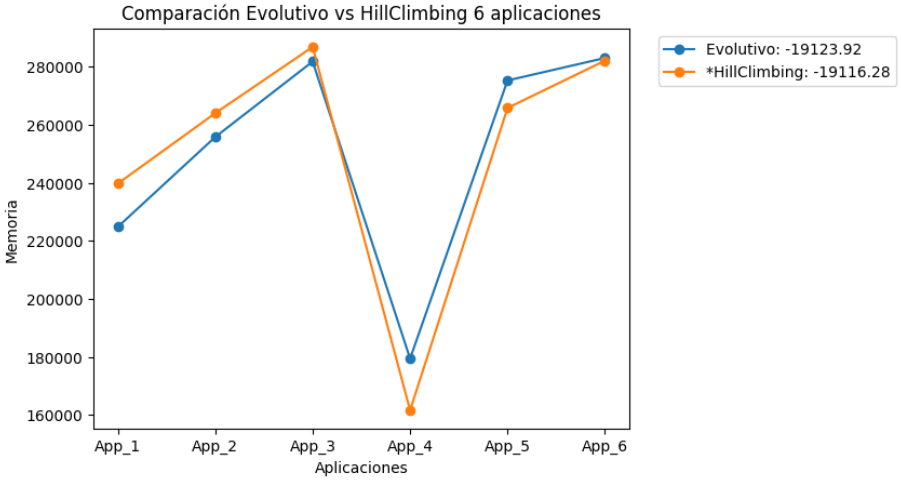
Anexo 3 Comparación de los algoritmos usando una configuración de 4 aplicaciones



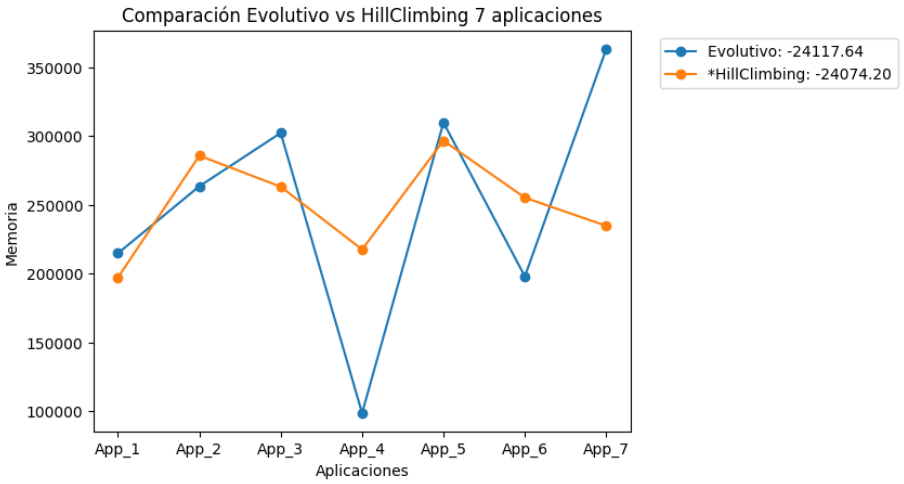
Anexo 4 Comparación de los algoritmos usando una configuración de 5 aplicaciones



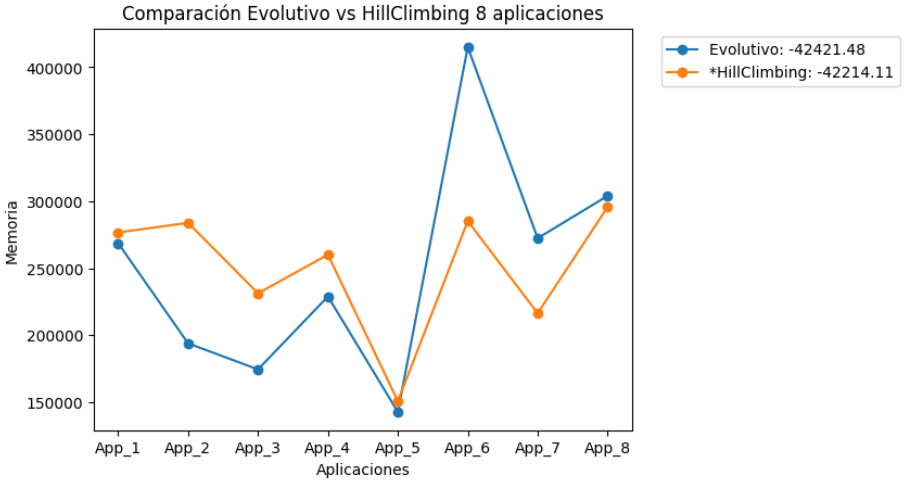
Anexo 5 Comparación de los algoritmos usando una configuración de 6 aplicaciones limit = 1000



Anexo 6 Comparación de los algoritmos usando una configuración de 6 aplicaciones limit = 10000



Anexo 7 Comparación de los algoritmos usando una configuración de 7 aplicaciones



Anexo 8 Comparación de los algoritmos usando una configuración de 8 aplicaciones