
UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA DE SISTEMAS

PROYECTO DE GRADO

# Viabilidad de la Arquitectura Serverless en Aplicaciones Web: Análisis y Desarrollo



Por

Br. Ali David Rodríguez Terán

Tutor: Dra.  Beatriz Sandia

Febrero 2020

Viabilidad de la Arquitectura Serverless en Aplicaciones Web: Análisis y Desarrollo

Br. Ali David Rodríguez Terán

Proyecto de Grado — Sistemas Computacionales, 59 páginas

## Resumen

La constante evolución del desarrollo web da espacio a la creación de nuevos conceptos que buscan minimizar los costos de implementación y operación. Serverless nace como una arquitectura que pretende disminuir la dependencia de un servidor constantemente levantado y a la espera de peticiones por largos periodos de inactividad.

En este trabajo se presenta la determinación de la viabilidad de la arquitectura serverless a través del desarrollo de dos servicios webs, uno en una arquitectura tradicional y el otro bajo la arquitectura Serverless, que han sido sometidos a una batería de pruebas de carga con la finalidad de obtener métricas reales para la comparación entre ambas arquitecturas.

Para la realización del proyecto se ha empleado el método evolutivo con entrega de prototipos, en el cual progresivamente se avanza sobre las distintas etapas de desarrollo, hasta alcanzar los objetivos definidos. La implementación abarca una amplia gama de lenguajes y herramientas, tales como: Frameworks, Spring-Boot, Node.js y Serverless; os lenguajes Java y JavaScript; herramientas como Apache Jmeter, Apache Tomcat y servicios como AWS Lambda y control de versiones con GitHub.

**Palabras clave:** Servicio Web, Ingeniería Computacional, Serverless, Prueba de Carga, Monolithic.

**Índice**

## Índice de Figuras

# Índice de Tablas

# Capítulo 1

## Introducción

El desarrollo web es una de las áreas del mundo de la programación que más ha tenido auge en la última década. La capacidad de los prestadores de servicios para ofrecer sus productos mediante plataformas electrónicas representa un avance monumental en el mercado contemporáneo y ha cambiado definitivamente los paradigmas del mismo.

En la constante evolución del desarrollo web se buscan, de manera permanente, soluciones que economicen y aumenten el rendimiento, tanto de los procesos de desarrollo como del mantenimiento de los servicios y aplicaciones webs.



El concepto de Serverless nace de la necesidad de minimizar los tiempos de ocio de los servidores, pues los modelos de servidores tradicionales requieren de un servidor activo y en espera, incluso si no existen peticiones que atender por extensos periodos de tiempo. Serverless propone levantar solamente una instancia del servidor cuando un cliente realice un llamado al mismo, y una vez procesadas todas las peticiones, y caer en un nuevo periodo de inactividad, se eliminaría la instancia actual del servidor. Por lo tanto los prestadores de servicios solamente podrían cobrar por el tiempo real de uso de los servidores.

Sin embargo, el auge de un concepto o tecnología nueva genera muchas dudas, entre las cuales las más recurrentes son; ¿Es esta tecnología eficiente?, ¿Vale la pena migrar o adaptar los servicios ya existentes a este nuevo concepto?, ¿Es

rentable desarrollar bajo este paradigma?. La adopción de una nueva idea siempre es un proceso lento que requiere amplio estudio y consideración. Los estudios comparativos de los modelos ya existentes contra las nuevas ideas y propuestas, son esenciales para la adopción o el olvido de las nuevas tecnologías.

Por otro lado, en la actualidad, se cuentan con numerosas herramientas y servicios que facilitan la creación de pruebas de conceptos para realizar mediciones sobre los parámetros reales de ejecución.

En este proyecto se plantea la comparación de dos servicios webs creados uno bajo la arquitectura Monolithic tradicional denotado con el nombre “Roshar”, y el otro bajo la arquitectura Serverless, denotado por el nombre “Scadrian”. Estos servicios cuyo objetivo es el procesamiento de peticiones para el cálculo del camino más corto en un grafo, constituirán un modelo al cual se le aplicara una batería de pruebas de carga empleando el producto Apache Jmeter, el cual proveerá las métricas necesarias para evaluar el desempeño de la arquitectura Serverless, y así determinar su viabilidad.

## 1.1 Planteamiento del Problema

El desarrollo de aplicaciones web nace de la necesidad de compartir soluciones de problemas locales a un ámbito global, la capacidad de compartir soluciones a dicha escala implica una amplia demanda de recursos, entre los cuales encontramos memoria y capacidad de cómputo, es por esta razón que nace un nuevo mercado donde proveedores ofrecen distintas soluciones a los problemas de quienes buscan desarrollar de manera rápida y efectiva los servicios webs.

Múltiples soluciones a lo largo del tiempo se fueron desarrollando hasta llegar a la arquitectura dependiente de servidores; realizar el procesamiento en una maquina aislada restando el consumo de recursos del cliente, no solo ampliaba la gama de clientes que pueden acceder al servicio sino que también ofrece un incremento en la seguridad, flexibilidad y centralización de los datos. Este modelo es el más popular en la actualidad. Sin embargo, existe una variedad de aplicaciones que no se benefician completamente de este paradigma, específicamente las aplicaciones ligeras de baja latencia y baja interacción con los usuarios; es decir, aplicaciones con un amplio espectro de tiempo ocioso, las cuales están obligadas a mantener sus servidores operativos en todo momento.

Muy recientemente aparece la arquitectura Serverless para aplicaciones web, cuya idea fundamental es disminuir la necesidad de tener un servidor funcional constantemente. Esto implica el cargo del servicio completo, indiferentemente del tiempo que el mismo no sea empleado. La premisa fundamental de Serverless es eliminar totalmente el servidor y solamente levantar el mismo en caso de que exista una solicitud de un usuario, lo que significaría que los proveedores de servicio solo

pueden cobrar por los recursos utilizados, disminuyendo así el costo operativo de los titulares de las aplicaciones y aumentando la efectividad del servicio.

Por ser la arquitectura Serverless un desarrollo tecnológico en proceso, con una alta difusión, pero aun con pocas implementaciones, se plantea desarrollar pruebas que permitan determinar la viabilidad de su uso, por su rendimiento y la reducción de costos.

## 1.2 Justificación

El tiempo que se invierte en el desarrollo de las aplicaciones web representa un costo, si se logra simplificar la configuración de las aplicaciones y los servidores, este puede disminuir, para empresas que deben contratar estas horas hombres implican economizar, si además estos entes logran disminuir el tiempo que se mantienen activos los servidores esto hace que se maximicen las ganancias, reduciendo los gastos en la fase operacional y de implementación, haciendo que la competitividad en el mercado aumente, además por ser un desarrollo reciente esto lograría que el peligro de llegar a quedar obsoletos sea erradicado.



## 1.3 Objetivos

### 1.3.1 Objetivo General

Determinar la viabilidad de la Arquitectura Serverless en Aplicaciones Web mediante su análisis y desarrollo.

### 1.3.2 Objetivos Específicos

1.3.2.1 Analizar la arquitectura Serverless

1.3.2.2 Desarrollar una aplicación web con arquitectura Serverless y analizar su rendimiento

1.3.2.3 Desarrollar una aplicación web con arquitectura Monolithic y analizar su rendimiento

1.3.2.4 Ejecutar pruebas de estrés a las aplicaciones web desarrolladas.

1.3.2.5 Comparar la arquitectura Monolithic contra la arquitectura Serverless.

## 1.4 Alcance

Se pretende desarrollar un servicio web basado en la arquitectura Serverless alojado y servido por AWS Lambda y el Framework Serverless en el lenguaje Java Script. Además un servicio web basado en la arquitectura monolithic alojado en un servidor de Apache Tomcat y basado en el Framework Spring Boot en el lenguaje Java. Por último una batería de pruebas de cargas empleando la aplicación Apache Jmeter con la finalidad de determinar la viabilidad de la arquitectura serverless en el desarrollo de servicios web.

# Capítulo 2 Marco Teórico

## 2.1 Antecedentes

Uno de los aspectos fundamentales para las migraciones a nuevas tecnologías es la obtención de beneficios de algún tipo, en el caso específico de la arquitectura Serverless, si bien es cierto que en el aspecto técnico el reducir tiempo en la configuración de políticas de réplicas de servidores, políticas de seguridad, el auto escalado, el balanceo de cargas, proxy, contenedores de recursos entre otros para que estos los asuma AWS Lambda, es un beneficio considerable, aunque para algunos la incapacidad de customización de las configuraciones, termina por no serlo totalmente. Lo que lleva a que se plantee cual es el beneficio real que ofrece Serverless.



En el año 2019 una serie de investigadores, entre los que se encuentran Alverro Alda, Fernando Álvarez, Gabriel Díaz, Martin Evgeniev, Hornillo Pancho junto con BBVA Labs brindan una respuesta significativa a esta interrogante mediante su trabajo de investigación “La Economía de las Arquitecturas ´Serverless´”. En dicho estudio se plantea cuatro casos de estudio para determinar cuándo es más apropiado usar AWS Lambda en vez de AWS EC2 (Amazon Elastic Compute Cloud), siendo este último un servicio web que proporciona capacidad informática en la nube segura y de tamaño modificable. Cada caso enfocaba distintas variedades de distribución de los llamados a la aplicación de forma que sus resultados presentaran si era económicamente viable migrar a Serverless.

De acuerdo a Alda y otros (2019), la distribución de tráfico (visitas) es la que tiene mayor impacto en los costes, aunque no debe tampoco menospreciarse el ahorro que brinda el apalancamiento de la infraestructura HTTP del proveedor Cloud, el de planificación y despliegue de alta disponibilidad que hace Lambda es muy económico.

## 2.2 Bases Teóricas

### 2.2.1 Arquitectura Web

Disciplina encargada de la organización de la información y contenidos de una web, se encarga de la jerarquía entre los elementos y la relación de estos (enlazados internos).



### 2.2.2 Arquitectura Serverless

El término fue utilizado por primera vez en el 2012 por Ken Fromm, pero no fue hasta el 2015 que se hizo popular, esto por dos servicios lanzados por Amazon entre el 2014 y el 2015 que son AWS Lambda y API Gateway. Así pues es un modelo de computación con el cual se crean y ejecutan aplicaciones y servicios que presta como ventaja no tener que administrar la infraestructura, uso eficiente de los recursos de computación, es fácil de operar y ofrecen características integradas de escalado.

### 2.2.3 Arquitectura Monolithic

Es en la que el software se estructura de manera que todos los aspectos funcionales del mismo quedan acoplados, creando entornos de trabajo estratificados pero con poca flexibilidad, toda la información de sistema está alojada en un único servidor de forma estable. Pero lo que termina por resultar una desventaja es que su código único hace difícil identificar y solucionar problemas concretos, además su desarrollador debe entender la totalidad del mismo, pues un error pequeño puede comprometer el código en su conjunto. Su ventaja principal era el costo de desarrollo, frente a la tecnología de micro servicios, pero poco a poco esta ventaja se ha ido disminuyendo, por lo complicado que es su crecimiento.

### 2.2.4 Servicios Web

Según Diego Lázaro (2018) Es un método de comunicación entre dos aparatos electrónicos en una red, permitiendo que diferentes sistemas operativos y aplicaciones se comuniquen entre sí por mensajes estandarizados con un mismo protocolo, sin la intervención humana.



### 2.2.5 Rest o Representational State Transfer (Transferencia de Estado Representacional)

Es un estilo de arquitectura software, el término fue utilizado por primera vez en el año 2000 por Roy Fielding, y actualmente tiene ciertas premisas:

-Un protocolo cliente/servidor sin estado, pues cada mensaje http contiene toda la información necesaria para comprender la petición.

-Operaciones bien definidas las más importantes son GET: Se usara para solicitar consultar a los recursos, POST: Se usará para insertar nuevos recursos, PUT: Se

usará para actualizar recursos y DELETE: Se usará para borrar recursos, estas se requieren para la persistencia de datos.

-Sintaxis Universal: para identificar los recursos y direccionados únicamente a través de su URI

-Uso de hipermedios como html y xml sin requerir registros y otra infraestructura adicional.

### 2.2.6 Load Testing

Es una prueba de software que se utiliza para entender el comportamiento de una aplicación en situaciones normales y en condiciones extremas cuando múltiples usuarios la usan al mismo tiempo, identifica si la infraestructura de alojamiento web en la aplicación es suficiente, establece cuantos llamados o usos en simultaneo puede manejar la aplicación en términos de hardware, capacidad de red y cuantos usuarios pueden acceder a la aplicación, ayuda identificando la capacidad máxima de operación, así como cualquier cuello de botella que cause degradación de un elemento como la memoria consumida, la red o la respuesta en el tiempo de la banda ancha.



El factor principal de medida es el desempeño más no la estabilidad, ni si es robusta, solo se mide su respuesta a múltiples llamados.

### 2.2.7 Aplicaciones Web

Software que se codifica en un lenguaje soportado por navegadores web y que es ejecutado por los mismos, que en general no requieren instalación en el dispositivo utilizado, que guardan de manera permanente la información que manejan en

servidores web o de internet, que envían a los usuarios los datos que necesitan dejando solo copias temporales en los equipos en uso.

### 2.2.8 Java

Lenguaje de programación orientado a objetos, ejecutable en cualquier contexto y ambiente lo que la hace adaptable y es además una plataforma informática, fue comercializada por Sun Microsystems en 1995.

### 2.2.9 Java Script

Es un lenguaje de programación creado originalmente por Brendan Eich de Netscape, orientado a los objetos, que es ligero e interpretado, que es altamente estructurado e imperativo, pero que su tipado es dinámico. Para 1997 se propuso como estándar en la European Computer Manufacturers y poco después en ISO (International Organization for Standardization). Aunque principalmente se utiliza del lado del cliente también opera en el lado del servidor.



### 2.2.10 Node JS

Es un entorno de ejecución para Java Script creado por Ryan Dahl multiplataforma, de código abierto, que es útil en la creación de programas de red altamente escalables, que se ejecuta en el servidor y no en el navegador, que proporciona una capa para programación asíncrona (llamas que podrán ser cumplidas ahora o en un futuro, asignadas en cualquier momento de ejecución del programa, en la espera que se complete una operación) y otros módulos fundamentales y que está construido con Java Script V8 de Chromes.

### 2.2.11 Spring-Boot

Es una infraestructura ligera que desarrollada en Java busca simplificar el proceso de programación de aplicaciones empresariales aún más que Spring pues reduce el despliegue en servidor y el seleccionar jars con maven, horrando líneas de código, haciendo más eficaz el proceso y evitando tareas repetitivas.

### 2.2.12 AWS (Amazon Web Services)

Es una colección de servicios de computación ofrecidos en amazon.com, que están en una plataforma pública (la nube).

### 2.2.13 AWS Lambda

Es un servicio web de Amazon que sin aprovisionar servidores ejecuta un código proporcionado en respuesta a eventos, administrando automáticamente los recursos informáticos y en el cual se paga por el tiempo de cómputo que se consume.



### 2.2.14 Apache JMeter

Es un servicio informático creado en Java, que se usa como herramienta para realizar pruebas automáticas de carga, que miden y analizan rendimientos de servicios en especial en aplicaciones web.

### 2.2.15 Apache Tomcat

Es un servidor web, en entorno libre, desarrollado con Java, funciona como contenedor de servlets que es un lenguaje de programación utilizada para ampliar las capacidades de un servidor que compila en JsPs (Java Server Pages) que es una tecnología para crear páginas web dinámicas y las transforma en servlets.

### 2.2.16 Json

Es el acrónimo de JavaScript Object Notation (Notación de Objeto JavaScript), es además un formato estándar para intercambio de datos, que se basa en pares atributo-valor, delimitado por llaves que es ligero y rápido.

### 2.2.17 Algoritmo de Dijkstra

Descrito por primera vez por Edsger Dijkstra en 1959, es un algoritmo para la determinación del camino más corto, dado un vértice origen hacia el resto de los vértices de un grafo que tiene pesos en cada arista.

# Capítulo 3 Modelado y Análisis de Requerimientos

## 3.1 Metodología

Para la producción de los Servicios Webs y la Batería de Pruebas a ser estudiados se usó la metodología de desarrollo evolutivo con entrega de prototipos. Esta metodología se adapta perfectamente en el desarrollo con tecnologías novedosas cuyo potencial aún no se explota a totalidad, dado que cada prototipo entregado representa en si una nueva exploración de las capacidades y beneficios de la tecnología en cuestión.

Es importante señalar que a pesar de que la arquitectura Monolithic es ampliamente conocida y documentada, Serverless por su parte cuenta con poco material para desarrolladores iniciados en sus conceptos, lo cual dificulta cada iteración de los prototipos entregados.



La metodología de desarrollo evolutivo con entregas de prototipos consta de seis etapas (Pressman, 2005):

1. Recolección y refinamiento de requisitos:

Para determinar la mejor estrategia y requisitos para comparar la arquitectura Serverless contra una arquitectura tradicional, se consultó con los representantes de la compañía BUCARES © quienes constituyen el cliente principal con interés en los resultados de las pruebas a aplicar.

2. Diseño rápido:

Se realiza un Análisis de los requerimientos recolectados y se define la especificación y modelo de negocio. Para cada servicio se definió de manera breve

y concisa un simple ‘work-flow’ o flujo de trabajo que deben cumplir cada servicio por separado y luego un esquema de pruebas para la sección final de load balance. Se proveen los recursos necesarios para documentar el proceso como diagrama de componentes, diagrama de casos de uso, entre otros gráficos requeridos.

3. Construcción de prototipos:

Se diseñan prototipos que representan los aspectos que conforman la estructura de cada servicio web, es decir, cada prototipo de los servicios debe englobar una parte de los requerimientos obtenidos en las etapas anteriores. Vale destacar que dada la naturaleza de la arquitectura Serverless, la cual representa un frente poco explorado y con documentación limitada, varias fases del prototipo del servicio Scadrian se limitaron a llegar a estados funcionales.

Para el servicio Scadrian se realizaron 4 prototipos funcionales; los 3 primeros se centraron en la configuración de un servicio Serverless en la plataforma lambda de AWS, el primero se centró en la creación del App base en el framework Serverless, el segundo la creación del stack o área de almacenamiento dentro de lamba y finalmente el despliegue del servicio base en Serverless. El último prototipo entregado contenía la implementación necesaria para la resolución del algoritmo de Dijkstra dentro del servicio ya desplegado.

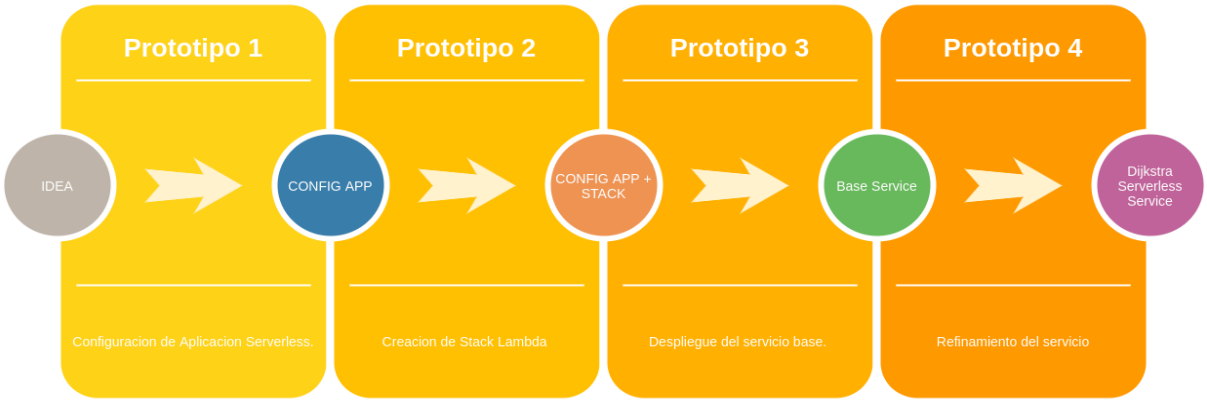



Figura 1: Diagrama Prototipo Evolutivo para el servicio Scadrian

Por su parte para el servicio Roshar que comprendía la misma funcionalidad pero en una arquitectura Monolithic, su proceso de prototipos fue más convencional, pasando de la primera etapa donde se generó un proyecto en spring-boot vació o "Boilerplate" usando la herramienta provista por Spring para la tarea "Spring initializr" con dependencias "spring web". Luego pasando por un segundo prototipo refinado con la implementación de Dijkstra el cual incluía la creación de modelos para la representación de las aristas y nodos, configuración del endpoint de cálculo y métodos utilitarios para los cálculos. Finalmente en la tercera etapa se crea el prototipo final que comprende el despliegue del prototipo anterior en un servidor privado, levantando un servidor de apache Tomcat para la tarea.

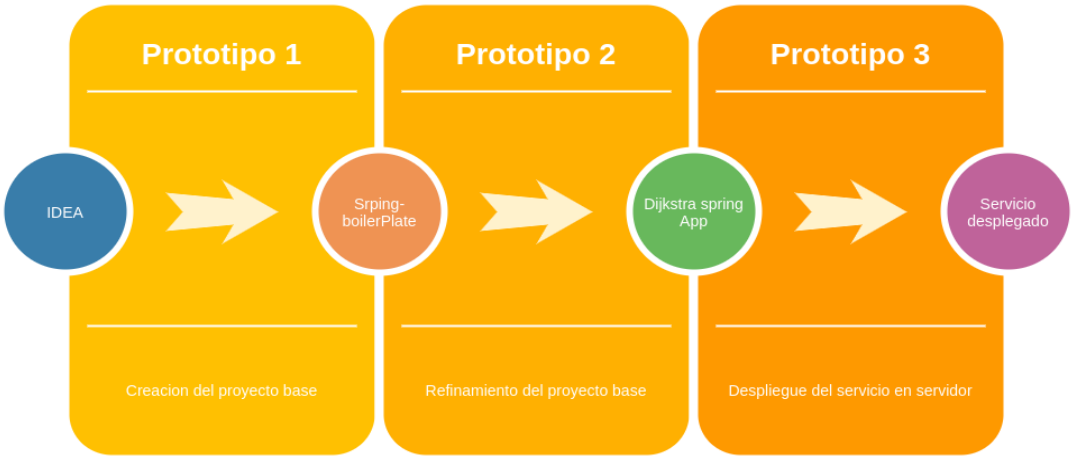



Figura 2: Diagrama Prototipo Evolutivo para el Servicio Roshar

4. Evaluación de prototipos por el cliente:

Cada prototipo creado y refinado es entregado al cliente para su evaluación, desde el primer prototipo. Este proceso de evaluación es constante dado a las reuniones Diarias realizadas con los usuarios, por lo tanto el refinamiento de los mismos es realizado de forma rápida y efectiva.

5. Refinamiento/Re-fabricación:

Una vez entregados los prototipos, con el objetivo de cumplir con los requerimientos establecidos, se realizan las correcciones, y se afinan los detalles necesarios, en esta etapa se sugieren mejoras y se aplican a los servicios implementados. Dada la comunicación diaria con el cliente este proceso es constante. Del proceso de refinamiento el punto más importante del servicio Scadrian bajo arquitectura Serverless, es la petición de limitar la memoria del servidor a 1GB. Requerimiento que dictaría la estructura de las pruebas finales.

6. Producto de ingeniería:

Se repiten las etapas 4 y 5 hasta lograr un producto que cumpla mínimamente con los requerimientos establecidos. La finalidad es lograr unos servicios robustos y de calidad que cumplan con las especificaciones, particularmente el requerimiento de limitar la memoria de Scadrian género como consecuencia repetir todos los prototipos y el proceso de desarrollo del servicio.

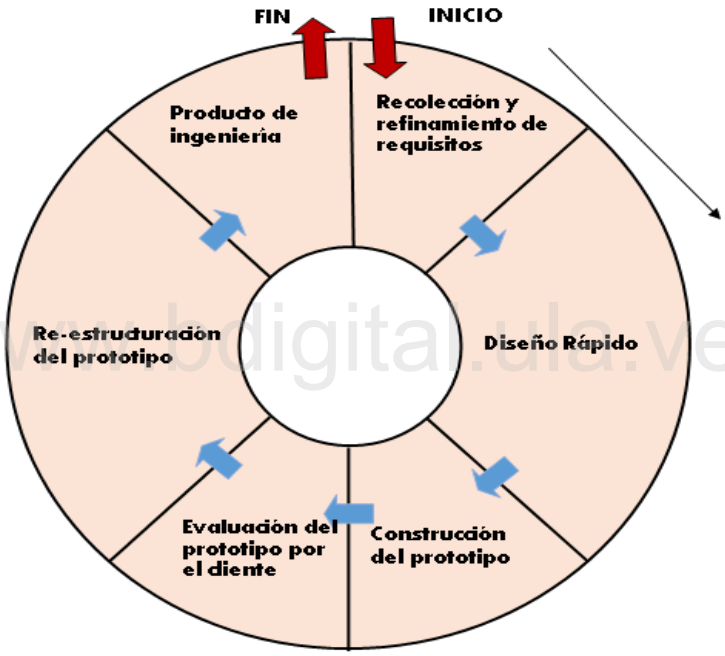



Figura 3: Modelo de Prototipo Evolutivo

## 3.2 Requerimientos del sistema

Del proceso de obtención y refinamiento de requisitos englobado en la primera sección de la metodología de desarrollo evolutivo se determinaron los siguientes requerimientos para cada Servicio:

### 3.2.1 Requerimientos funcionales

#### 3.2.1.1 Scadrian

- Despliegue dentro del Framework Serverless, permitiendo el monitoreo y configuración desde el Dashboard en línea.
- Calculo del camino más corto entre dos nodos de un grafo (algoritmo de Dijkstra).
- Proveer la capacidad de procesar cualquier grafo provisto en el cuerpo de la solicitud.

#### 3.2.1.2 Roshar

- Calculo del camino más corto entre dos nodos de un grafo (algoritmo de Dijkstra).
- Proveer la capacidad de procesar cualquier grafo provisto en el cuerpo de la solicitud.



#### 3.2.1.3 Pruebas de Carga

- Calculo de tiempos de respuesta para un numero configurable de solicitudes a los servicios Scadrian y Roshar.
- Generación de una tabla o reporte global de las pruebas.

### 3.2.2 Requerimientos no funcionales

#### 3.2.2.1 Scadrian

- Disponibilidad constante del servicio para ser accedido desde cualquier dispositivo.

#### 3.2.2.2 Roshar

- Disponibilidad constante del servicio para ser accedido desde cualquier dispositivo.

### 3.2.3 Requerimientos de calidad

Para los servicios y las pruebas de carga se definieron los siguientes requisitos de calidad:

- Robustez y estabilidad.
- Intuitividad y facilidad de uso.
- Disponibilidad de los recursos fuentes.



## 3.3 Modelado del Sistema

Para la representación gráfica de los servicios se utilizó el lenguaje estándar de modelado UML para ofrecer una vista general del sistema.

### 3.3.1 Actores

- Usuario: Se identifica como usuario al ente que interactúa con los servicios o con la batería de pruebas de carga, para el caso concreto de los servicios, la batería de pruebas pasa a ser el usuario quien recibe y procesa las respuestas.

### 3.3.2 Entidades de Negocio

- Grafo: es el objeto dinámico enviado en formato JSON sobre el cual se realizan los cálculos.

### 3.3.3 Diagrama de Clases

La comparación en los tiempos de respuesta entre las arquitecturas serverless y monolithic requiere un problema de cálculo común entre ambos servicios. Se plantea el algoritmo del menor camino entre dos nodos de un grafo (Algoritmo de Dijkstra). Las clases requeridas para abstraer el concepto de un grafo y realizar las operaciones necesarias para el cálculo de Dijkstra se plantearon de la siguiente forma:

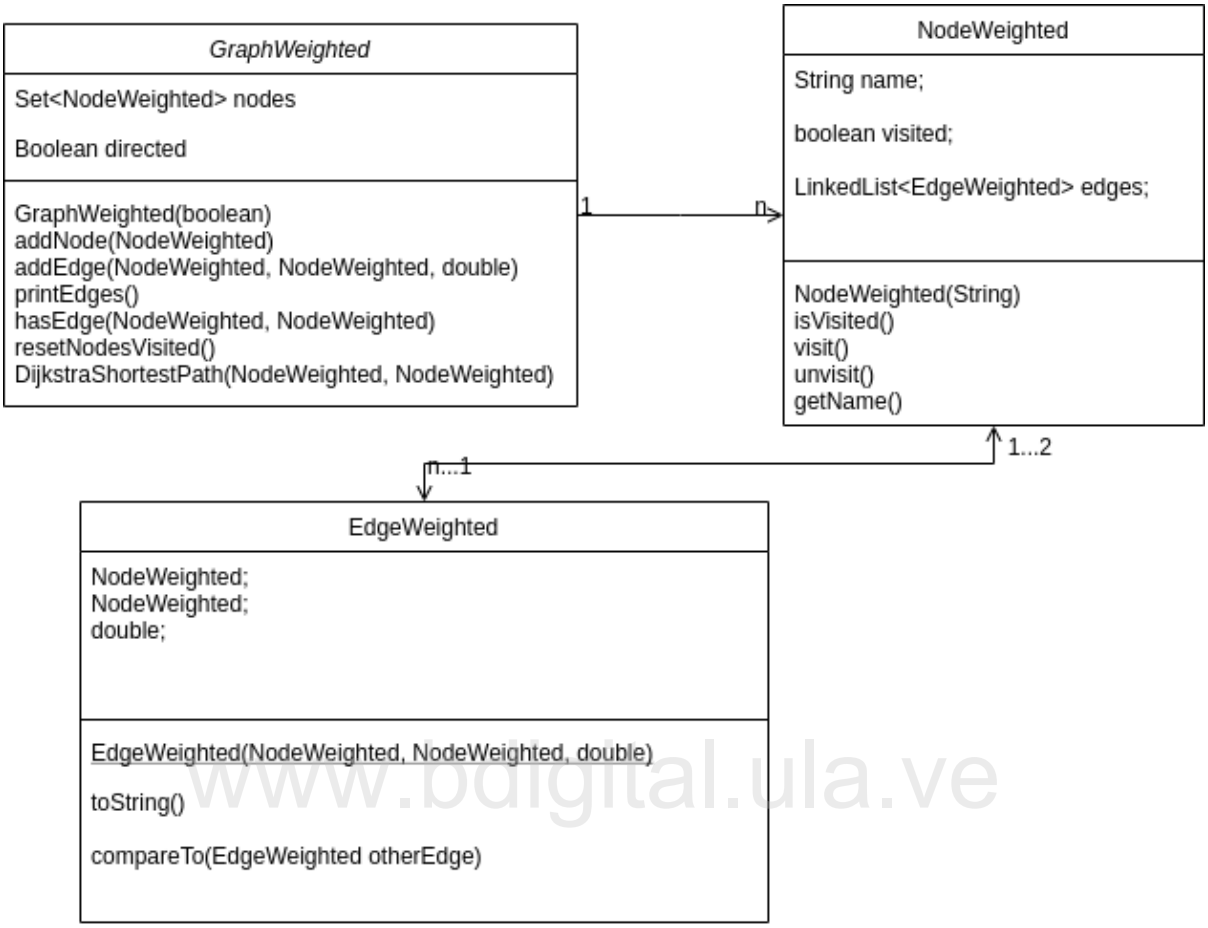



Figura 4: Diagrama de Clases

### 3.3.4 Diagramas de casos de uso

Aun cuando la implementación de lógica de negocio es compartida, el caso de uso similar y la interacción con el usuario en un solo sentido, la distribución y procesamiento del problema se efectúa de manera diferente en ambos casos.

Para el caso de serverless, es AWS quien se encarga de almacenar la lógica de negocio, recibir las peticiones webs, procesar el cálculo y retornar la respuesta en el formato apropiado.

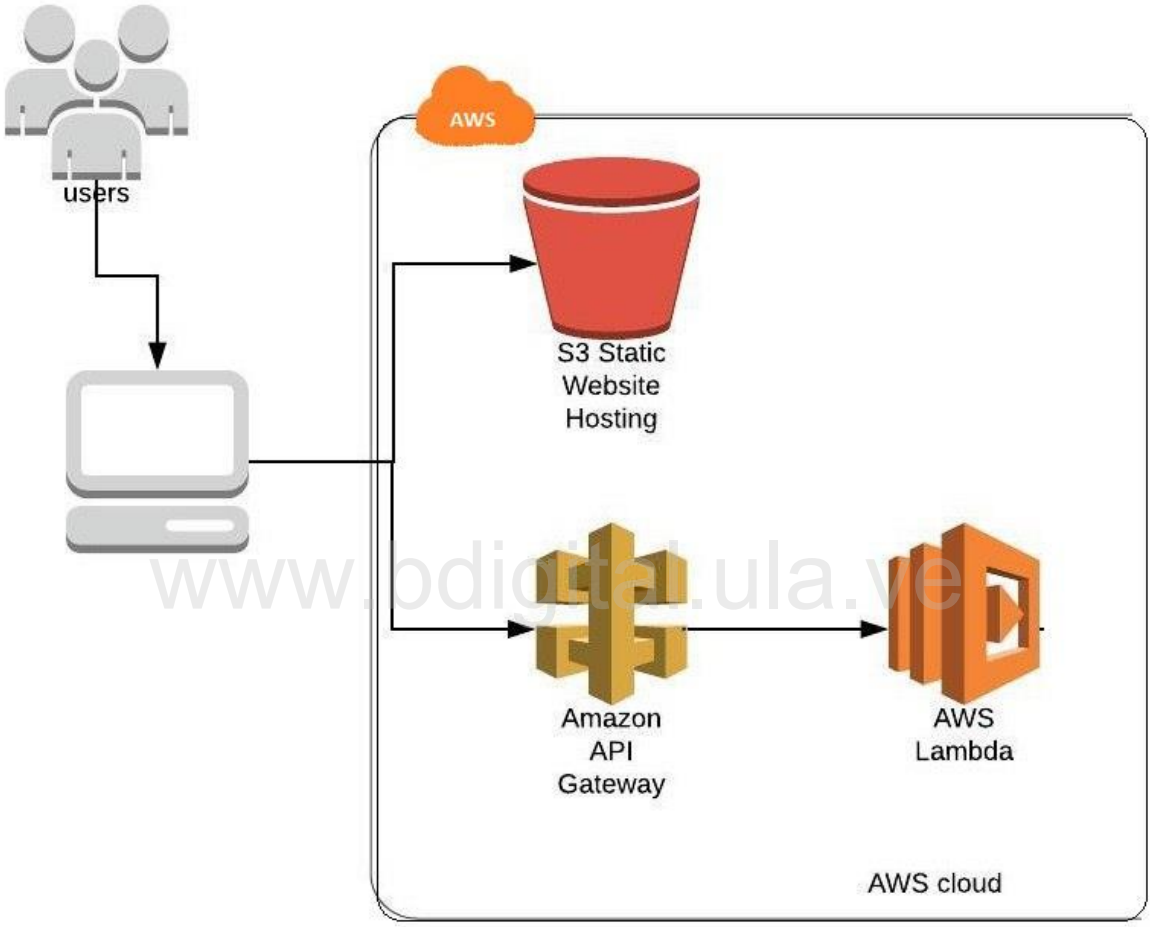





Figura 5: Diagrama de Caso de Uso Servicio Scadrian

Por su parte la arquitectura monolithic requiere más configuración y centraliza todos los recursos en un único servidor de Apache Tomcat.

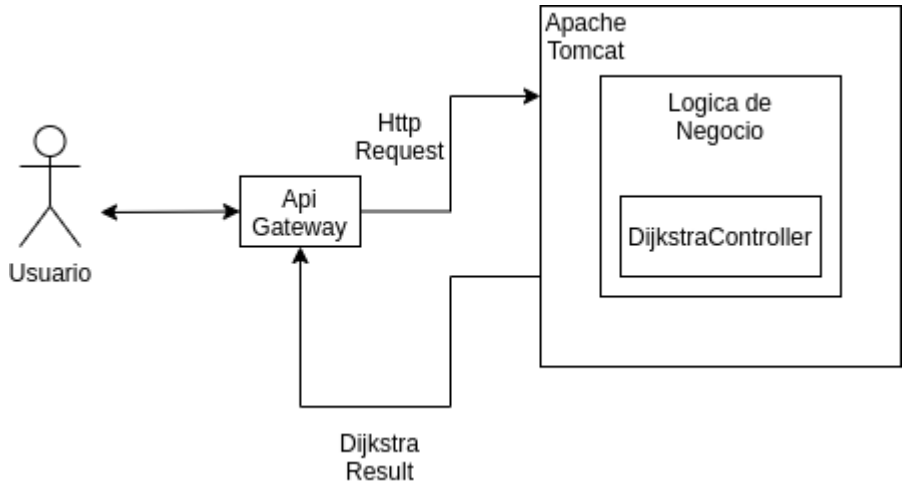



Figura 6: Diagrama de Caso de Uso Servicio Roshar

Finalmente el flujo completo del trabajo se centra en la batería de pruebas de carga invocando ambos servicios en simultáneo y generando un informe final de pruebas.



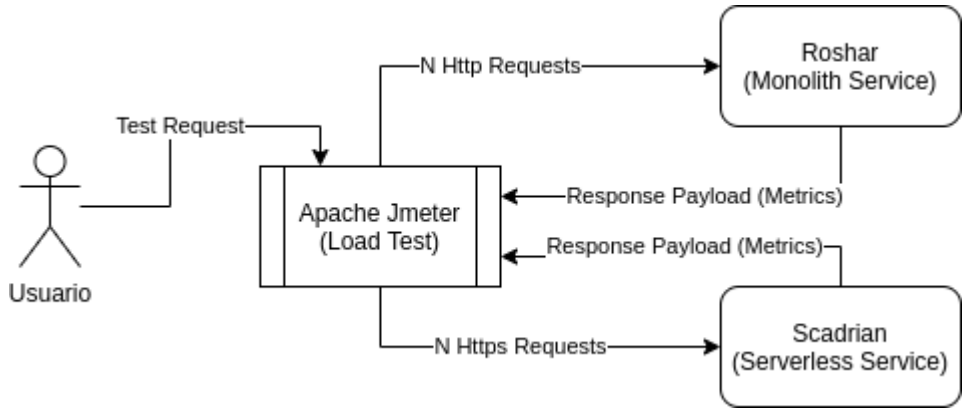



Figura 7: Diagrama de Caso de Uso Flujo Completo

# Capítulo 4 Diseño y Arquitectura

## 4.1 Justificación y diseño

Dada la naturaleza ligera de los servicios web implementados y el hecho que ambos están enmarcados por sus propios Frameworks o modelos de desarrollo (Serverless y Spring-boot respectivamente), los estándares de desarrollos de ambos son bastante rígidos y predefinidos, Brindando todas las bondades de los desarrollos bajo arquitecturas webs, entre los cuales se encuentran la practicidad del acceso mediante un navegador web o cliente REST, capacidad de atender múltiples peticiones en simultaneo, facilidad de actualización e independencia del sistema operativo y finalmente la carente necesidad de distribuir e instalar cualquier tipo de software en máquinas de terceros.



Por su parte, Apache JMeter también brinda un estándar para la creación de pruebas de carga, permitiendo la configuración de las mismas de manera fácil, rápida y centralizada.

La necesidad de avances continuos y prototipos funcionales definidos en la metodología llevada a cabo demanda contar con herramientas de desarrollo de software que proporcionen al programador mecanismos de rápida implementación para de esta forma reducir el tiempo de desarrollo.

A continuación se exponen las herramientas usadas para el desarrollo de los servicios webs Roshar y Scadrian, así como la batería de pruebas de carga:

- Para el servicio Scadrian bajo arquitectura Serverless, se empleó para la lógica interna el lenguaje de programación JavaScript soportado en el Framework Node.js, siendo este perfecto para la creacion de servicios ligeros y rápidos.
- Para el despliegue del servicio Scadrian se empleó el Framework Serverless utilizando los recursos de AWS Lambda para alojar el servidor.
- Para el Servicio Roshar bajo arquitectura Monolithic, se empleó el Framework Spring-Boot bajo el lenguaje de programación JAVA. Quien constituye una opción robusta pero pre configurada para un desarrollo rápido y eficaz.
- Para el despliegue del servicio Roshar se utilizó un servidor de Apache Tomcat embebido en el Framework Spring-Boot
- El gestor de pruebas de carga empleado fue Apache JMeter, el cual permite realizar una batería de pruebas a cualquier servicio web abierto, sin limitaciones de lenguajes o arquitecturas.



## 4.2 Diagrama de Despliegue

El paradigma cliente-servidor permite que un software cliente en este caso la batería de pruebas de Apache JMeter pueda comunicarse con el servidor que realiza los cálculos, esto es cierto para ambas arquitecturas, de forma simultanea JMeter se encarga bajo petición del usuario final de realizar las peticiones HTTP Y HTTPS a los diferentes servicios y de procesar las respuestas en un formato más visual y entendible para su estudio.

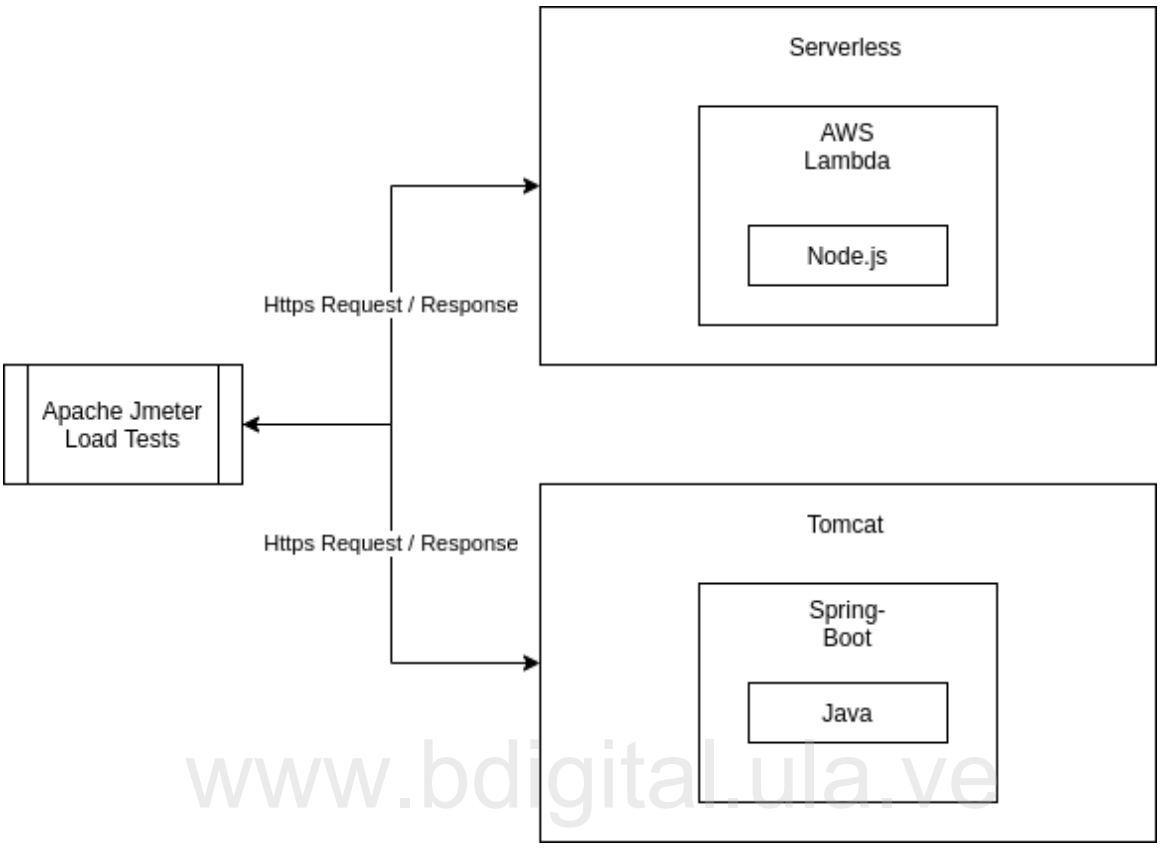





Figura 8: Diagrama de Despliegue

## 4.3 Arquitectura de software

Para comprender la arquitectura global que abarca ambos servicios y la batería de pruebas, debemos estudiar cada arquitectura y sus bondades.

### 4.3.1 Scadrian (Serverless)

El servicio Scadrian sigue el patrón arquitectónico Serverless, el cual mediante la configuración de un único archivo de configuraciones en formato YML se encarga

del despliegue completo del servidor; también por su parte configura automáticamente los certificados de seguridad brindando una conexión Https.

La estructura del servicio comprende dos componentes: el primero, el archivo de configuración que gobierna toda la configuración de Lambda y el Framework Serverless; el segundo los archivos contenedores de la lógica a ejecutar, estos son configurados para cada Endpoint en el archivo de configuración.

Para mantener un patrón de desarrollo ordenado se puede aplicar un modelo MVC (modelo, vista, controlador), dado que finalmente el servicio a desplegar es una aplicación de node.js con todas las bondades que brinda el Framework de desarrollo, para el caso de estudio, dada la ligereza del servicio un único controlador es suficiente para cubrir la lógica necesaria.

Scadrian ofrece un único endpoint abierto de tipo POST para el cálculo del camino más corto, dado un grafo cualquiera provisto en el cuerpo de la solicitud en formato JSON, siguiendo el siguiente patrón:



```
{
        "graph": {
                  "start": {"string": int, … },
                  "string": {"string": int ...},
                 ...
                 "finish": {“string”: int, ...}
        }
}
```

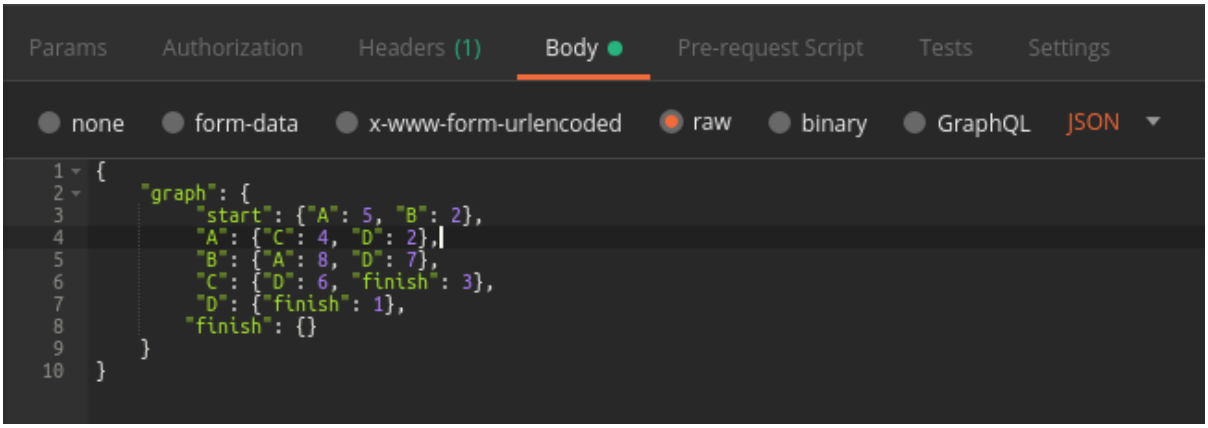



Figura 9: Cuerpo de Solicitud Servicio Scandrian

La solicitud expedida por el usuario, sea un cliente Rest o la batería de pruebas de JMeter, va directamente al servicio desplegado en AWS Lambda quien maneja la puerta de conexión, el procesamiento y responde con el siguiente cuerpo para un grafo conexo con solución alcanzable:



```
{
  "message": "Dijkstra successfully calculated",
  "Result": {
    "distance": int,
    "path": [
      string,
          ...
    ]
  }
}
```

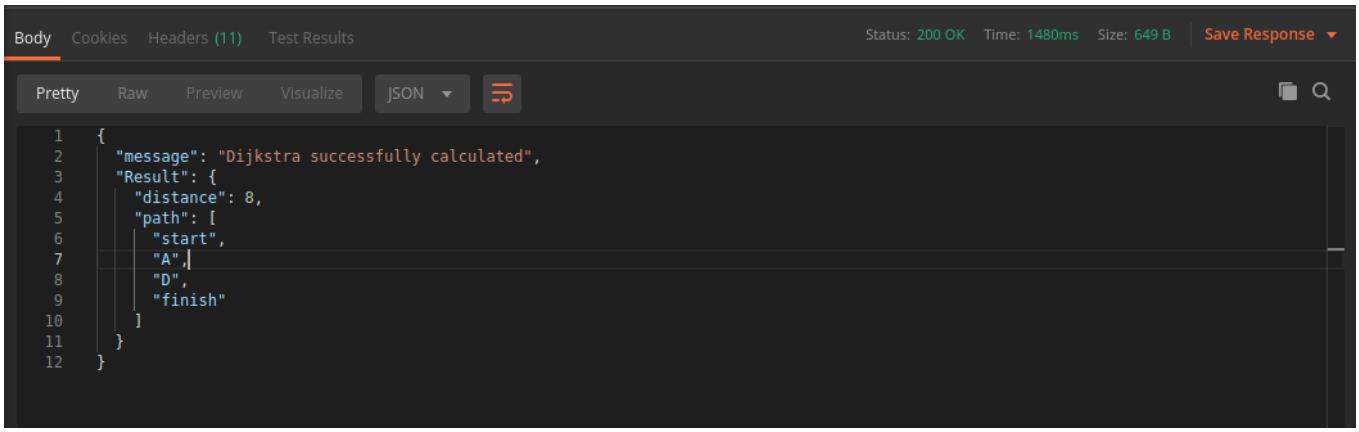



Figura 10: Respuesta Positiva Servicio Scadrian

### 4.3.2 Roshar (Monolithic)

Por su parte el servicio Roshar emplea una arquitectura tradicional, desplegándose en un servidor de apache Tomcat. Todas las configuraciones de despliegue están enmarcadas en los archivos de configuración de la aplicación, distribuidos en múltiples puntos del cuerpo de la misma. El más relevante es el archivo de propiedades localizado en la carpeta de “recursos” donde se localizan entre otras las configuraciones TLS del servidor. Vale acotar que Roshar emplea el protocolo HTTP no securizado, dado que Spring-Boot no provee las configuraciones necesarias de base para utilizar el protocolo seguro HTTPS, dicha configuración requiere la generación de un certificado de seguridad local, y la inclusión del mismo a los registros de seguridad del lado del servidor.

Roshar emplea el modelo de desarrollo MVC (modelo, vista, controlador) separando la lógica de puntos de entrada al servicio (Endpoints) y el modelado del grafo a calcular. Por su parte carece de vistas por no ser un servicio con interfaz de usuario, Roshar se limita objetivo para el cual fue creado, calcular el camino más corto entre

dos nodos para un grafo provisto en el cuerpo de la solicitud, el cual posee el mismo formato del servicio Scadrian:

```
{
        "graph": {
                "start": {"string": int, … },
                "string": {"string": int ...},
                ...
                "finish": {“string”: int, ...}
        }
}
```




Figura 11: Cuerpo de Solicitud Servicio Roshar

La solicitud expedida por el usuario, sea un cliente Rest o la batería de pruebas de JMeter, va directamente al servicio desplegado en Tomcat quien maneja la puerta de conexión, internamente el despliegue en Spring-Boot en módulos denominados

Beans realizan el procesamiento y responde con el siguiente cuerpo para un grafo conexo con solución alcanzable:

```
{
   "code": 0,
   "data": {
      "message": "The path with the smallest weight between start and finish is:",
      "result": {
        "distance":  Double,
         "path": [
             string,
                 ....
                ]
       }
   },
  "message": "Dijkstra successfully calculated"
}
```

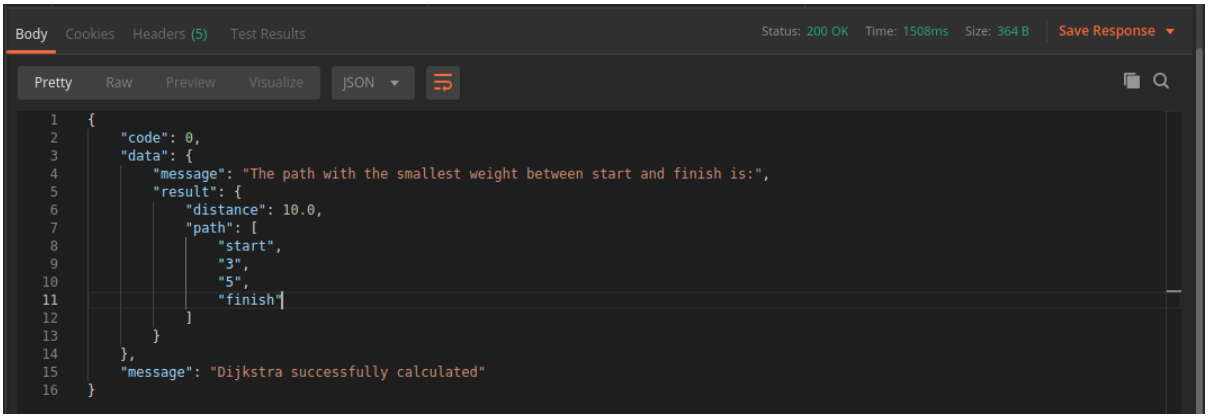


Figura 12: Respuesta Positiva Servicio Roshar

Aun cuando el formato de la respuesta varia, la información devuelta por Roshar en comparación con Scadrian es en esencia la misma.

### 4.3.3 Load Test (JMeter)

EL componente encargado de realizar los llamados Rest sobre los servicios desplegados en sus arquitecturas y con sus características, pero finalmente disponibles de manera similar mediante los protocolos HTTP Y HTTPS, es la batería de pruebas de Apache JMeter.

La batería de pruebas consiste de dos pruebas bajo una misma configuración, cada una de ellas realiza una llamada similar, variando tanto el URL y el puerto de entrada, como el protocolo de conexión para cada caso específico.



La configuración de estas pruebas no es más complicado que el de cualquier cliente Rest, Apache JMeter ofrece una interfaz intuitiva para la construcción de modelos de pruebas para aplicaciones web de cualquier tipo.

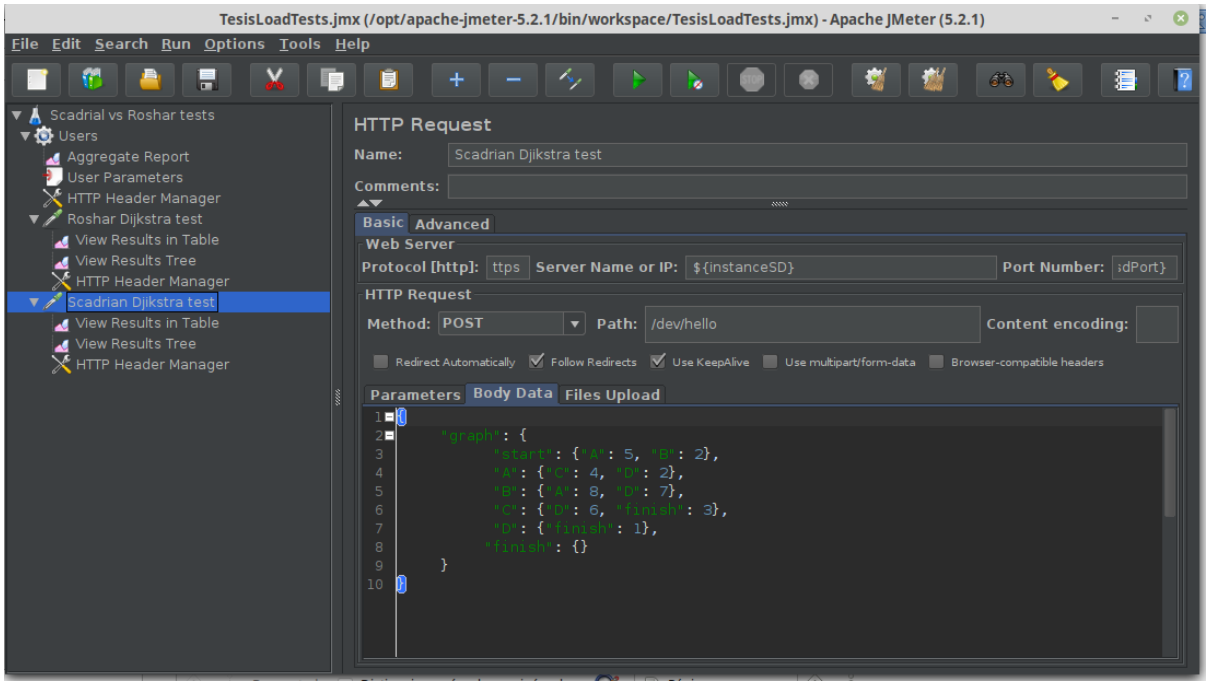



Figura 13: Configuración Prueba de Carga para el Servicio Scadrian



Una vez invocadas las pruebas desde la interfaz de JMeter, se realizaran las solicitudes a los servicios y se recolectara en la sección “Aggregated Report” las métricas generales de las consultas. Por su parte cada prueba cuenta con sus propias tablas de resultados en caso de necesitar información más detallada de las pruebas.

El volumen de pruebas puede ser fácilmente controlado en la sección “Thread Group” asociado al identificador “Users”. Aquí se localizan los parámetros que controlan tanto la cantidad de usuarios en simultáneo que se simularan accediendo al recurso de forma concurrente, así como la cantidad de solicitudes o “loops” que realizara cada usuario creado sobre los servicios.

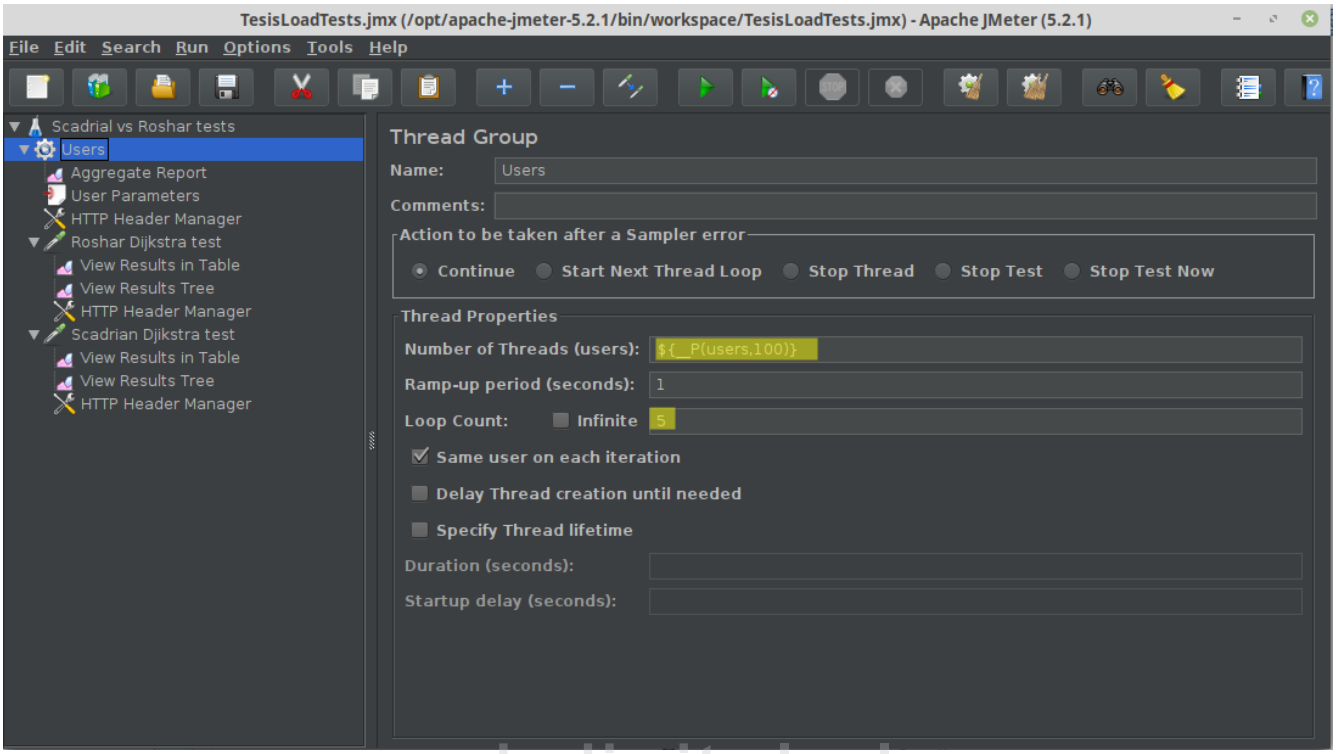





Figura 14: Configuración de Thread Group Batería de Pruebas

# Capítulo 5. Especificaciones, desarrollo, implementación y resultados.

## 5.1 Estructura Spring-Boot

Spring-boot ofrece la herramienta “Initializr” bajo el dominio “https://start.spring.io/ “. Siguiendo los pasos descritos en el portal web, obtendremos un proyecto base en Spring que permite su despliegue local en Apache Tomcat que viene embebido en la estructura base. A partir de este proyecto base se crearon las clases pertinentes bajo el modelo MVC (Modelo, Vista, Controlador) para la resolución del algoritmo de Dijkstra.

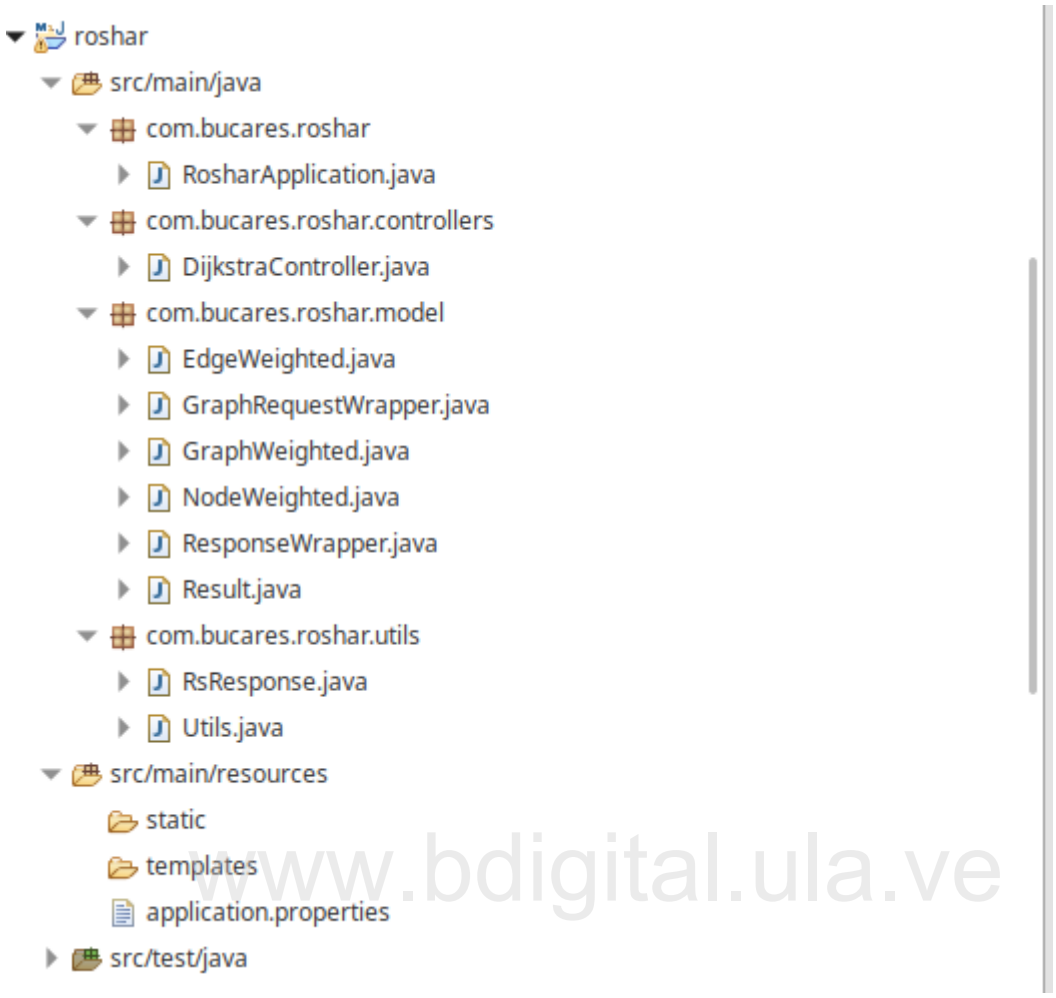



Figura 15: Vista de Proyecto Servicio Roshar parte 1


Figura 16: Vista de Proyecto Servicio Roshar parte 2

Los archivos de más relevancia dentro de la estructura de Roshar son los siguientes:

- /com.bucares.roshar: contiene la clase principal del proyecto desde la cual inicia el despliegue del servicio.

- /com.bucares.roshar.controllers: en este paquete se encuentra un único controlador, el cual es el servidor del punto de acceso (Endpoint) que ejecuta la lógica asociada al problema a resolver. Es la entrada del usuario a la lógica.

- /com.bucares.roshar.model: es el paquete encargado de contener las definiciones de los objetos que abstraen las entidades involucradas en el cálculo, Java es un lenguaje orientado a objeto por lo cual la abstracción de las entidades como método organizativo para la resolución de problemas es la norma de desarrollo. Aquí podemos encontrar las clases que abstraen un nodo, una arista, el grafo, así como las entidades o envoltorios de respuestas retornadas al cliente una vez realizado el cálculo.



- src/main/resources: el directorio resources contiene los recursos necesarios para el funcionamiento del servicio, el archivo de más importancia es el archivo de propiedades que rige la configuración global del mismo.

- pom.xml: el archivo pom.xml es de suma importancia en los servicios y aplicaciones desarrolladas con spring-boot, pues es el archivo de configuración de dependencias, plugings y servidor de información al archivo

de manifiesto de la aplicación. También puede regir el comportamiento del servicio a la hora de empaquetar, levantar, correr pruebas entre otros procesos de control sobre el servicio. Para el caso de Roshar es un manejador de dependencias.

## 5.2 Estructura Serverless

Serverless por su parte soluciona de manera interna todo el manejo del servidor, por lo cual solamente se requiere el desarrollo del cómputo para la resolución de Dijkstra. Es en el dashboard de Serverless donde se apreciara el despliegue completo de la aplicación.



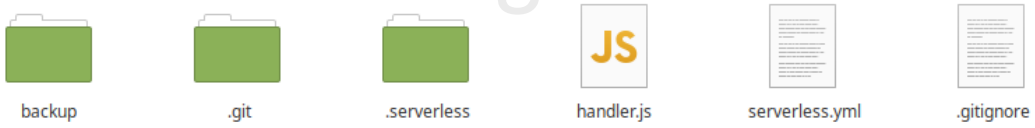



Figura 17: Estructura de Archivos Servicio Scadrian

De la estructura del servicio Scadrian los archivos más importantes son los siguientes:

- handler.js: es el archivo contenedor de la lógica para la solución del problema de Dijkstra, dada la facilidad de resolver este algoritmo en JavaScript no se requirieron clases utilitarias.

- Serverless.yml: es el archivo de configuración de Serverless, el cual asocia la lógica contenida en el archivo handler.js con el endpoint que sirve de entrada al servicio. De igual forma contiene las configuraciones de memoria y recursos máximos con los que puede contar el servidor.

## 5.3 Serverless Dashboard

Estudiando la estructura del servicio Scadrian se observa la facilidad en la configuración de un proyecto bajo esta arquitectura, sin embargo es en el dashboard de Serverless donde podemos obtener un control y una mejor idea de las configuraciones que se administran desde el archivo con formato yml.



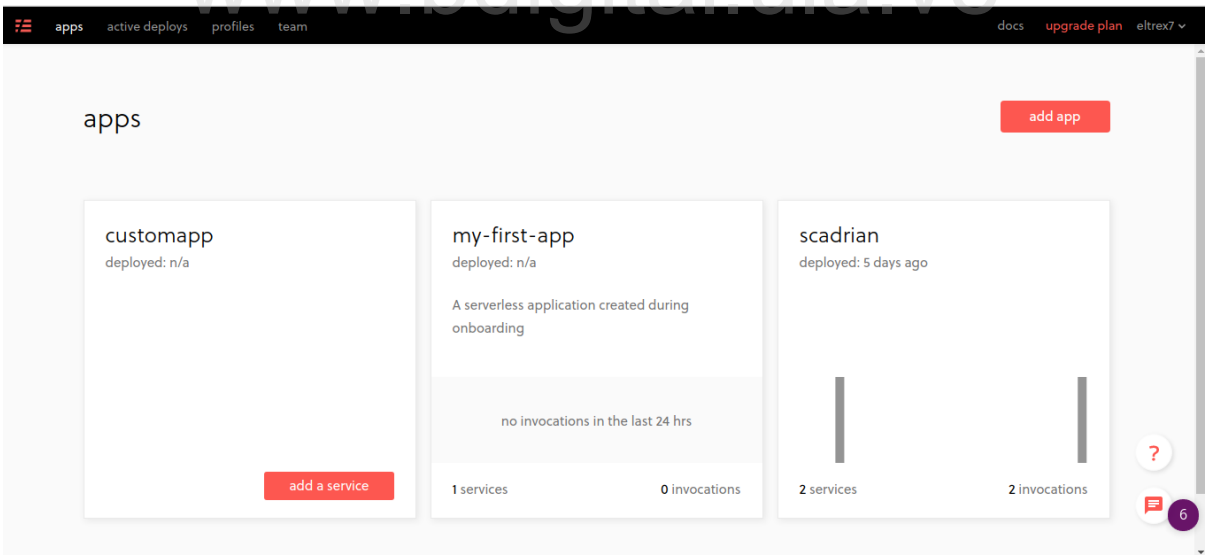



Figura 18: Vista de Aplicaciones Dashboard Serverless

El Dashboard de Serverless ofrece información pertinente a las invocaciones sobre el servicio de Scadrian, entre la que encontramos, memoria empleada, numero de invocaciones, tiempos de respuestas, numero de errores entre otros.

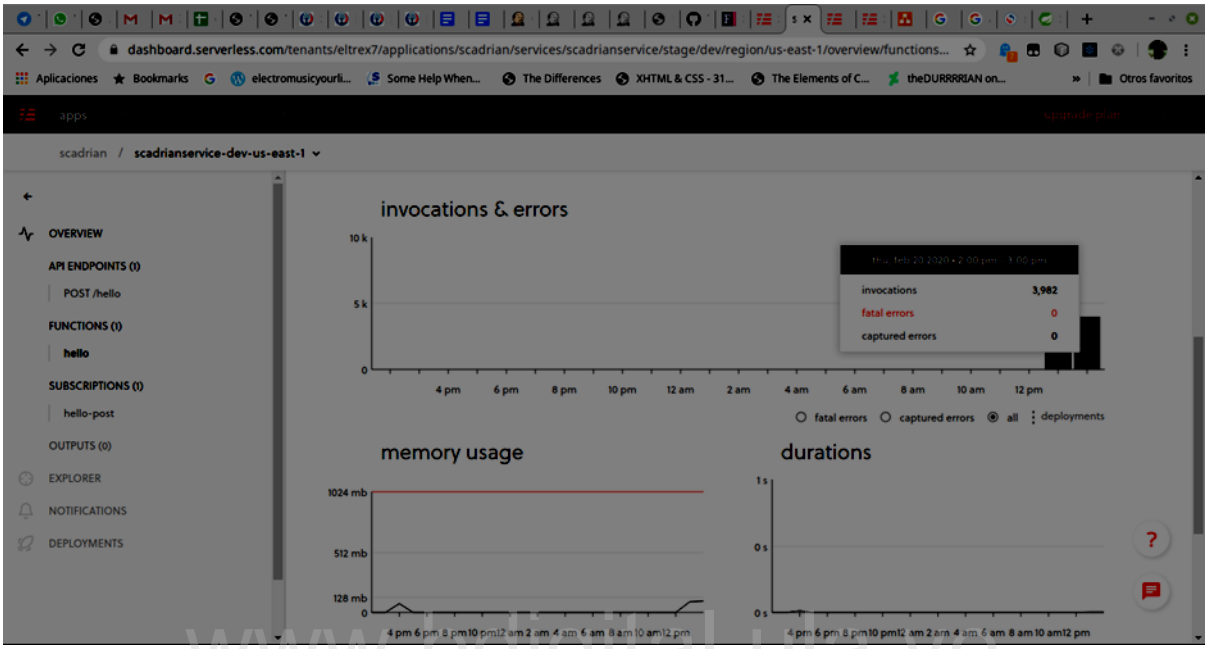





Figura 19: Vista de Funciones Dashboard Serverless

Constituye una herramienta fundamental no solamente para el monitoreo de las aplicaciones y servicios sino que también provee las herramientas para habilitar y deshabilitar los servicios al igual que alterar sus configuraciones.

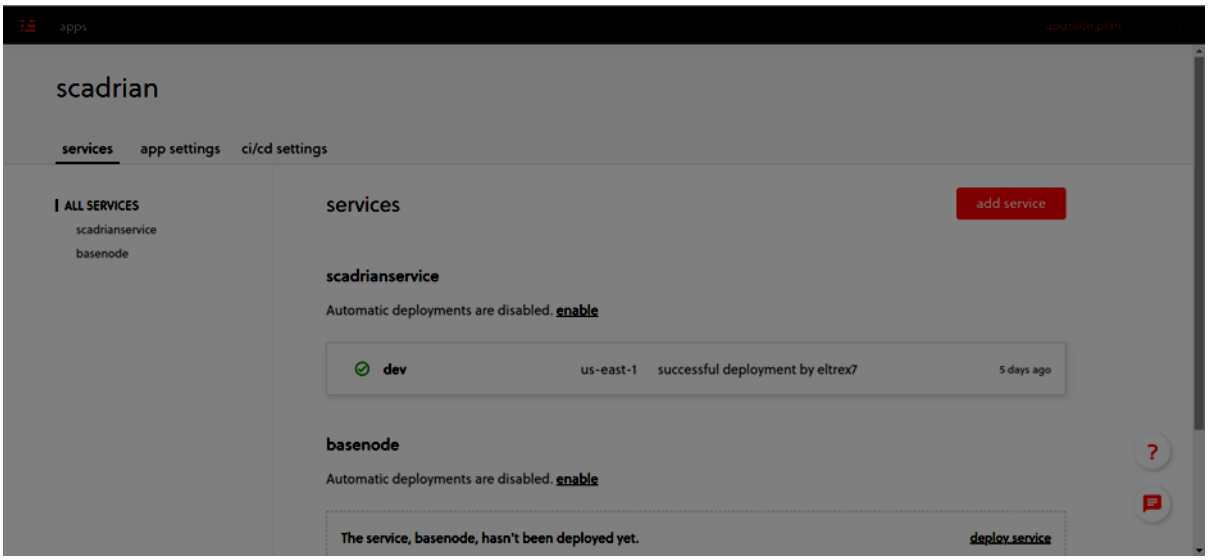



Figura 20: Vista de Servicio Scadrian Dashboard Serverless

## 5.4 Control de versiones



Para mantener un control sobre los prototipos entregados, así como respaldar el desarrollo y poseer la capacidad de compartir el código fuente de los servicios. Se configuraron dos repositorios de Github, uno para cada servicio.

Se eligió Github como plataforma para cargar las fuentes dado a su naturaleza gratuita y a las facilidades que ofrece para trabajar las configuraciones desde los mismos proyectos con Git.

## 5.5 Limitaciones

Uno de los parámetros bajo los cuales se configuro el servicio Scadrian en Serverless y que condiciona el cálculo realizado es el espacio de memoria con el que cuenta, el cual fue limitado a 1GB de espacio. El escalado del servicio sobre

esta capacidad implicaba costos adicionales en los cuales no se quiso incurrir para una prueba de concepto como la que simboliza el proyecto actual.

Para cumplir con esta limitante se diseñaron múltiples grafos a ser procesados, de los cuales podemos resaltar dos casos:

- El primer aproximamiento al problema se realizó sin una referencia por lo cual se trató de abarcar un grafo que generara una fuerte carga computacional, el mismo estaba distribuido de la siguiente forma:

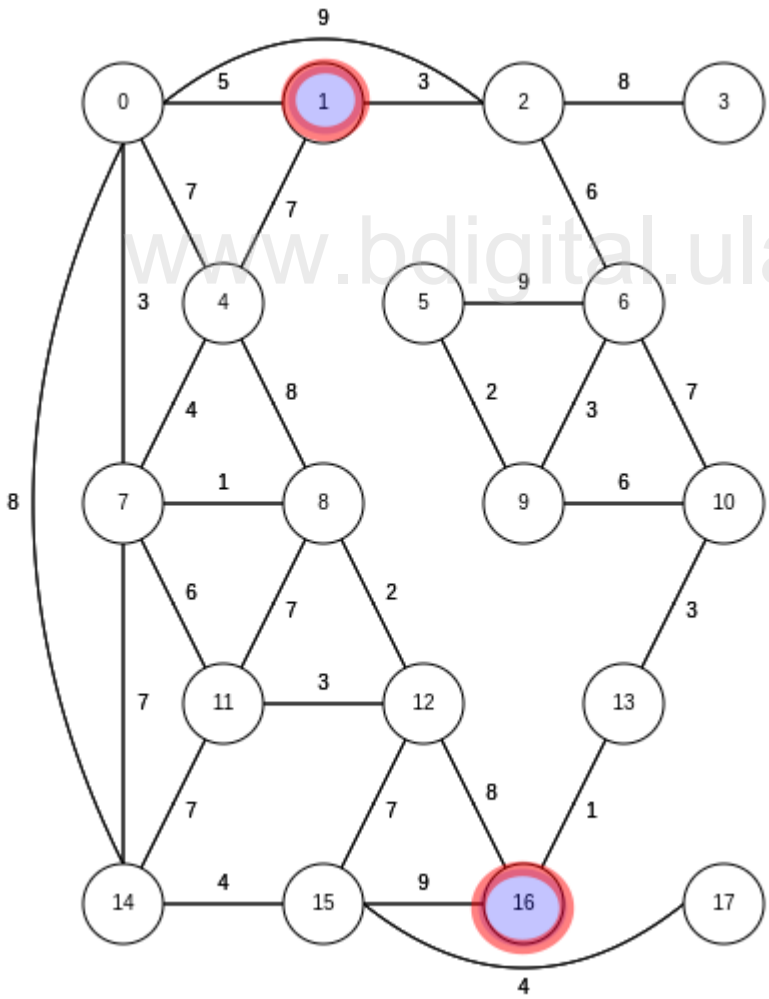



Figura 21: Grafo de Complejidad Alta

Para un grafo de esta magnitud la memoria del servicio se rebaso, generando errores en las solicitudes realizadas.

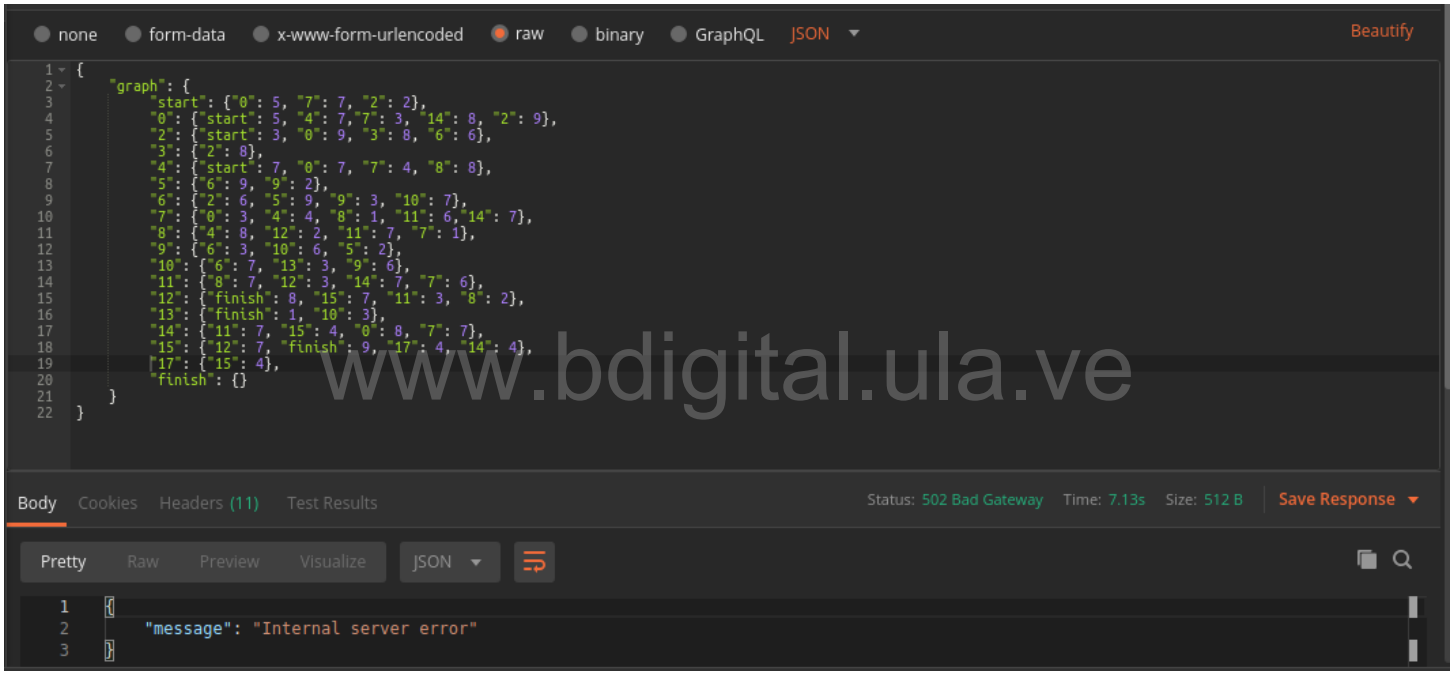



Figura 22: Cuerpo y Respuesta a Grafo de Alta Complejidad Servicio Scadrian

Este error corresponde al incremento súbito de memoria y la incapacidad de escalar la memoria del servicio al estar limitada.

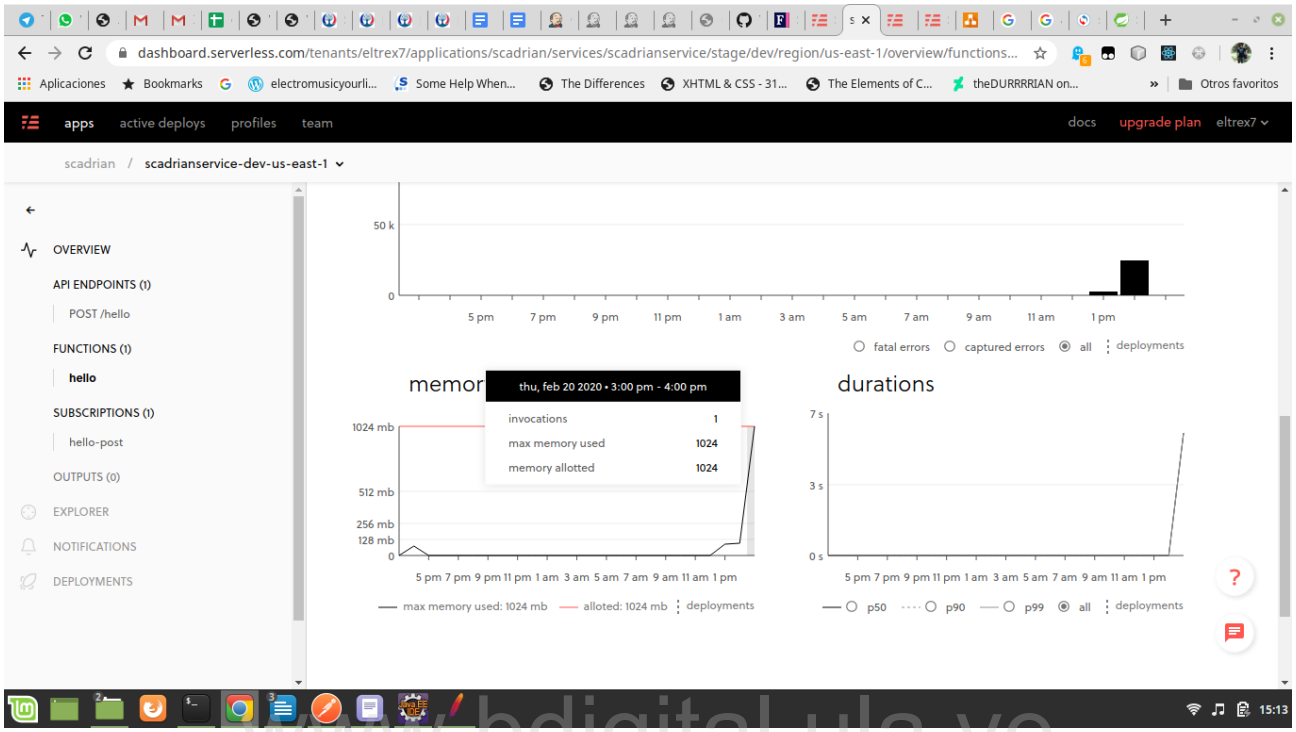


Figura 23: Vista de Función, Gráfico de Memoria Servicio Scadrian

Se fue refinando el grafo progresivamente disminuyendo el número de nodos y realizando un grafo unidireccional hasta lograr el grafo final empleado para realizar la comparación de resultados:

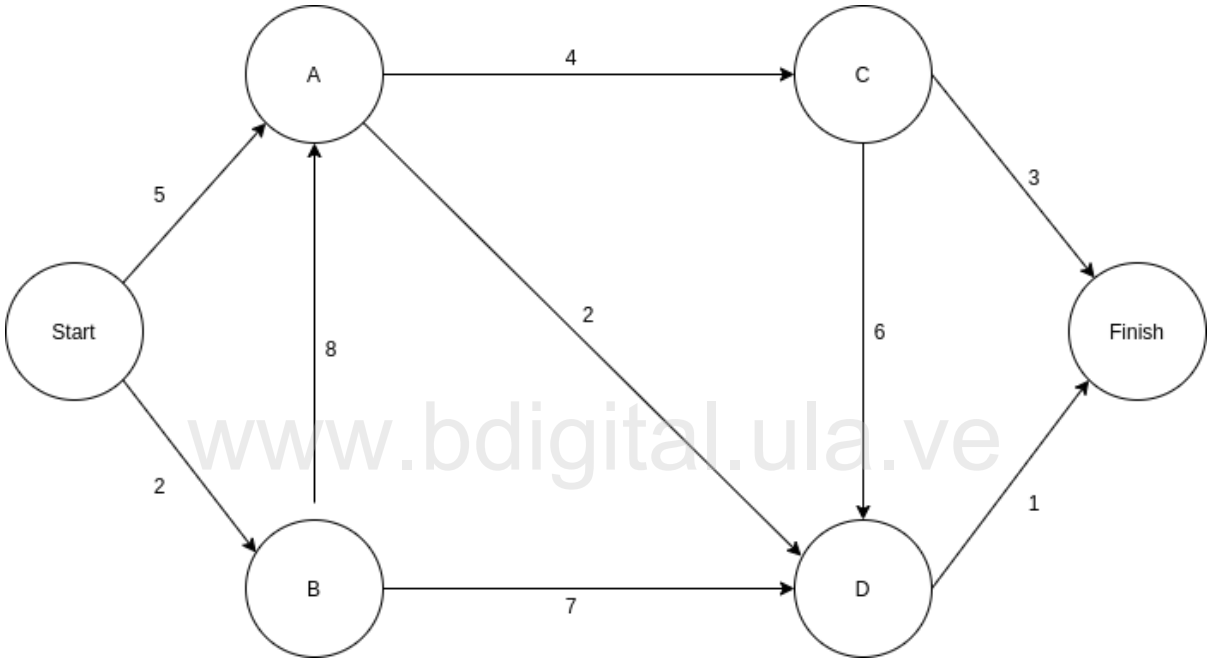





Figura 24: Grafo Final de Prueba

A pesar de ser un grafo mucho más sencillo del que ambiciosamente se diseñó en principio, constituye una referencia más que aceptable para generar la tabla de resultados. Vale recalcar que el problema en si no es tan importante como el estudio de las respuestas de las arquitecturas, en principio se requiere generar un cómputo

y una carga computacional suficientemente elevada que hiciera trabajar de forma equitativa a ambos servicios.

## 5.6 Resultados

Interpretar los resultados expuestos en esta sección requiere definir las métricas obtenidas en el informe final de Apache JMeter:

- Label: Etiqueta que aplica JMeter a cada prueba creada en base al nombre de la prueba en cuestión, nombre modificable en las configuraciones de cada módulo de prueba.
- Samples: número de peticiones realizadas al Endpoint configurado en la prueba
- Average: tiempo promedio de respuesta del servidor, medido en milisegundos.
- Median: es el tiempo medio entre todas las peticiones realizadas, medido en milisegundos.
- 90% line: es el tiempo referencial en el cual el 90% de la muestra de las peticiones se encuentran por debajo de este tiempo, medido en milisegundos.
- 95% line: es el tiempo referencial en el cual el 95% de la muestra de las peticiones se encuentran por debajo de este tiempo, medido en milisegundos.

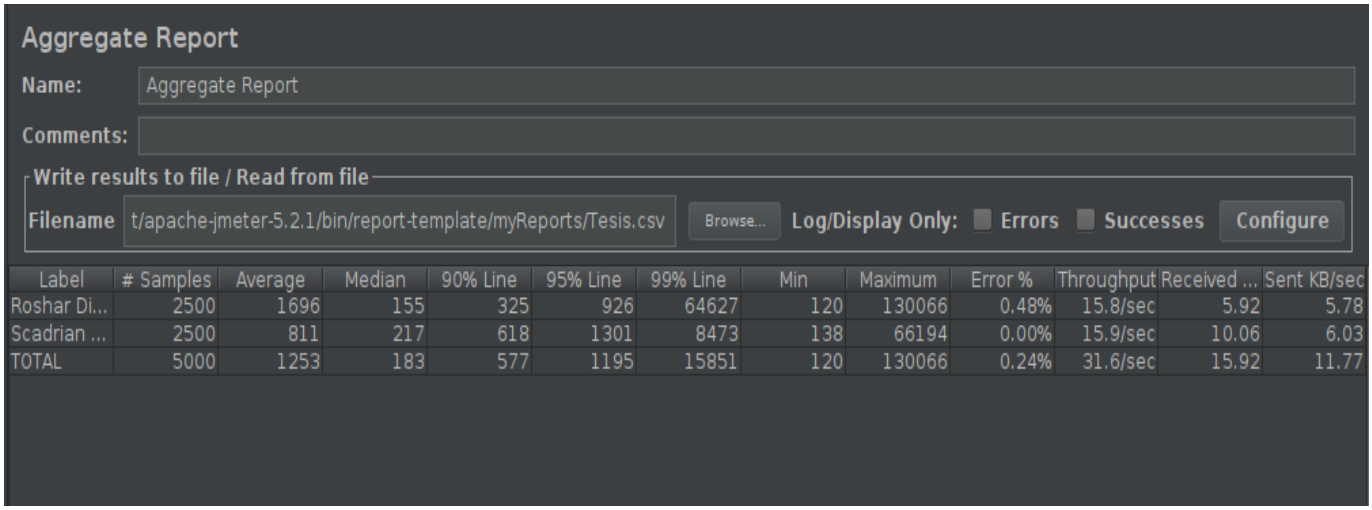
- 99% line: es el tiempo referencial en el cual el 99% de la muestra de las peticiones se encuentran por debajo de este tiempo, medido en milisegundos.
- Minimum: tiempo mínimo entre todas las peticiones, medido en milisegundos.
- Maximum: tiempo máximo entre todas las peticiones, medido en milisegundos.
- Error%: es el porcentaje de errores encontrados entre las peticiones.
- Throughtput: representa el número de peticiones procesadas por unidad de tiempo.
- Received kb/sec: representa el peso en kb recibido por unidad de tiempo.
- Sent kb/sec: representa el peso en kb enviados por unidad de tiempo.



Para la toma de resultados se configuraron las pruebas simulando 50 usuarios en simultáneo realizando un número infinito de peticiones a los servicios y se tomaron distintos cortes de datos para realizar una comparación a distintos volúmenes.

### 5.6.1 Corte de volumen bajo

Se realizó el primer corte de datos en un volumen de 5000 peticiones, de este corte se obtuvo la siguiente tabla:



Aggregate Report

Name: Aggregate Report

Comments:

Write results to file / Read from file

Filename t/apache-jmeter-5.2.1/bin/report-template/myReports/Tesis.csv Browse... Log/Display Only: Errors Successes Configure

| Label | # Samples | Average | Median | 90% Line | 95% Line | 99% Line | Min | Maximum | Error % | Throughput | Received ... | Sent KB/sec |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Roshar Di... | 2500 | 1696 | 155 | 325 | 926 | 64627 | 120 | 130066 | 0.48% | 15.8/sec | 5.92 | 5.78 |
| Scadrian ... | 2500 | 811 | 217 | 618 | 1301 | 8473 | 138 | 66194 | 0.00% | 15.9/sec | 10.06 | 6.03 |
| TOTAL | 5000 | 1253 | 183 | 577 | 1195 | 15851 | 120 | 130066 | 0.24% | 31.6/sec | 15.92 | 11.77 |

Figura 25: Tabla de Resultados Corte de Volumen Bajo



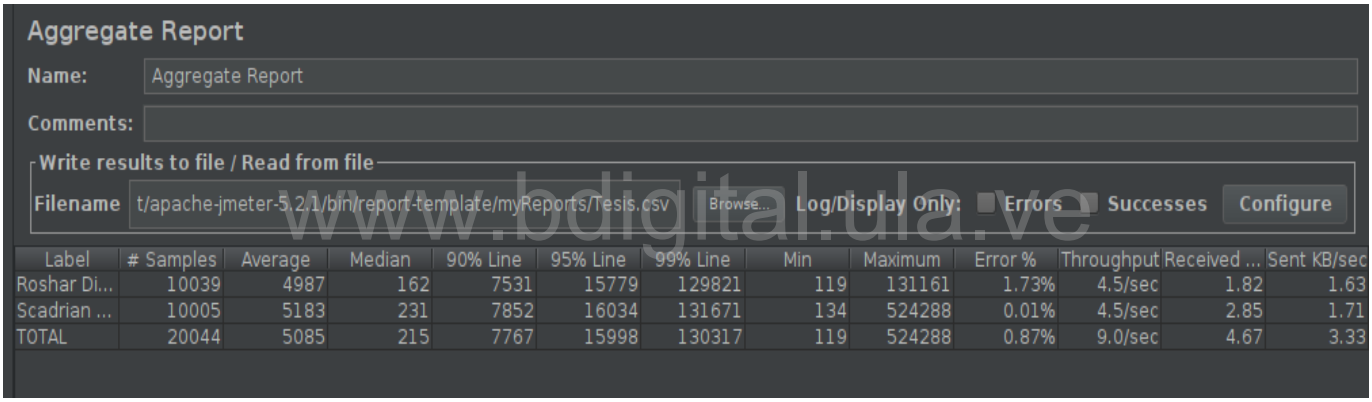
Podemos destacar del primer corte el tiempo promedio favoreciendo al servicio con arquitectura Serverless. Sin embargo, la media y el parámetro “95% Line” favorecen al servicio Monolithic, la explicación del tiempo promedio bajo para el servicio Serverless se debe a la fiabilidad del servicio, solamente el 1% de la muestra se encuentra sobre los 8473 ms mientras que para el servicio Monolithic el 1% de la muestra se encuentra sobre los 64627ms, estos valores se pueden contrastar contra el “Error %” explicando así que estos valores atípicos en los tiempos se deben a fallos en las peticiones.

Es importante resaltar que en una prueba de carga así como en una prueba de estrés de un servidor, es esperable que exista un porcentaje bajo pero existente de errores, estos se deben en su mayoría por desbordamientos de los recursos con los

que cuentan los servidores, desbordamiento de los recursos con los que cuenta el equipo o servidor desde el cual se realizan las pruebas, interrupciones en la comunicación entre cliente servidor como los picos de conexiones, caídas de conexión, entre otros.

### 5.6.2 Corte de Volumen Medio

Se realizó un corte posterior con un volumen de datos de 20000 peticiones, la tabla obtenida es la siguiente:






| Label | # Samples | Average | Median | 90% Line | 95% Line | 99% Line | Min | Maximum | Error % | Throughput | Received ... | Sent KB/sec |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Roshar Di... | 10039 | 4987 | 162 | 7531 | 15779 | 129821 | 119 | 131161 | 1.73% | 4.5/sec | 1.82 | 1.63 |
| Scadrian ... | 10005 | 5183 | 231 | 7852 | 16034 | 131671 | 134 | 524288 | 0.01% | 4.5/sec | 2.85 | 1.71 |
| TOTAL | 20044 | 5085 | 215 | 7767 | 15998 | 130317 | 119 | 524288 | 0.87% | 9.0/sec | 4.67 | 3.33 |

Figura 26: Tabla de Resultados Corte de Volumen Medio

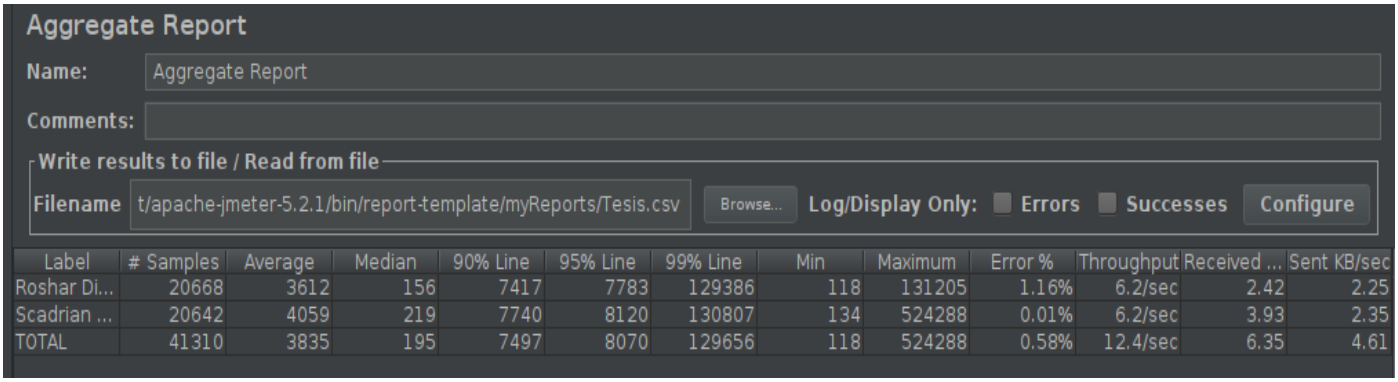
Aumentando el volumen de tráfico sobre los servidores podemos observar un tiempo promedio bastante similar entre ambos, 4987ms para el servicio Serverless y 5183ms para el servicio Monolithic.

Por su parte el resto de métricas de tiempo favorecen al servicio Monolithic, especialmente el parámetro “Maximum” el cual es 5 veces superior para el servicio Serverless. Sumado al parámetro “Error %” donde vemos a ambas aplicaciones fallando, podemos observar que el tiempo de espera de solicitud del servidor es superior al configurado en el servidor Monolitic, esto constituye simplemente una discrepancia en las configuraciones. De mantener la homogeneidad en la configuración de ambos servidores, se observaría un porcentaje superior de fallas para el servicio Scadrian, pero unas métricas de tiempo inferiores.

### 5.6.3 Corte de Volumen Alto

Finalmente se tomó un último corte parta un volumen superior a las 40000 peticiones, del cual se obtuvo la siguiente tabla:






| Label | # Samples | Average | Median | 90% Line | 95% Line | 99% Line | Min | Maximum | Error % | Throughput | Received ... | Sent KB/sec |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Roshar Di... | 20668 | 3612 | 156 | 7417 | 7783 | 129386 | 118 | 131205 | 1.16% | 6.2/sec | 2.42 | 2.25 |
| Scadrian ... | 20642 | 4059 | 219 | 7740 | 8120 | 130807 | 134 | 524288 | 0.01% | 6.2/sec | 3.93 | 2.35 |
| TOTAL | 41310 | 3835 | 195 | 7497 | 8070 | 129656 | 118 | 524288 | 0.58% | 12.4/sec | 6.35 | 4.61 |

Figura 27: Tabla de Resultados Corte de Volumen Alto

El corte final es una extrapolación del corte anterior. Las métricas se mantienen porcentualmente similares con tiempos promedios de 3612ms para el servicio Monolithic y 4059ms para el servicio Serverless.

Vale la pena destacar la fiabilidad del servicio Serverless con un porcentaje de error de 0.01% contra 1.16% para el servicio Monolithic. Sin embargo, como se explicó anteriormente este valor es un reflejo de una discrepancia en los parámetros de configuración de los servidores, donde el servicio Serverless cuenta con un tiempo de espera de solicitud del servidor 5 veces superior a su contraparte, afectando sus métricas de tiempo aceptando solicitudes más longevas, pero disminuyendo su porcentaje de error.



### 5.6.4 Comparación entre Arquitecturas.

Para comparar la arquitectura Monolithic con la arquitectura Serverless la métrica más importante a considerar es el tiempo promedio de respuesta. Analizando los resultados obtenidos se observa en el corte de alto volumen una diferencia del 11% favoreciendo a la arquitectura Monolithic. Sin embargo no es de extrañar esta diferencia dada la métrica de porcentaje de error, La cual es del 0,01% para el servicio Serverless contra 1,16% para el servicio Monolithic. Esta diferencia tan significativa en los porcentajes de error se debe a la configuración del tiempo de espera de solicitud del servidor el cual es 5 veces superior para el servidor Serverless.

Funcionalmente ambas arquitecturas solventan el problema suministrado, sin embargo la arquitectura Serverless es bastante más sencilla de configurar y la cantidad de clases y código asociado es muy inferior a un servicio con arquitectura tradicional, donde cada componente requiere su propia configuración.

| | Arquitectura Serverless | Arquitectura Monolithic |
|---|---|---|
| **Tiempo de respuesta** | | **X** |
| **Porcentaje de error** | **X** | |
| **Complejidad de desarrollo** | **X** | |
| **Accesibilidad a Documentación** | | **X** |
| **Costo de mantenimiento** | **X** | |

Tabla 1: Comparativa Arquitectura Serverless Vs. Arquitectura Monolithic

# Capítulo 6 Conclusiones y Recomendaciones

## 6.1 Conclusiones.

- La arquitectura Serverless para el desarrollo de servicios webs es tan eficiente como las arquitecturas tradicionales bajo ciertas condiciones como servicios ligeros con largos o medianos periodos de inactividad y una cantidad de peticiones bajas y periódicas.

- Si bien no se puede afirmar que es más sencillo desarrollar en arquitecturas Serverless debido a múltiples factores como la limitada información disponible sobre un concepto en evolución, si podemos concluir que el volumen de código generado y la cantidad de clases requeridas para mantener un modelo organizado es mucho menor.

- El nivel de customización de un servicio bajo arquitectura Serverless se ve limitado por la cantidad de configuraciones automatizadas que se realizan en el proceso de generación de los recursos de lambda. Lo cual puede acarrear problemas a la hora de contar con requerimientos muy específicos en cuanto al desempeño y desenvolvimiento del servidor.

- La fiabilidad y robustez del servicio Lambda de AWS quedó patente en todo el proceso de desarrollo, demostrando la puesta futuro y el respaldo de Amazon con el concepto de arquitectura Serverless emergente.

## 6.2 Recomendaciones.

- Debido a la naturaleza de Apache Jmeter se realizaron pruebas continuas en un intervalo de tiempo, situación que favorece a la arquitectura Monolithic, por tal razón se recomienda realizar pruebas periódicas a intervalos variables donde Serverless demuestra su potencial.



- De igual forma se recomienda realizar una medición sobre los costos reales de un servicio Serverless alojado con Lambda de AWS contra un servicio tradicional alojado en un servidor como EC2 de AWS.

- Evitar limitantes como el máximo de memoria establecido en 1gb para el procesamiento en Lambda de AWS posibilitaría pruebas con mayor costo computacional, las cuales serían de mayor interés y enriquecimiento.

- Se recomienda aplicar la arquitectura Serverless únicamente en servicios ligeros con largos o medianos periodos de inactividad y una cantidad de peticiones bajas y periódicas.

- Serverless es un concepto en constante evolución, limitarlo a una arquitectura en solitario significaría desaprovechar en gran medida su potencial, razón por la cual se recomienda considerar Serverless como una herramienta de la cual aprovechar sus mejores virtudes, adoptar arquitecturas mixtas puede ser más provechoso que encasillarse con la idea de aplicar una arquitectura única.

**Bibliografía**

Alda Alverro, Alvarez Fernando, Díaz Gabriel, Evgeniev Martín, Horrillo Pancho y BBVA Labs. (2019). La Economía de las Arquitecturas “Serverless” Recuperado de https://www.bbva.com/es/economia-arquitecturas-serverless/

Álvarez C., Cecilio. (2020). Blog sobre Java EE. Recuperado de https://www.arquitecturajava.com/que-es-spring-boot/

Apache Software Foundation. (2019). Apache JMeter TM. Recuperado de https://jmeter.apache.org/

Aplyca Tecnología S.A.S. (2018). De sistemas monolíticos a microservicios. Recuperado de http://aplyca.com/es/blog/aplicaciones-monoliticas-o-microservicios

Aprederaprogramar.com. (2020). ¿Qué es y para qué sirve JSON? Especificación oficial Javascript Object Notation. Diferencia de XML (CU01213F) Recuperado de https://aprenderaprogramar.com/index.php?option=com_content&view=article&id=956:ique-es-y-para-que-sirve-json-especificacion-oficial-javascript-object-notation-diferencia-de-xml-cu01213f&catid=83&Itemid=212

Amazon Web Services, Inc. (2020). Creación de aplicaciones con arquitectura sin servidor. Recuperado de https://aws.amazon.com/es/lambda/serverless-architectures-learn-more/

Borillo, Ricardo y Genbeta. (2019). ¿Qué es serverless y por qué adoptarlo en el desarrollo de tu próxima aplicación? Recuperado de https://www.genbeta.com/desarrollo/que-serverless-que-adoptarlo-desarrollo-tu-proxima-aplicacion

Geeksforgeeks. (2019). Difference between Load Testing and Stress Testing Recuperado de https://www.geeksforgeeks.org/difference-between-load-testing-and-stress-testing/

Goodwill comunity foundation, Inc. (2020). Informática Básica.- ¿Qué son las aplicaciones web? Recuperado de https://edu.gcfglobal.org/es/informatica-basica/que-son-las-aplicaciones-web/1/



Lázaro, Diego. (2018). Introducción a los Web Services. Recuperado de https://diego.com.es/introduccion-a-los-web-services

Mozilla and Individual contributors. (2020). Java Script. Recuperado de https://developer.mozilla.org/es/docs/Web/JavaScript

Oracle. (2000). Java. Recuperado de https://www.java.com/es/download/faq/whatis_java.xml

Pressman, R. (2005). Ingeniería del Software. Un enfoque práctico. España: Mc Graw Hill.

Proyecto A (Ajpd Soft). (2018). Tomcat, Apache Tomcat, Jakarta Tomcat. Recuperado de https://www.ajpdsoft.com/modules.php?name=Encyclopedia&op=content&tid=769

Tryqa.com. (2019). What is Load Testing? Recuperado de http://tryqa.com/what-is-load-testing-in-software/

Viewnext. (2019). Arquitectura de Microservicios vs Arquitectura Monolítica. Recuperado de https://www.viewnext.com/arquitectura-de-microservicios-vs-arquitectura-monolitica/



Wiboo. (2017). ¿Qué son las Aplicaciones Web? Ventajas y Tipos de Desarrollo Web. Recuperado de https://wiboomedia.com/que-son-las-aplicaciones-web-ventajas-y-tipos-de-desarrollo-web/