
UNIVERSIDAD TÉCNICA DEL NORTE

Facultad de Ingeniería en Ciencias Aplicadas Carrera de Ingeniería en Sistemas Computacionales

# DESARROLLO DE UNA ARQUITECTURA EFICIENTE ORIENTADA A MICROSERVICIOS CON GRAPHQL UTILIZANDO LA CALIDAD EXTERNA DE LAS NORMAS ISO/IEC 25023.

**Trabajo de grado presentado ante la ilustre Universidad Técnica del Norte previo a la obtención del título de Ingeniero en Sistemas Computacionales.**

Autor:

Paul Alejandro Rocha Castro

Director:

Msc. Antonio Quiña

Ibarra – Ecuador 2021

# UNIVERSIDAD TÉCNICA DEL NORTE

## BIBLIOTECA UNIVERSITARIA

# AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

1.- IDENTIFICACIÓN DE LA OBRA

En cumplimiento del Art. 144 de la Ley de Educación Superior, hago la entrega del presente trabajo a la Universidad Técnica del Norte para que sea publicado en el Repositorio Digital Institucional, para lo cual pongo a disposición la siguiente información:

| DATOS DE CONTACTO | | | |
|---|---|---|---|
| **CÉDULA DE IDENTIDAD:** | 1003034863 | | |
| **APELLIDOS Y NOMBRES:** | PAUL ALEJANDRO ROCHA CASTRO | | |
| **DIRECCIÓN:** | Maldonado 17-41 y Juan Francisco Bonilla | | |
| **EMAIL:** | paul.rocha.81@gmail.com | | |
| **TELÉFONO FIJO:** | 062957968 | **TELÉFONO MÓVIL:** | 0994987718 |

| DATOS DE LA OBRA | |
|---|---|
| **TÍTULO:** | DESARROLLO DE UNA ARQUITECTURA EFICIENTE ORIENTADA A MICROSERVICIOS CON GRAPHQL UTILIZANDO LA CALIDAD EXTERNA DE LAS NORMAS ISO/IEC 25023. |
| **AUTOR (ES):** | PAUL ALEJANDRO ROCHA CASTRO |
| **FECHA: DD/MM/AAAA** | 18/06/2021 |

| SOLO PARA TRABAJOS DE GRADO | |
|---|---|
| **PROGRAMA:** | **PREGRADO** ■ **POSGRADO** □ |
| **TITULO POR EL QUE OPTA:** | INGENIERO EN SISTEMAS COMPUTACIONALES |
| **ASESOR /DIRECTOR:** | Msc. Antonio Quiña |

**CONSTANCIA.**

El autor manifiesta que la obra objeto de la presente autorización es original y se la desarrolló, sin violar derechos de autor de terceros, por lo tanto, la obra es original y que es el titular de los derechos patrimoniales, por lo que asume la responsabilidad sobre el contenido de la misma y saldrá en defensa de la Universidad en caso de reclamación por parte de terceros.

Ibarra, a los 18 días del mes de Junio de 2021

**EL AUTOR:**

PAUL ALEJANDRO ROCHA CASTRO

Firmado digitalmente por PAUL ALEJANDRO ROCHA CASTRO
Fecha: 2021.06.17 22:47:57 -05'00'

Nombre: Paul Alejandro Rocha Castro

C.I.:1003034863

**CERTIFICACIÓN DEL DIRECTOR**

En mi calidad de tutor de Trabajo de Grado presentado por el egresado **PAUL ALEJANDRO ROCHA CASTRO** para obtener Título de Ingeniería en Sistemas Computacionales cuyo tema es: **Desarrollo de una arquitectura eficiente orientada a microservicios con Graphql utilizando la calidad externa de las normas ISO/IEC 25023.** Considero que el presente trabajo reúne los requisitos y méritos suficientes para ser sometido a la presentación pública y evaluación por parte del tribunal examinador que se designe.

En la ciudad de Ibarra, a los 18 días del mes de junio del 2021

JOSE ANTONIO QUIÑA MERA

Firmado digitalmente por JOSE ANTONIO QUIÑA MERA
Fecha: 2021.06.18 08:27:19 -05'00'

Msc. Antonio Quiña

**DIRECTOR DE TRABAJO DE GRADO**

# DEDICATORIA

A Dios:

Por darme la vida y estar siempre conmigo, guiándome en mí camino.

A mis Padres:

El esfuerzo y las metas alcanzadas, refleja la dedicación, el amor que invierten sus padres en sus hijos.

A mi madre Jaqueline, por ser el pilar más importante y por demostrarme siempre su cariño y apoyo incondicional, por estar conmigo en todo momento y que con su ejemplo me ha demostrado a siempre luchar por mis sueños.

A mi padre, quien con su amor, paciencia y esfuerzo ha coadyuvado para llegar a cumplir hoy un sueño más, gracias por inculcar en mí el ejemplo de esfuerzo y valentía, de no temer a las adversidades porque Dios está conmigo siempre.

A mi hermano, Juan Carlos por su cariño y apoyo incondicional, durante todo este proceso.

A toda mi familia porque con sus oraciones, consejos y palabras de aliento hicieron de mí una mejor persona y de una u otra forma me acompañan en todos mis sueños y metas.

A Gaby:

En el camino encuentras personas que iluminan tu vida, que con su apoyo alcanzas de mejor manera tus metas, a través de sus consejos, de su amor y paciencia. Mi amor y gratitud para Gaby por apoyarme cuando más la necesito, por extender su mano en momentos difíciles y por el amor brindado cada día, de verdad mil gracias, siempre la llevo en mi corazón.

> "La dicha de la vida consiste en tener siempre algo que hacer, alguien a quien amar y alguna cosa que esperar". **Thomas Chalmers**

# AGRADECIMIENTO

Agradezco a Dios por ser la luz y guiarme en el transcurso de mi vida, brindándome paciencia y sabiduría para culminar con éxito mis metas propuestas.
A mis padres por ser el pilar fundamental y brindarme su apoyo incondicionalmente, en todo momento.
Mi especial agradecimiento a mi director de tesis Msc. Antonio Quiña quien con su experiencia, conocimiento y motivación me oriento en la investigación.
A todos docentes que, con su sabiduría, conocimiento y apoyo, motivaron a desarrollarme como persona y profesional en la UTN.

A la Universidad Técnica del Norte por ser la sede de todo el conocimiento adquirido durante mi carrera universitaria.

"La gratitud, como ciertas flores, no se da en la altura y mejor reverdece en la tierra buena de los humildes" **— José Martí**

# Tabla de Contenido

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

# RESUMEN

El presente trabajo tiene como finalidad el desarrollo de una arquitectura eficiente orientada a microservicios con GraphQL utilizando la calidad externa de las normas ISO/IEC 25023,el cual se centra en la teoría y conceptos de muchas de las tecnologías que se utilizarán para su desarrollo, siendo el foco principal las definiciones de arquitectura, los microservicios y comparaciones de los mismos frente a sus predecesores, la definición de GraphQL y el impacto que tiene el mismo en la actualidad, así como una gran parte de teórica con respecto a lo que se refiere el estándar ISO/IEC 25023.

Continuando con el desarrollo del trabajo se realiza la parte práctica para demostrar lo definido en los objetivos tanto general como específicos, para lo cual se hace uso de un ambiente de experimentación en el cual se utilizó GraphQL y librerías de apoyo, así como diferentes lenguajes de programación más utilizados en la industria del desarrollo de software, todo esto siguiente las normas del estándar ISO/IEC 25023

Finalmente se muestra los resultados obtenidos en el ambiente de experimentación mediante tablas de valores y gráficas para hacer una comparativa del comportamiento de la arquitectura frente a diferentes casos de uso.

# ABSTRACT

The purpose of this work is to develop an efficient architecture oriented to microservices with GraphQL using the external quality of the ISO / IEC 25023 standards, which at the beginning of this study focuses on the theory and concepts of many of the technologies that will be used. for its development, the focus being the architecture definitions, microservices and comparisons of the same against their predecessors, the definition of GraphQL and the impact it has today, as well as a large part of theoretical with respect to as regards the ISO / IEC 25023 standard.

Continuing with the development of the work, the practical part is carried out to demonstrate what is defined in both the general and specific objectives, for which use is made of an experimentation environment in which GraphQL and support libraries will be used, as well as different languages of programming such as JavaScript and its ReactJS framework, all this following the standards of the ISO / IEC 25023 standard.

Finally, the results obtained in the experimentation environment are shown by means of tables of values and graphs to make a comparison of the behavior of the architecture against different use cases.

# INTRODUCCIÓN

## Problema

### *Antecedentes*

El continuo crecimiento de internet y la introducción de dispositivos tecnológicos como teléfonos, el Internet de las cosas (IoT), computadoras de escritorio y una variedad de aplicaciones de servidor, presentan diferentes características; motivado a esto, los desarrolladores diseñan un conjunto de funciones y procedimientos para mantener una Interfaz de Programación de Aplicaciones, con sus siglas en inglés API, para cada una de estas. De esta manera, se generan muchos gastos y costos en desarrollo y mantenimiento de las API; sin embargo, el protocolo web más usado es REST (Wiesbauer, 2019)

Los diseños de arquitecturas como REST tienen el problema de obtención de información en exceso o deficiente debido a que siempre devuelven una estructura fija. No se puede obtener exactamente los datos que se solicita a menos que se cree un punto final específico para esto (Maldonado, 2019)

El estilo arquitectónico de microservicios ha obtenido una significativa atención desde el punto de la investigación. Sin embargo, hoy en día es difícil para los investigadores y profesionales tener una visión clara de soluciones existentes para la arquitectura de microservicios (Di Francesco, Lago, & Malavolta, 2019)

Las migraciones a microservicios por parte de las empresas se han vuelto muy populares en los últimos años por diferentes razones, por ejemplo, porque esperan mejorar la calidad de su sistema o facilitar el mantenimiento del software (Lenarduzzi, Lomio, saarimaki, & Taibi, 2020)

### *Situación actual*

Empresas como Amazon, Deutsche Telekom, LinkedIn, Netflix, SoundCloud, The Guardian, Uber y Verizon están adoptando rápidamente enfoques basados en microservicios. A menudo, los microservicios se utilizan para modernizar aplicaciones heredadas, ya que su objetivo es dividir sistemas monolíticos en microservicios para respaldar así la modernización incremental del software heredado (Lenarduzzi, Lomio, saarimaki, & Taibi, 2020). Por consiguiente, han aparecido nuevos protocolos web, uno de ellos es GraphQL, el cual fue desarrollado por la empresa de tecnología Facebook en 2012, pero solo en 2017 se patentó

esta tecnología. Es así como inmediatamente tuvo auge entre los desarrolladores, pero, por supuesto, todavía existen dudas sobre esta tecnología si se compara con el protocolo web REST que se ha utilizado por desarrolladores desde 2000 (Hartina, Lawi, & Enrico, 2018)

Ya que existe un bajo grado de análisis, al hallarse con varias arquitecturas y problemas de migraciones, algunas empresas optan por escoger arquitecturas sin antes realizar un estudio de verificación de eficiencia de su comportamiento orientado a microservicios.

### *Prospectiva*

La demanda de arquitecturas más robustas y eficientes para todo tipo de usuario se ha convertido en una necesidad primordial en el campo de la ingeniería de software.

Esta investigación consiste en establecer una arquitectura eficiente orientada a microservicios utilizando el lenguaje de consultas GraphQL. Para verificar la eficiencia de la arquitectura, se realizará un laboratorio de experimentos que compare la eficiencia de comportamiento con el estándar internacional ISO/IEC 25023 y, de esta manera, aportar un modelo de arquitectura orientada a microservicios con GraphQL.

### *Planteamiento del problema*

La arquitectura de software juega un papel indispensable en el éxito o el fracaso de cualquier sistema de software, ya que se ocupa de la estructura base, los subsistemas y las interacciones entre estos subsistemas (Farshidi, Jansen, and Werf 2020).

Existen arquitecturas orientadas a microservicios que han sido implementadas en los últimos años en diferentes tipos de sistemas empresariales. Varias de estas, al no estar validadas por un estándar internacional como ISO, pueden presentar ciertos inconvenientes en su desempeño, lo cual da como resultado un rendimiento ineficiente de sus aplicaciones web en sus arquitecturas orientadas a microservicios establecidas.

## Objetivos

### *Objetivo general*

Desarrollar una arquitectura (eficiente) orientada a microservicios utilizando el lenguaje de consultas GraphQL y validar con el estándar ISO/IEC 25023.

### *Objetivos específicos*

- Elaborar un marco conceptual y tecnológico para establecer una arquitectura

orientada a microservicios.

- Comparar tecnologías de *backend* y *frontend* utilizando el lenguaje de consultas GraphQL mediante experimentos de laboratorio.
- Establecer una arquitectura eficiente y validar el funcionamiento de la arquitectura establecida mediante una prueba de concepto

**Alcance**

Mediante la siguiente investigación, se obtendrá una arquitectura orientada a microservicios eficientes utilizando el lenguaje de consultas GraphQL que se valide con el estándar ISO/IEC 25023.

El estudio consiste en realizar una búsqueda de tecnologías más eficientes de *backend* y *frontend* para GraphQL mediante una revisión bibliográfica y principales portales tecnológicos como GitHub, Prisma y APIGuru con el fin de establecer la arquitectura orientada a microservicios que se utilizará para el experimento.

Una vez establecidas, se procederá a comparar las tecnologías de *backend* y *frontend* utilizando el lenguaje de Consultas GraphQL mediante casos de uso con experimentos de laboratorio.

Con la arquitectura se procederá a validar su funcionamiento mediante pruebas de concepto con referencia al estándar ISO/IEC 25023 para comprobar la eficiencia de comportamiento de esta específicamente en el tiempo medio de respuesta y de rendimiento.

De esta manera se podrá presentar una arquitectura orientada a microservicios con sus tecnologías ya establecidas, además su eficiencia ya medida por el estándar ISO/IEC 25023.

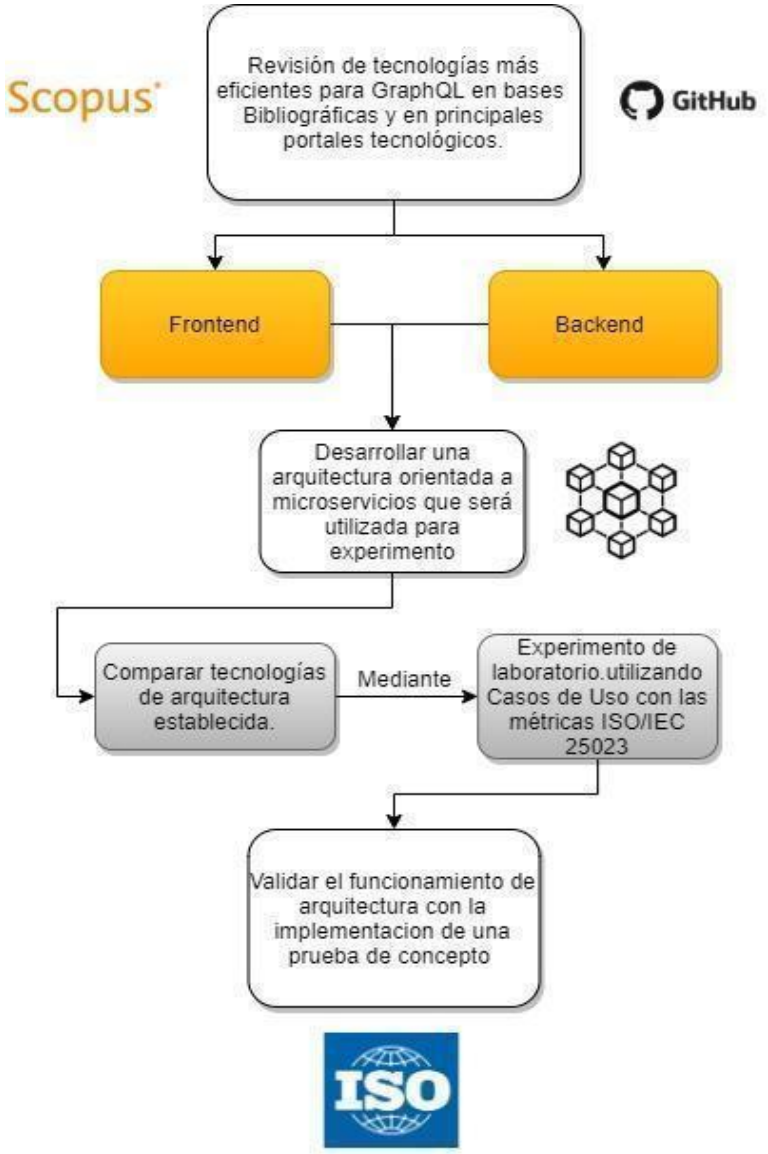


*Figura 1: Mentefacto*
*Autor: Elaboración Propia*

**Justificación**

***ODS***

El desarrollo comprobado de una arquitectura eficiente orientada a microservicios con GraphQL apoyará al fortalecimiento de los Objetivos de Desarrollo Sostenible (ODS), en especial el #9 “Industria, Innovación e Infraestructura” específicamente la meta #9.b que hace referencia a apoyar el desarrollo de tecnologías, la investigación y la innovación nacionales en los países en desarrollo, incluso garantizando un entorno normativo propicio a la diversificación industrial y la adición de valor a los productos básicos, entre otras cosas (Naciones Unidas

2015).

***Impacto Tecnológico***

Los microservicios se desarrollan, implementan y mantienen por separado, lo cual permite que los equipos sean autónomos y que puedan decidir qué tecnología utilizar y adaptarse a las necesidades actuales del comportamiento empresarial. El objetivo de los microservicios es dividir el comportamiento empresarial en pequeños servicios que puedan funcionar de forma independiente entre sí (Bottero, Datola, & Monaco, 2017). Por otro lado, el lenguaje de consulta GraphQL se utiliza para crear aplicaciones cliente y su objetivo es combinar todas las solicitudes de datos en una, lo que resulta en un número reducido de estas para la recuperación / entrega de datos (Khan & Noor, 2020)

***Impacto Metodológica***

ISO / IEC 25023 - Medición de la calidad del producto del sistema y del software: proporciona medidas que incluyen funciones de medición asociadas a las características de calidad en el modelo de calidad del producto (Organización Internacional de Normalización, 2016)

**Contexto**

El desarrollo de esta investigación se enmarcó con el contexto de otras iniciativas, que han abordado algunos tópicos relacionados, entre estas se mencionan las siguientes:

- Arquitectura de software basada en microservicios para desarrollo de aplicaciones web de la Asamblea Nacional, desarrollado en la Universidad Técnica del Norte, Ibarra, Ecuador.Propone una arquitectura de software basada en microservicios para el desarrollo de aplicaciones web en la Coordinación General de Tecnologías de la Información y Comunicación de la Asamblea Nacional del Ecuador (Hinojosa, 2017)
- Desarrollo de un envoltorio del api-rest de Mendeley con GraphQL, emprendido en la Universidad Técnica del Norte, Ibarra, Ecaudor. Con el objetivo de desarrollar un envoltorio del API-REST del gestor bibliográfico Mendeley utilizando el lenguaje de consultas GraphQL y validado con un marco de trabajo de calidad en uso basado en el estándar ISO/IEC 25000 (Rodríguez, 2020)
- Estudio de una arquitectura de microservicios mediante Spring Cloud para el

desarrollo del módulo de registro y seguimiento médico de los deportistas en la Federación deportiva de Imbabura Universidad Técnica del Norte, Ibarra, EcaudorEstudiar una arquitectura de microservicios mediante Spring Cloud para el desarrollo del módulo de registro y seguimiento médico de los deportistas en la Federación Deportiva de Imbabura (Flores, 2020).

- Migración hacia una arquitectura basada en microservicios del sistema de gestión centralizada de laboratorios de la DGIP. Ejecutado en la Escuela Politécnica Nacional, Quito, Ecuador. Este trabajo se plantea la implementación de una arquitectura de microservicios basado en una propuesta metodológica de migración de arquitecturas monolíticas hacia microservicios (Chulca & Molina, 2020).
- Diseño de un prototipo de una arquitectura basada en microservicios para la integración de aplicaciones Web altamente transaccionales. Caso: Entidades financieras. Ejecutado en la Universidad Politécnica Salesiana, Quito, Ecuador. Como resultado obtuvo una arquitectura de microservicios que nació como un enfoque para mejorar las arquitecturas tradicionales (monolíticas), enfocadas en la estabilidad, la confiabilidad, un alto grado de tolerancia a fallas y una escala simple (Chicaiza, 2020).
- Aplicación de un modelo para evaluar el rendimiento en el proceso de migración de una aplicación monolítica hacia una orientada a microservicios, en la Universidad Técnica Particular de Loja, Ecuador. Trabajo de fin de titulación consiste en la aplicación de un modelo que permita evaluar el rendimiento en el proceso de migración de una aplicación monolítica hacia una orientada a microservicios (Yaguachi, 2017)

# CAPÍTULO I

# Marco teórico

## 1.1. Arquitectura de software

La arquitectura de software de un sistema, está compuesto por un conjunto de elementos conocidos como aplicaciones, las relaciones entre ellas y sus propiedades, el orden o la estructura que adopten le dará una funcionalidad que responderá a una solicitud o finalidad, esta herramienta permite plasmar el diseño de una solución o un producto informático que resuelve una problemática, con la automatización de actividades y procesos, así como flujos y procesamientos de datos(Bass, Clements, & Kazman, 2003).

De igual forma, Rozanski y Woods (2012) mencionan que la arquitectura del software considera aspectos a nivel macro, ya que se trata de una colección de módulos, procesos, datos del sistema, la estructura del software y la relación entre ellos. Otros aspectos que se incluyen en la arquitectura son los métodos de expansión y modificación de la tecnología de la que depende el rendimiento del sistema y la flexibilidad, los métodos y planes de implementación o modificación del sistema, entre otros elementos. En este sentido, la arquitectura del software puede considerarse que juega un papel decisivo en las funciones básicas y en la estructura principal de un sistema. El diseño de la arquitectura de software consiste en analizar las necesidades del cliente, recabar requisitos funcionales y transformar las necesidades del cliente en un modelo de software. Por consiguiente, la arquitectura del software es una solución abstracta para los requisitos y puede proporcionar la orientación correspondiente para todos los aspectos del diseño de sistemas a gran escala, por lo cual, se ubica en la etapa inicial del desarrollo de software (Rozanski y Woods, 2012).

### *1.1.1. Evolución de Arquitectura de Software*

El desarrollo de la arquitectura de software ha pasado por un proceso desde la arquitectura monolítica, distribuida, SOA hasta la arquitectura de microservicios.

#### a) Arquitectura monolítica

Se empaquetan todos los módulos funcionales juntos y se los coloca en un contenedor web para que se ejecuten. Debido a que todos los módulos funcionales usan la misma base de datos, se lograron apreciar ciertas características, ventajas y desventajas que

se describen a continuación:

**Características:**

- Todas las funciones están integradas en un proyecto.
- Todas las funciones se implementan en un paquete, el cual será aplicado en el servidor.
- Mejora el rendimiento del sistema mediante la implementación de grupos de aplicaciones y de bases de datos.

En la tabla 1.1, se muestra un resumen de las ventajas y desventajas de las arquitecturas monolíticas.

TABLA 1.1
Ventajas y desventajas de la arquitectura monolítica.

| Ventajas | Desventajas |
|---|---|
| • Debido a la estructura de proyecto simple, bajo costo de pre-desarrollo y ciclo cortó, es la primera opción para proyectos pequeños. | • Todas las funciones están integradas en un proyecto, lo cual dificulta el desarrollo, la expansión y el mantenimiento en grandes proyectos. |
| • Es fácil de implementar, posee bajo costo de operación y mantenimiento, y se encuentra empaquetado directamente en un paquete completo, consiste en copiar en un directorio del contenedor web para ejecutar. | • La velocidad de iteración de la versión se está ralentizando gradualmente. Para modificar un lugar, toda la aplicación debe compilarse, implementarse, iniciarse, desarrollarse y probarse por un período prolongado. |
| • Alta eficiencia de desarrollo, las llamadas a métodos locales se utilizan para la interacción entre módulos. | • No se puede escalar según la demanda, realizar una expansión horizontal mediante la agrupación en clústeres ni escalar según la demanda para un determinado negocio. |
| • Es fácil de probar, ya que los IDE están diseñados para el desarrollo de una sola aplicación y puede iniciar un sistema completo localmente. | |

Nota: Elaboración Propia

### b) Arquitectura distribuida

Según Rozanski y Woods (2012) se divide un sólo sistema en varios subsistemas verticalmente según el requerimiento. Adicionalmente, los autores indican las características, ventajas y desventajas que se detallan a continuación:

**Características:**

- Dividida verticalmente en un sólo sistema por empresa, esta arquitectura también se denomina arquitectura vertical.
- Hay redundancia de datos entre los sistemas.

- El acoplamiento es relativamente grande.
- Las interfaces entre sistemas se utilizan principalmente para sincronización de datos.

En la taba 1.2, se muestra un resumen de las ventajas y desventajas de las arquitecturas distribuidas.

TABLA 1.2
Ventajas y Desventajas de la Arquitectura distribuida.

| Ventajas | Desventajas |
|---|---|
| • A través de la división vertical, cada subsistema se convierte en un sistema pequeño con funciones simples, bajo costo de predesarrollo y ciclo corto.<br>• Cada subsistema se puede escalar según sea necesario.<br>• Cada subsistema puede utilizar diferentes tecnologías. | • Hay redundancia de datos, redundancia funcional y alto acoplamiento entre subsistemas.<br>• El escalado bajo demanda no es suficiente para realizar diferentes servicios en el mismo subsistema. |

Nota: Elaboración Propia

### c) Arquitectura Orientada a Servicios (SOA siglas en ingles)

Hinojosa (2017) indica que la arquitectura orientada a servicios, basada en una arquitectura distribuida que divide diferentes funciones comerciales en servicios y las conecta a través de interfaces y protocolos bien definidos entre estos servicios. Asimismo, el autor menciona las siguientes características, ventajas y desventajas:

#### Características

- Su flexibilidad, que permite la reutilización.
- Su versatilidad, que hace posible que los servicios puedan ser consumidos por los clientes en aplicaciones o procesos de negocio distintos.
- Sus posibilidades, que optimizan el trabajo con datos y su coordinación.
- Aumenta la eficiencia en los procesos.
- Amortiza la inversión realizada en sistemas.
- Reduce costes de mantenimiento.
- Fomenta la innovación orientada al desarrollo de servicios.
- Simplifica el diseño, optimizando la capacidad de organización.

En la tabla 1.3, se muestra un resumen de las ventajas y desventajas de las arquitecturas SOA.

TABLA 1.3
Ventajas y desventajas de la Arquitectura SOA.

| Ventajas | Desventajas |
|---|---|
| • Extraer funciones repetitivas como servicios, mejorar la eficiencia del desarrollo y mejorar la capacidad de reutilización y mantenimiento del sistema.<br>• Se puede escalar bajo demanda según las características de los diferentes servicios.<br>• Utilice ESB para reducir el acoplamiento de interfaces en el sistema. | • La difuminación del límite entre el sistema y el servicio conducirá a la granularidad excesiva del servicio extraído y al alto acoplamiento entre el sistema y el servicio.<br>• Aunque se utiliza ESB, el protocolo de interfaz de servicio no es fijo y hay muchos tipos, lo que no favorece el mantenimiento del sistema. |

Nota: Elaboración Propia

### d) Arquitecturas orientadas a microservicios

Este tipo arquitectura se basa en la idea de SOA mediante una capa de servicio detallada, para satisfacer las necesidades actuales de Internet móvil para proyectos a gran escala y múltiples clientes. Cada servicio tiene una granularidad muy pequeña, sólo completa una función comercial específica (Baresi y Garriga 2020). Los mismos autores expresan que, entre las características, ventajas y desventajas de este tipo de arquitectura se pueden mencionar las siguientes:

**Características:**

- Altamente mantenible y comprobable
- Débilmente acoplada
- Desplegable independientemente
- Organizada en torno a las capacidades comerciales
- Propiedad de un pequeño equipo
- La capa de servicio se divide en microservicios, que tienen una sola responsabilidad, según el negocio.
- Se utilizan protocolos de transmisión ligeros como RESTful y RPC entre microservicios. Es propicio para adoptar una arquitectura de separación de *frontend* y *backend*.

  En la figura 2, muestra de manera gráfica claramente la diferencia entre las arquitecturas monolíticas y microservicios.

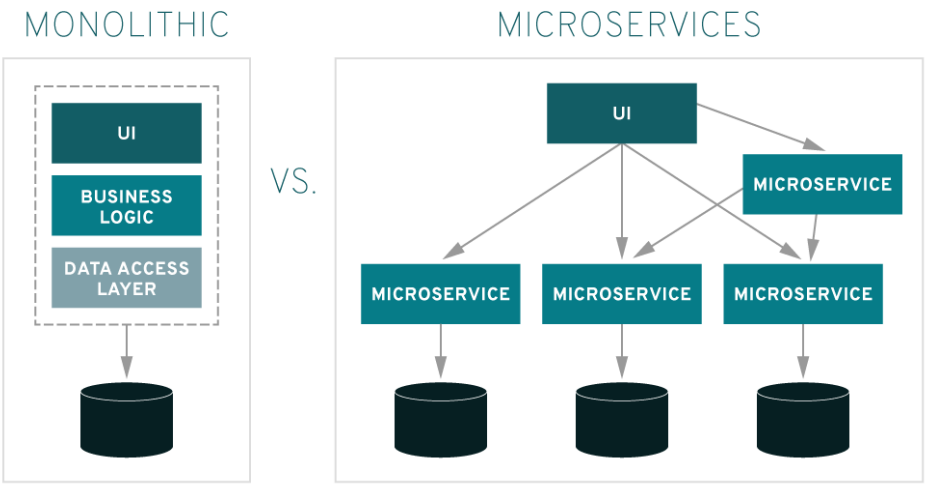


Figura 2: Monolítica Vs Microservicios

En la tabla 1.4, se muestra un resumen de las ventajas y desventajas de las arquitecturas microservicios.

TABLA 1.4
Ventajas y desventajas de los microservicios

| Ventajas | Desventajas |
|---|---|
| • Reducir el riesgo: permite que cada característica se desarrolle e implemente de forma independiente. Esta separación reduce el riesgo, ya que la falla de un servicio no provocará la falla de los otros servicios.<br>• Proporcionar almacenamiento de datos flexible: Permite a las organizaciones almacenar datos en múltiples ubicaciones y que los desarrolladores elijan el tipo de almacenamiento que mejor se adapte a las necesidades de cada servicio.<br>• Habilitar la programación políglota: Permite a los desarrolladores seleccionar el lenguaje de programación más adecuado para el trabajo.<br>• Reducir el desorden: Permite mantener las aplicaciones libres de funciones no utilizadas.<br>• Aumentar la tolerancia a fallas y el aislamiento de fallas: Si un servicio falla, no comprometerá los otros servicios de la aplicación.<br>• Aumentar la velocidad de implementación: Permite que diferentes equipos puedan trabajar en diferentes módulos simultáneamente, sin esperar a que otros grupos completen proyectos antes de que puedan avanzar.<br>• Permitir una mayor escalabilidad y flexibilidad: Otorga a las organizaciones la libertad de aplicar nuevas pilas de tecnología en un solo servicio y permite a los equipos escalar cada servicio de forma horizontal e independiente. | • Ser más complejos que la mayoría de las aplicaciones monolíticas: Los microservicios involucran muchas más partes móviles que las aplicaciones tradicionales, lo que requiere un esfuerzo enorme, una planificación cuidadosa y automatizaciones aplicadas estratégicamente para garantizar que los procesos de comunicación, monitoreo, prueba e implementación se ejecuten sin problemas.<br>• Requiere cambios culturales: Cualquier iniciativa de microservicios requiere que las organizaciones modifiquen su cultura interna. Antes de iniciar el proceso de migración, ya debería existir una cultura Agile y DevOps madura, ya que cada equipo de microservicios opera como una empresa independiente. Esta estructura permite a los equipos trabajar en pequeños grupos o de forma independiente y también ofrecer más control y mayor productividad.<br>• Más caros que las aplicaciones monolíticas: Los servicios deberán comunicarse entre sí, lo que implica un gran volumen de llamadas remotas. Esto puede aumentar la latencia de la red y los costos de procesamiento más allá de lo que podría esperar pagar al usar arquitecturas tradicionales.<br>• Presenta amenazas a la seguridad: Los microservicios conllevan algunos desafíos de seguridad importantes debido al fuerte aumento del volumen de datos intercambiados entre módulos. Debido a que está trabajando con varios contenedores pequeños, se expone más de su sistema a la red, lo que a su vez implica una vulnerabilidad ante a posibles atacantes. |

Nota: Elaboración Propia

### 1.1.2. Análisis del rol específico de la arquitectura de software

El empleo de la arquitectura tiene su utilidad en diferentes etapas del ciclo de vida del software, desde la fase de diseño y desarrollo, como en la ejecución de mantenimiento y actualización. En la tabla 1.5, se muestra el rol específico de la arquitectura de software.

TABLA 1.5
Rol específico de la arquitectura de software.

| | |
|---|---|
| Proceso de desarrollo de nuevos productos | ● La denominada arquitectura de software es un puente de comunicación construido entre el mundo real y el campo de la informática. Las funciones principales de este proceso son los siguientes:<br>● Lograr objetivos comerciales superiores: en esencia, la arquitectura del software a menudo tiene la responsabilidad de llevar a cabo la planificación general correspondiente para completar los objetivos comerciales.<br>● Proporcionar restricciones y orientación efectivas para el trabajo de desarrollo técnico posterior, dependiendo del plan de diseño de la arquitectura de software, el cual cambia según los requisitos relacionados con el negocio y la dirección hacia la cual está orientada a la tecnología.<br>● Mejorar de manera efectiva la calidad de los nuevos productos.<br>● Implementar el desarrollo del producto, así como la entrega incremental con la ayuda de las iteraciones correspondientes. |
| Línea de productos de software | Se refiere a una arquitectura general correspondiente diseñada con base en una serie de productos dentro de una empresa u organización.<br>Existen similitudes en una serie de productos de este tipo que se pueden implementar una arquitectura idéntica o similar con el fin de maximizar la productividad.<br>La arquitectura de la línea de productos de software tiene principalmente las siguientes funciones:<br>● Considerar una serie de cambios claramente permitidos.<br>● Documentar en su totalidad los procesos.<br>● Establecer una guía del creador del producto correspondiente. Por ejemplo, describir el proceso de la arquitectura. |
| Proceso de mantenimie del software | Las principales fuentes del trabajo de mantenimiento de software correspondiente son los errores y los cambios en la demanda.<br>El corregir un error y agregar una nueva función suele implicar una cadena de colaboración de módulos en el enlace de la arquitectura. En este caso, la arquitectura del software es indicada para el mantenimiento. |
| Proceso de actualizació del software | La arquitectura de software corresponde a la modificación y refactorización continua del sistema de software. La implementación de la refactorización se da principalmente en dos situaciones:<br>● En los casos de arquitecturas particularmente caóticas, se implementan cambios pequeños que afectarán a todo el cuerpo.<br>● En las situaciones donde los nuevos requisitos no se adapten, se debe recurrir a la reingeniería, la cual implementará una actualización de software fundamental que posea elementos del software anterior. |

Nota: Elaboración Propia

## 1.2. GraphQL

GraphQL es un lenguaje de consulta para API que comenzó en Facebook antes de ser de código abierto en 2016. La simplificación de las consultas y respuestas proporcionadas por GraphQL se originó en el uso dentro de aplicaciones móviles, pero tiene una utilidad general para simplificar el uso de API complejas al tiempo que reduce la cantidad de datos devueltos (Malhotra, 2019).

El proyecto GraphQL se creó para resolver un problema específico. En 2011, debido a la interacción de los usuarios con las aplicaciones desde sus navegadores, se produjo una fuerte migración de servidores web a dispositivos móviles. En el caso de Facebook, esto requirió llamar a múltiples API que debían entregar distintos volúmenes de información desde múltiples ubicaciones diferentes en la aplicación. Para resolver este desafío, Facebook creó una forma de simplificar la entrega de información y reducir el tráfico total requerido (Maldonado, 2019). Originalmente, la tecnología de GraphQL se utilizó en su totalidad por el equipo móvil de iOS de Facebook y luego se propagó al resto de las interfaces móviles. A partir de 2015, gracias al anuncio de Facebook acerca de la implementación de GraphQL, se produjo una rápida adopción de esta tecnología a nivel mundial.

Al aplicar GraphQL, los desarrolladores de clientes no necesitan tratar directamente con la interfaz API de *backend*, sino que solicitan los datos de destino declarando la estructura de datos requerida en GraphQL (Rios 2020). La introducción de GraphQL puede mejorar efectivamente la eficiencia de desarrollo del cliente, de modo que el desarrollador del cliente ya no se limita a la interfaz proporcionada por el servidor, sino que puede personalizar libremente el contenido de datos requerido de acuerdo con las necesidades comerciales. GraphQL ha sido adoptado por muchos equipos de desarrollo, por ejemplo, Twitter, GitHub, Coursera y muchos más (Chicaiza, 2020).

GraphQL se utiliza como lenguaje de consulta. Los desarrolladores inician solicitudes al servicio GraphQL y obtienen los datos correspondientes declarando el formato de datos que necesitan. En el proceso tradicional de comunicación cliente / servidor, el cliente envía una solicitud de red al servidor, y el servidor realiza las operaciones correspondientes y devuelve los datos después de recibir la solicitud. La figura 3 muestra la estructura del sistema sin GraphQL:

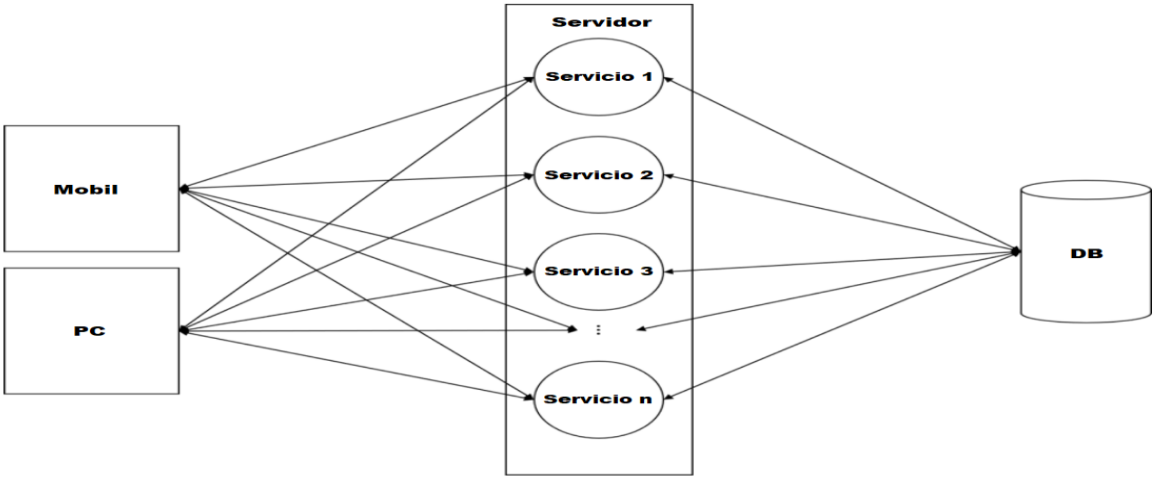


Figura 3: Estructura del sistema sin GraphQL
Fuente Elaboración Propia basado en (Hartina et al. 2018)

Luego de la introducción de GraphQL, el cliente ya no se comunica directamente con el servidor, sino que obtiene datos a través del servicio GraphQL. La figura 4 muestra la estructura del sistema de introducción de GraphQL:

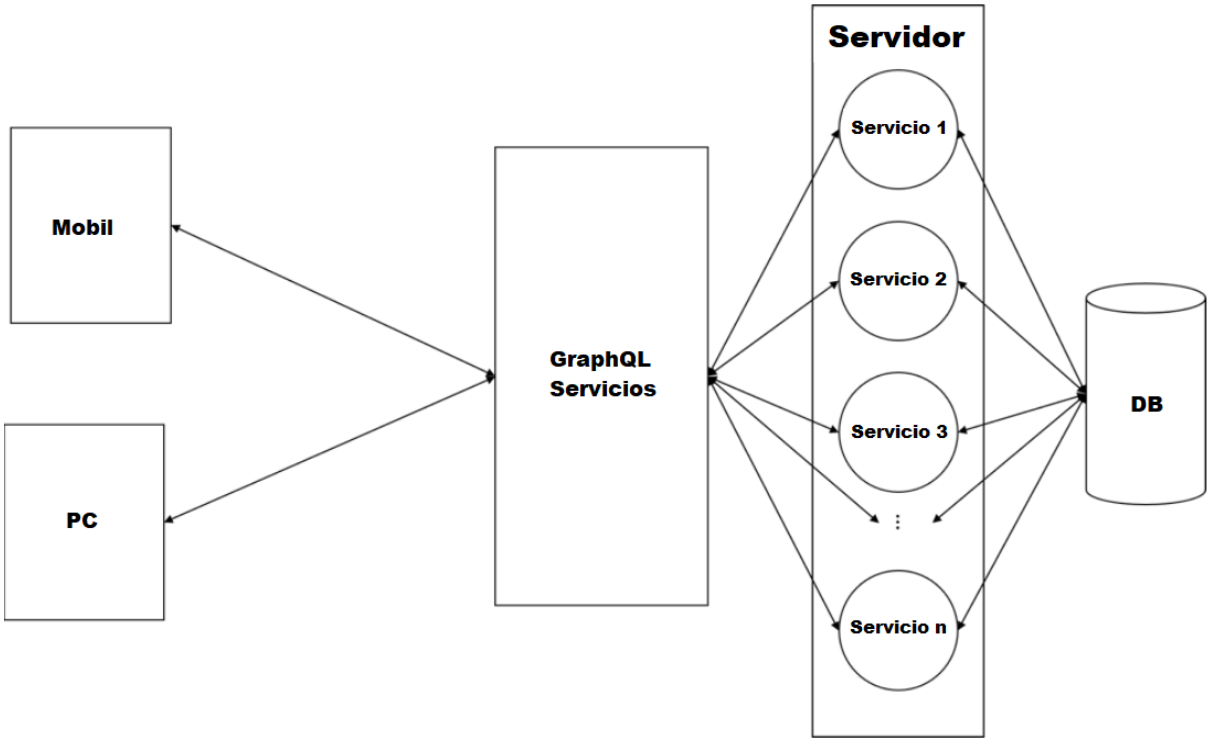


Figura 4: Estructura del sistema con GraphQL
Fuente: Elaboración Propia según (Hartina et al. 2018)

La figura 5 muestra el antes y el después de la introducción de GraphQL al solicitar datos específicos de varios servicios:

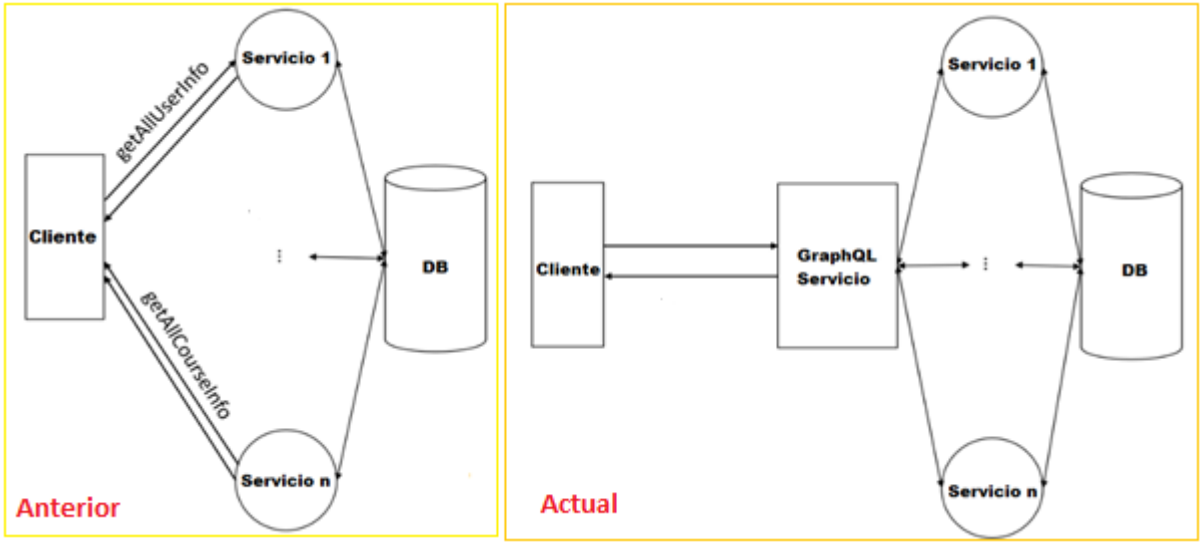


Figura 5: Anterior y el actual de la introducción de GraphQL
Fuente: Elaboración Propia según (Hartina et al. 2018)

En comparación con el modo de comunicación cliente / servidor tradicional, la introducción de GraphQL agrega una capa intermedia a todo el sistema, protegiendo la estructura de datos específica de los *frontend* y *backend*. Entre sus principales ventajas, según Maldonado (2019), se encuentran las siguientes:

- Personalización de los datos requeridos: El cliente puede personalizar la estructura de datos que desee y obtiene los datos deseados del servicio GraphQL al declarar la estructura de datos requerida en la solicitud. La estructura de datos devuelta por el servicio será exactamente la misma que la solicitud del cliente, lo cual reduce la transmisión de datos redundantes. Sin GraphQL, el formato de los datos devueltos no está controlado por el cliente.
- Solicitud única para obtener varios recursos: En GraphQL, el cliente ya no se preocupa por el origen de la solicitud, sino que escribe directamente los recursos necesarios en el campo de solicitud. Sin utilizar GraphQL, cuando es necesario obtener datos de diferentes servicios, los desarrolladores de clientes deben iniciar

solicitudes para diferentes servicios.

- Estructura es el documento: GraphQL tiene una interfaz de depuración amigable y los desarrolladores pueden consultar directamente los servicios proporcionados por GraphQL en esta interfaz. Esta estructura de datos es la forma en que se muestra el documento, lo que permite a los desarrolladores encontrar fácil y rápidamente los datos que necesitan.
- Función de suscripción: GraphQL proporciona una función de suscripción, que permite al cliente monitorear los cambios de datos. De este modo, el servicio GraphQL envía activamente los datos modificados al cliente y los muestra en la interfaz en tiempo real.

En resumen, GraphQL es una interfaz de programación de aplicaciones (API siglas en inglés) de código abierto relativamente nueva que combina un lenguaje de consulta, motor de ejecución y especificación de código abierto que definen la apariencia y función de la implementación GraphQL.

En la actualidad, GraphQL está cambiando la forma de crear aplicaciones cliente y API, ya que los desarrolladores de aplicaciones para el usuario pueden consultar libremente los datos que deseen. Por otro lado, los desarrolladores de aplicaciones en segundo plano pueden separar los requisitos del programa cliente de la arquitectura del sistema.

Algunas empresas utilizan GraphQL mediante la creación de una "capa" API GraphQL sobre sus servicios de *backend* existentes. Esto permite que los programas cliente obtengan el rendimiento y la eficiencia operativa que buscan y, al mismo tiempo, los equipos de *backend* mejoran, en caso de ser necesario, el sistema detrás de la capa GraphQL. Por lo general, estos cambios están optimizados para el rendimiento, lo que ayuda a garantizar que los programas que utilizan GraphQL se ejecuten de manera eficiente. Debido a las capacidades de abstracción proporcionadas por GraphQL, el equipo del sistema puede llevar a cabo estas mejoras sin dejar de cumplir con las especificaciones de nivel de API de GraphQL (Khan & Noor, 2020).

### 1.2.1. Origen de la tecnología GraphQL

Antes que apareciera GraphQL, las API más populares usaban una adaptación de REST, acrónimo de *Representational State Transfery,* el cual fue definido por Fielding

(2000), como un estilo arquitectónico en su tesis doctoral, REST es un enfoque integral para el diseño de software que utiliza las características básicas del protocolo HTTP de la web para trabajar con aplicaciones.

REST siendo una herramienta masifica plataformas como Facebook la empleaba, sin embargo, el responsable de Facebook News Feed fue Lee Byron, notó que faltaban grandes segmentos de datos en el *feed*, por ello juntamente con su equipo comenzaron a buscar una nueva forma de abordar la publicación de datos de News Feed. En ese momento, Facebook había lanzado una nueva tecnología, FQL (Facebook Query Language), que fue una derivación de SQL. A diferencia de SQL, que normalmente interactúa directamente con el motor de base de datos dado, FQL fue diseñado para realizar consultas en una capa de abstracción de código que representaba los datos de News Feed. Esta capa de abstracción de código unió varias partes del nivel de aplicación de Facebook para cumplir con las consultas FQL (Byron, 2015)

Por consiguiente, FQL abordó el problema del cuello de botella de la red, pero no consiguió abordar el problema de recursividad. Es decir, escribir consultas FQL recursivas fue difícil y los equipos de desarrollo que usaban FQL necesitaban tener al menos un miembro que tuviera un conocimiento profundo de su funcionamiento para que las operaciones del lado del servidor funcionaran. No existía mucho personal con este tipo de talento, ante un número limitado de desarrolladores que podían hacer el trabajo de optimización de FQL y la creciente complejidad y volumen de las consultas de *backend* creadas para soportar las demandas de News Feed, Byron decidió buscar una mejor manera, lo que requería que él y sus compañeros ingenieros cambiaran su forma de pensar sobre las estructuras de datos. Por consiguiente, se necesitaban alejarse de la conceptualización de conjuntos de datos como tablas hacia un tipo diferente de estructura de datos, el gráfico de objetos (Byron, 2015).

Aunque FQL permitió a los desarrolladores de *frontend* acceder más rápido a los datos de News Feed de Facebook, no resolvió un problema arquitectónico fundamental de clavija redonda y agujero cuadrado. Tanto los equipos del lado del cliente como del lado del servidor se sentían incómodos trabajando con FQL. Por consiguiente, los desarrolladores del lado del cliente y los desarrolladores del lado del servidor hablaron sobre los datos en términos de un gráfico de objetos; sin embargo, FQL era

fundamentalmente un concepto de tabulación (Byron, 2015).

En consecuencia, surgió la idea de GraphQL, construido desde cero por sus cocreadores Lee Byron, Dan Schafer y Nick Schrock, para ser una API y un lenguaje de consulta para gráficos de objetos. GraphQL está diseñado para crear API que admita modelos que se puedan recuperar mediante una sola solicitud del servidor y para admitir la recursividad declarativa desde una sola consulta. La recursividad declarativa significa que los desarrolladores pueden crear una única consulta que diga, efectivamente, muéstrame una lista de películas según el título, la fecha de lanzamiento, los directores y los actores y muéstrame quién conoce y a quién le gusta cada director (ver Figura 6). El desarrollador puede profundizar en el gráfico si así lo desea (Byron, 2015).

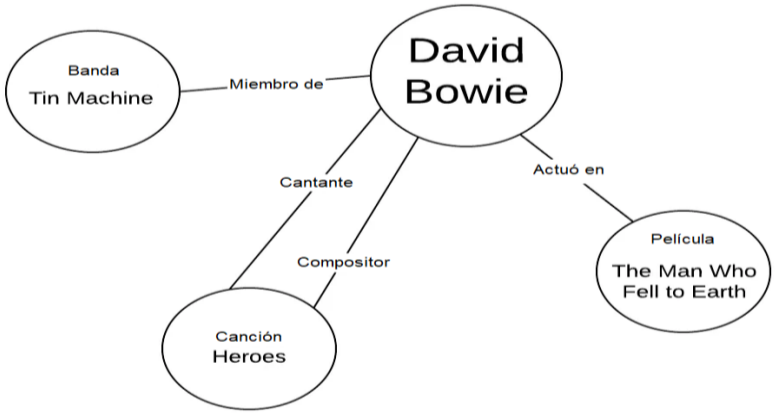


Figura 6: Gráfico de objetos estructura los datos de acuerdo con nodos y bordes
Autor: Elaboración Propia según (Byron 2020)

El cumplimiento de la consulta se realiza entre bastidores. El desarrollador no tiene que hacer combinaciones sofisticadas como aquellas que son típicas al trabajar con tablas en una base de datos relacional. El gráfico de objetos es el bloque de construcción sobre el que se ejecutan las consultas.

Es importante mencionar que GraphQL está destinado a proporcionar una forma de recuperar datos estructurados y recursivos dentro de la restricción de una sola solicitud al servidor. En otras palabras, una vez que se realiza la declaración recursiva

inicial, no es necesario realizar ninguna otra acción. Además, otro punto importante sobre GraphQL es que es solo una especificación, así como SQL es solo una especificación. GraphQL en sí mismo no es una API ni es un producto. La implementación de alguna tecnología para respaldar esa especificación es otra actividad en conjunto. La especificación es el mecanismo que permite a cualquiera trabajar con una API GraphQL independientemente de la tecnología subyacente y el lenguaje utilizado para publicar datos a través de la API. GraphQL es independiente de la plataforma y, de hecho, hay varias implementaciones para una variedad de plataformas. Sin embargo, GraphQL es una especificación de código abierto para implementar API compatible con GraphQL en un marco tecnológico específico. Por ejemplo, la implementación utilizada en esta serie es Apollo Server, el cual funciona con node.js. También existen implementaciones en C# / .NET, Go, Ruby, Java y Python, entre otros (Byron, 2015).

### *1.2.2. Principales lenguajes de programación utilizados con GraphQL*

GraphQL al ser un lenguaje que trabaja en el lado de cliente tiene la flexibilidad de ser relacionado con cualquier otro tipo de lenguaje de programación, ya que existen bibliotecas disponibles para distintos lenguajes de programación, por ejemplo, Go, Java, JavaScript, PHP, Python o Ruby, lo que otorgan una gran libertad al usuario a la hora de elegir el lenguaje, así que en realidad se puede decir que GraphQL, es interpretativo con casi todos los lenguajes de programación modernos existentes, sin embargo la elección de estos, para esta investigación, se realizó con la ayuda de una de las principales plataformas de desarrollo colaborativo como lo es GitHub, que gestiona y presentan las valoraciones y aceptación por parte de la comunidad de programadores y usuarios, que emplean el lenguaje de consulta GraphQL y se comparó con lo que se obtuvo de la revisión bibliográfica, de la coincidencia resultante se construyó la tabla 1.6.

TABLA 1.6
Principales lenguajes de programación utilizados con GraphQL.

| **Lenguajes de programación** | | | |
|---|---|---|---|
| C# / .NET | Go | Kotlin | Ruby |
| Clojure | Groovy | Perl | Rust |
| Elixir | Java | PHP | Scala |
| Erlang | JavaScript | Python | Swift |

Nota: Elaboración propia, adaptado de GitHub (2020), Fecha de revisión 03/03/2021

## 1.3. Esquema para pruebas de software.

El presente trabajo que tiene como finalidad realizar pruebas con diversas arquitecturas orientadas a servicios, por ello se considera lo planteado por Wohlin et al. (2012), quienes establecen una secuencia de pasos formales para la diseño, formulación, ejecución, operación y análisis de los resultados de las pruebas a software. Esta propuesta para la experimentación en la ingeniera de software plantea como definir estrategias completas para definir casos de estudios, iniciado con la planificación, definiendo el alcance y el contexto en que se desarrollaron los experimentos, así como también concebir el protocolo de recolección de data (observaciones o métricas) y análisis o evaluación con métodos dependiendo de la naturaleza de la misma (cualitativa o cuantitativa), garantizando en todo momento la validación lo que le dará confiabilidad a la ejecución u operación de las pruebas.

Por último, los autores, propone las características y estructura del reporte que se debe usar para discutir y presentar los resultados obtenidos, considerando las variables e hipótesis definas en las fases iníciales. Para esta investigación se combinó esta propuesta de experimentación con norma ISO/IEC 25023, específicamente en la definición de métrica de calidad a medir, delimitando el alcance del proyecto a evaluar las arquitecturas por la eficiencia de su desempeño.

En función de lo expuesto anteriormente, el esquema para desarrollar las pruebas a las arquitecturas orientadas a servicios, siguiendo la recomendación de Wohlinetal. (2012), se sería el siguiente:

- Definición del Alcance.
- Planificación.
- Operación y ejecución.
- Análisis y presentación de resultados.

## 1.4. ISO/IEC 25023

La normativa internacional ISO / IEC 25023: 2016 define medidas de calidad para evaluar cuantitativamente la calidad del sistema y del producto de software en términos de características y subcaracterísticas definidas en la ISO / IEC 25010. Además, tiene disposiciones en común con la ISO / IEC 2503n y la ISO / IEC 2504n en lo que respecta a la satisfacción de las necesidades del usuario en relación con el producto de software o la calidad del sistema. Se establece un conjunto básico de medidas de calidad para cada característica y subcaracterística; asimismo, presenta una explicación detallada sobre cómo aplicar las medidas de calidad de los productos y sistemas de software (Organización Internacional de Normalización, 2016)

Se pueden mencionar dos desventajas principales de esta normativa. En primer lugar, no asigna rangos de valores de las medidas a niveles nominales o grados de cumplimiento porque estos valores se definen con base a la naturaleza del sistema, producto o parte del producto y dependen de factores como la categoría del software, nivel de integridad y necesidades de usuarios. En segundo lugar, algunos atributos pueden tener un rango de valores deseables que no depende de las necesidades específicas del usuario sino de factores genéricos, por ejemplo, factores cognitivos humanos (Organización Internacional de Normalización, 2016)

Los principales usuarios de ISO / IEC 25023: 2016 son aquellos que llevan a cabo actividades de evaluación y especificación de requisitos de calidad en las tres siguientes fases: fase de desarrollo, en la que se incluye el análisis de requisitos, la especificación del diseño, la codificación y las pruebas mediante la aceptación durante el proceso del ciclo de vida; fase de gestión de la calidad, en la cual se lleva a cabo un examen sistemático del producto de software o sistema informático, por ejemplo, como parte de la garantía de calidad, el control de calidad y la certificación de calidad; y la fase de mantenimiento, la cual proporciona valor en la mejora del producto o sistema software basado en la medición de la calidad (Organización Internacional de Normalización, 2016)

La norma ISO/IEC 25023 indica que las características de calidad en el modelo del producto que por lo menos se deben evaluar son:

- **Adecuación Funcional**: referente a completitud, corrección y pertinencia funcional.

- **Eficiencia de desempeño**: capacidad de un producto o sistema software de proporcionar un rendimiento apropiado, respecto a la cantidad recursos utilizados bajo determinadas condiciones.
- **Compatibilidad**: referente a medir la coexistencia, interoperabilidad.
- **Usabilidad:** referente a medir capacidad para reconocer su adecuación, capacidad de aprendizaje, capacidad para ser usado, protección contra errores de usuario, estética de la interfaz de usuario, accesibilidad.
- **Fiabilidad:** referente a medir madurez, disponibilidad, tolerancia a fallos, capacidad de recuperación.
- **Seguridad:** referente a medir confidencialidad, integridad, no repudio, responsabilidad y autenticidad.
- **Mantenibilidad:** referente a medir modularidad, reusabilidad, analizable, capacidad para ser modificado, capacidad para ser probado.
- **Portabilidad:** referente a medir adaptabilidad, capacidad para ser instalado y capacidad para ser remplazado.

Para efectos de esta investigación se consideró la característica eficiencia de desempeño, acotando el alcance a la subcaracterística, **Comportamiento Temporal** que es capacidad de un sistema software para proporcionar los tiempos de respuesta y procesamiento apropiados.

## 1.5. Herramientas tecnológicas

En este apartado se describen los aspectos más resaltantes de herramientas que se emplean para conformar parte activa de las arquitecturas a probar.

### 1.5.1. PostgreSQL

PostgreSQL tiene más de 15 años de desarrollo activo y una arquitectura probada que ha ganado una sólida reputación de fiabilidad e integridad de datos. Se ejecuta en los principales sistemas operativos que existen en la actualidad tales como Linux, Unix (Mac OS X, Solaris) y Windows.

Es un potente sistema de base de datos relacional de objetos de código abierto que

utiliza y amplía el lenguaje SQL combinado con muchas características que almacenan y escalan de forma segura las cargas de trabajo de datos más complicadas. Los orígenes de PostgreSQL se remontan a 1986 como parte del proyecto POSTGRES, el cual tiene más de 30 años de desarrollo activo en la plataforma central (PostgreSQL, 2021).

PostgreSQL tiene una sólida reputación por su arquitectura probada, confiabilidad, integridad de datos, conjunto de características sólidas, extensibilidad y la dedicación de la comunidad de código abierto detrás del software para ofrecer soluciones innovadoras y de alto rendimiento de manera consistente (PostgreSQL, 2021).

Las principales características de PostgreSQL, más importantes del software de base de datos son:

- Base de datos relacional.
- Orientada a objetos tales como la herencia de tablas y la gran cantidad de aplicaciones complementarias que se han desarrollado para la administración, diseño, migración, monitoreo, etc.
- Soporta los principales sistemas operativos (SO).

En la Tabla 1.7 se describe las ventajas y desventajas de las redes PostgreSQL

TABLA 1.7
Ventajas y Desventajas de PostgreSQL

| Ventajas | Desventajas |
|---|---|
| • Instalación y uso gratuito | • PostgreSQL está diseñado específicamente para ambientes con alto volumen de datos. |
| • Sistema disponible multiplataforma | • No presenta facilidad en comandos o sintaxis. |
| • Estabilidad y confiabilidad | • No cuenta con soporte oficial. |
| • Herramienta gráfica | • Es relativamente lento en inserciones y actualizaciones en bases de datos pequeños. |
| • Robustez y fiabilidad | • Está diseñado para ambientes de alto volumen. |

Nota: Elaboración propia

### *1.5.2. Lenguaje de programación JavaScript*

JavaScript dio vida a las páginas web; los programas en este idioma se llaman *scripts* y pueden escribirse directamente en el HTML de la página web y así ejecutarse a medida que se carga la página, los *scripts* se proporcionan y ejecutan como texto sin

formato. No necesitan preparación especial o compilación para ejecutarse.

### *1.5.3. GraphQL API Apollo Server*

Es un marco de programación, en lenguaje JavaScript, altamente escalable y coordinado que se ha implementado en clústeres de producción en Microsoft para programar miles de cálculos con millones de tareas de manera eficiente y efectiva en decenas de miles de máquinas diariamente. El marco toma decisiones de programación de manera distribuida utilizando información de clúster global a través de un mecanismo poco coordinado. Cada decisión de programación considera la disponibilidad futura de recursos y optimiza varios factores de rendimiento y del sistema juntos en un solo modelo unificado (Boutin, y otros, 2014)

### *1.5.4. Framework React*

React es un marco popular para la construcción de aplicaciones web, en lenguaje JavaScript. Las aplicaciones React están escritas en un estilo declarativo y orientado a objetos, y constan de componentes organizados en una estructura de árbol. Cada componente tiene un conjunto de propiedades que representan parámetros de entrada, un estado que consta de valores que pueden variar con el tiempo y un método de representación que especifica de forma declarativa los subcomponentes del componente. El concepto de reconciliación de React determina el impacto de los cambios de estado y actualiza la interfaz de usuario de forma incremental mediante el montaje y desmontaje selectivo de subcomponentes. El marco de React invoca enlaces del ciclo de vida que permiten a los programadores adquirir y liberar los recursos necesarios para un componente. Estos mecanismos exhiben una complejidad considerable (Madsen, Lhotak, & Tip, 2020)

### *1.5.5. Lenguaje de programación Python*

Python es un lenguaje de programación interpretado, orientado a objetos, de alto nivel con semántica dinámica, con estructuras de datos integradas de alto nivel, las cuales se combinan con tipiado y enlace dinámico. Esto lo hace muy atractivo para el desarrollo rápido de aplicaciones, así como para su uso como lenguaje de *scripts* para conectar componentes existentes. La sintaxis enfatiza la legibilidad y, por lo cual, reduce el costo de mantenimiento del programa. Asimismo, Python admite módulos y paquetes,

lo que fomenta la modularidad del programa y la reutilización del código (GraphQL).

### *1.5.6. Herramienta de servidor Graphene*

Graphene-Python es una biblioteca para construir APIs GraphQL en Python fácilmente, su objetivo principal es proporcionar una API simple pero ampliable para facilitar la vida de los desarrolladores.

Para poder tener un proceso de desarrollo rápido, es fundamental reutilizar la mayor cantidad de código posible. Graphene-Python ofrece integraciones con diferentes frameworks, estas integraciones incluyen (Github, 2021):

- **Django:** Graphene-Django
- **SQLAlchemy:** Grafeno-SQLAlchemy
- **Google App Engine**: Graphene-GAE

### *1.5.7. Herramienta de cliente GQL*

Este es un cliente GraphQL para Python 3.6+. Funciona muy bien con graphene, graphql-core, graphql-js y cualquier otra implementación de GraphQL compatible con la especificación.

Las principales características de GQL son (Github, 2021):

- Ejecutar consultas GraphQL utilizando diferentes protocolos (http, websockets)
- Posibilidad de validar las consultas localmente usando un esquema GraphQL proporcionado localmente u obtenido del *backend* usando una consulta de inspección
- Admite consultas, mutaciones y suscripciones de GraphQL
- Admite el uso sincronizado o asincrónico, lo que permite solicitudes simultáneas
- Admite la carga de archivos
- Script gql-cli para ejecutar consultas GraphQL desde la línea de comando
- Módulo DSL para componer consultas GraphQL dinámicamente

### *1.5.8. Framework Django*

Django es un framework web Python de alto nivel que fomenta un desarrollo rápido y un diseño limpio y pragmático. Creado por desarrolladores experimentados, se

encarga de gran parte de la molestia del desarrollo web, por lo que puede concentrarse en escribir su aplicación sin necesidad de reinventar la rueda. Es gratis y de código abierto.

### *1.5.9. Lenguaje de programación Go*

El lenguaje de programación Go, fue una tendencia entre los programadores, en su lanzamiento oficial fue a finales de 2009, con una primera versión estable en 2012, por lo que es relativamente nuevo. Fue desarrollado por Google y se caracteriza por su desempeño y por propiedades que lo nominan a este lenguaje de programación como el enésimo sustituto natural de C (Keepcoding Tech School, 2018)

### *1.5.10. Herramienta de servidor 99designs/gqlgenç*

EL gqlgen es una biblioteca de Go para construir servidores GraphQL sin ningún problema. Se basa en un primer enfoque de esquema, puede definir su API utilizando el lenguaje de definición de esquemas GraphQL, prioriza la seguridad y habilita Codegen: genera los bits aburridos, para que pueda concentrarse en crear su aplicación rápidamente (Github, 2021).

### *1.5.11 Herramienta de cliente machinebox/GraphQL*

Machine Box, con base en GO, pone capacidades de aprendizaje automático de vanguardia en los contenedores de Docker para que desarrolladores como usted puedan incorporar fácilmente el procesamiento del lenguaje natural, la detección facial, el reconocimiento de objetos, etc. En sus propias aplicaciones muy rápidamente.

Las cajas están diseñadas para escalar, por lo que cuando su aplicación realmente despegue, simplemente agregue más cajas horizontalmente, hasta el infinito y más allá. Ah, y es mucho más barato que cualquiera de los servicios en la nube (y podrían ser mejores) y sus datos no salen de su infraestructura. (Veritone, Inc., 2021)

Machinebox/GraphQL es un cliente GraphQL de bajo nivel para Go que posee las siguientes características.

- API simple y familiar
- Respeta el contexto Tiempos de espera y cancelación de contexto
- Cree y ejecute cualquier tipo de solicitud GraphQL

- Utilice tipos de Go fuertes para los datos de respuesta
- Use variables y cargue archivos
- Manejo simple de errores

### *1.5.12 Docker*

Usa el kernel de Linux y las funciones Cgroups y name spaces, para segregar los procesos, de modo que puedan ejecutarse de manera independiente. El propósito de los contenedores es esta independencia: la capacidad de ejecutar varios procesos y aplicaciones por separado para hacer un mejor uso de su infraestructura y, al mismo tiempo, conservar la seguridad que tendría con sistemas separados.

Un Docker, ofrecen un modelo de implementación basado en imágenes. Consisten compartir una aplicación, conjunto de servicios, con todas sus dependencias en varios entornos. Docker también automatiza la implementación de la aplicación (o conjuntos combinados de procesos que constituyen una aplicación) en este entorno de contenedores. (Red Hat, Inc., 2021)

# CAPÍTULO 2

## Diseño del Experimento

En el presente capítulo se plasma las etapas de la fase experimental de la investigación. Lo que incluye el alcance, planificación y ejecución del experimento concedido para evaluar y comparar arquitecturas bajo el concepto de microservicios. Esto, con el marco referencial de la norma ISO/IEC 25023, de la cual se tomaron métricas cuantitativas de desempeño.

### 2.1 Alcance.

Los experimentos con software conforman uno de los principales insumos de conocimientos para entender el porqué de determinados resultados en la implantación de estos, ya que permiten identificar, valorar y comprender variables que entran en juego en el rendimiento del software y de las conexiones entre ellos. Sin embargo, es fundamental que a cada experimento a realizar se le delimite su campo de acción, para que la interpretación de los resultados y las conclusiones estén dentro de un contexto conocido. Tomando en cuenta lo antes descrito, a continuación, se desglosa el alcance del experimento planteado en esta investigación, en términos de su objeto de estudio, propósito, perspectiva, enfoque de calidad y contexto.

#### 2.1.1 Definición de la meta.

La formulación del experimento se inicia con la visión de lograr hacer comparaciones entre arquitecturas de software, midiendo su rendimiento con métricas cuantitativas de tiempo, ante la demanda de servicios accionada por el lenguaje de consulta GraphQL.

##### *2.1.1.1 Objeto de estudio.*

Comparar y verificar la eficiencia de las arquitecturas orientadas a microservicios, utilizando como referencia el estándar descrito en el Modelo de Calidad del Producto Software (en su apartado de Calidad Interna y Externa) de la norma ISO/IEC 25023. El citado modelo, plantea 8 características de calidad, de las cuales para este estudio se considera sólo una, la *“Eficiencia en el Desempeño”*, la cual fue cuantificada mediante la medición de variables asociadas

a la sub-característica Comportamiento Temporal, específicamente se tomaron los registros del Tiempo de Espera y Rendimiento.

#### *2.1.1.2 Propósito.*

El experimento ejecutado tuvo como propósito determinar la Eficiencia de Desempeño de cada una de las arquitecturas preestablecidas orientadas a microservicios, ante las exigencias de servicios demandados por instrucciones bajo el lenguaje de consulta GraphQL.

#### *2.1.1.3 Perspectiva.*

Desde el punto de vista del investigador se estimó conocer cuáles son las combinaciones en la función de *frontend* y otra en *backend*, que registra el mejor desempeño ante la demanda de servicios empleando el lenguaje de consulta GraphQL. El resultado de este experimento deja un precedente de evaluación de software con registros de métricas ajustados al estándar ISO/IEC 25023.

#### *2.1.1.4 Enfoque de calidad.*

La conceptualización de este experimento se concibió bajo los estándares de la norma ISO/IEC 25023, los mismos orientados a la aplicación de las mejores prácticas de aseguramiento la calidad para los productos de software, por ello, se incluyen la medición de métricas asociadas a: los requisitos de calidad externa (que se utilizan para la verificación y validación técnica del producto) y a los requisitos de calidad interna (que se utilizan para verificar el producto a lo largo de las distintas etapas del desarrollo, y pueden utilizarse también para definir estrategias y criterios de evaluación y verificación). Las métricas de interés en la investigación son:

- Tiempo de espera, cuyos registros responden a la interrogante: ¿Qué arquitectura de software es más eficiencia en términos de tiempo de espera?

- Rendimiento, cuyos registros responde a la interrogante: ¿Qué arquitectura de software es más eficiente en términos de tareas procesadas por unidad de tiempo?

#### *2.1.1.5 Contexto.*

El experimento se realizó bajo un ambiente controlado, este se delimitó a un solo equipo computacional, en el cual se instaló los elementos informáticos sometidos a la comparación; lo que permitió obtener 9 combinaciones distintas de arquitecturas seleccionados para la investigación, además se dispuso de una base de datos con registros, a la cual se le aplicaron las consultas e inserciones de registros de información, contempladas en los casos de usos.

### 2.1.2 Resumen del alcance.

El alcance del experimento se centró en determinar y comparar la eficiencia de varias arquitecturas orientadas a microservicios, todas con un factor en común que fue utilizar el lenguaje de consultas GraphQL. En relación con el marco de referencia, se mantuvo como guía la norma ISO/IEC 25023 y la perspectiva del investigador, en registrar la eficiencia en el desempeño de cada arquitectura, considerando variables en función del tiempo.

## 2.2 Planificación.

Para lograr consolidar la planificación del experimento, se seleccionó un contexto acorde con el fin de este, la definición de variables dependientes e independientes, así como la selección de las herramientas informáticas que conformaron las arquitecturas a evaluar. En función de lo anterior, se realizó el diseño del experimento y se instrumentó para poder hacer la evaluación y validación necesaria de los resultados obtenidos.

### 2.2.1 Selección de Contexto.

El experimento se realizó en un ambiente controlado cuyas características se muestran en la tabla 2.1. El equipo seleccionado cumple con los requerimientos mínimos para la ejecución del experimento, ya que permite la instalación de todos los elementos informáticos que conforman las 9 arquitecturas sometidas a la comparación.

TABLA 2.1
Especificaciones de hardware

| Componentes | Detalle |
|---|---|
| Procesador | Intel core i7 – 5500U 2.40GHz |
| RAM | 16 GB |
| Disco duro | 450 GB HDD |
| Sistema Operativo | Ubuntu 18.04 |

Nota: Elaboración propia

Para la ejecución de los experimentos, se generó una base de datos conformada por tres tablas indexadas que se muestran en la figura 7, con información hipotética sólo para efectos del estudio. La primera tabla está diseñada para contener información sobreidentidadesy está identificada como usuarios, cuyo nombre de la tabla es *users;* la segunda mantiene registros de las publicaciones de dichos usuarios, esta tabla se identifica como *post;* y la terceraque posee los comentarios de dichas publicaciones, es reconocida como *comments*.

Para las consultas diseñadas en el experimento, cada tabla de la base de datos llegó a contener diferentes cantidades de registros, todos asociados a los incorporados en la tabla principal *users*, estas cantidades fueron: 1, 1.000, 25.000, 50.000 y 100.000 usuarios.

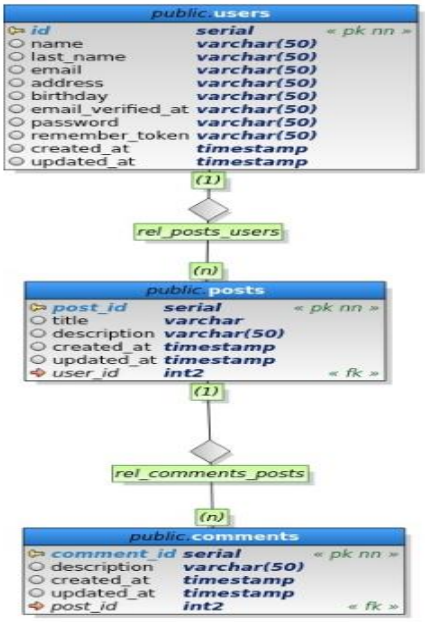


Figura 7: Diseño de la Base de Datos.
Fuente: Elaboración propia

En relación con las amenazas que se pudieron haber presentado, previamente se identificaron como riesgos: Interrupciones de las corridas, errores en los registros de los tiempos de respuestas de las arquitecturas, y sesgos en la interpretación de los datos por cambios en los criterios de observación. En vista de estas situaciones, se emplearon acciones de mitigación como:

- Se estimó controlar y monitorear los eventos en cada corrida, de tal modo que se pudo prevenir o identificar interrupciones de las corridas, como consecuencia de alguna falla en el ambiente de prueba.
- Se utilizaron librerías propias de los lenguajes de programación, probadas previamente, para la medición del tiempo.
- Se contempló repetir 3 veces cada corrida, para promediar los tiempos y registrarlo como el resultado obtenido, lo que estadísticamente es más representativo en lugar de evaluar un valor puntual; esto atenuando algún sesgo inducido en las mediciones.
- Se tuvo previsto mantener los mismos criterios de observación evitando sesgos en el análisis y en la interpretación de los resultados.

### 2.2.2 Formulación de la Hipótesis.

Desde el punto de vista académico se visualizó que el experimento fuese un aporte al conocimiento en el campo de investigación sobre las aplicaciones web, en este caso en particular sobre la eficiencia y compatibilidad de dichas aplicaciones con el lenguaje de consulta GraphQl, por lo tanto, se establecen las siguientes hipótesis:

- Hipótesis Nula ($H_0$): Todas las arquitecturas orientadas a microservicios, utilizando el lenguaje de consultas GraphQL, tienen la misma eficiencia.
- Hipótesis Alternativa ($H_1$): Es posible desarrollar una arquitectura orientada a microservicios, utilizando el lenguaje de consultas GraphQL, que sea más eficiente que otras.

Para la validación de ambas hipótesis se compararon las medias de los tiempos de respuestas obtenidos para cada una de las consultas.

### 2.2.3 Selección de Variables.

Para poder establecer las comparaciones entre las eficiencias de las arquitecturas, se declararon las siguientes variables dependientes e independientes.

Variables independientes son:

- Las arquitecturas de software orientadas a microservicios en el desarrollo Backend.
- Las arquitecturas de software orientadas a microservicios en el desarrollo Frontend.

Variable dependiente son:

- Tiempo medio de respuesta de arquitecturas orientadas a microservicio.
- Tiempo medio de rendimiento de arquitecturas orientadas a microservicio.

### 2.2.4 Selección de herramientas.

En el diseño del experimento, se ubicaron las herramientas que son compatibles con GraphQL, esto con la ayuda de una de las principales plataformas de desarrollo colaborativo como lo es GitHub, que se caracteriza por alojar códigos de proyectos típicamente de forma pública. Luego, como dicha plataforma permite que sus usuarios valoren las herramientas por el uso y versatilidad, se pudo seleccionar las aplicaciones que mostraban los mejores desempeños para el momento de la consulta, y así se confeccionaron las arquitecturas. De lo anterior, se obtuvo el conjunto de aplicativos listados en la tabla 2.2, estos presentan las más altas valoraciones y aceptación por parte de la comunidad de programadores y usuarios, que emplean el lenguaje de consulta GraphQL. Estas herramientas corresponden a tres lenguajes de programación en particular: JavaScript, Go y Python.

TABLA 2.2
Herramientas seleccionadas

| Nombre | Valoración en Github | Último Commit |
|---|---|---|
| Apollo Server | 11.1k | 02/10/2020 |
| React | 164k | 02/03/2021 |
| 99designs/gqlgen | 6k | 12/01/2021 |
| machinebox/graphql | 1k | 06/11/2018 |
| Graphene | 6k | 06/01/2021 |
| GQL | 1k | 14/12/2020 |
| Django | 55.9k | 02/03/2021 |

Nota: Elaboración propia, adaptado de GitHub (2020), Fecha de revisión 03/03/2021

Una vez seleccionadas las herramientas, se procedió a ubicar sus últimas versiones más estables para la fecha en que se inició la configuración del experimento. Luego de ser descargadas, se realizaron pruebas de funcionalidad por separado. En la tabla 2.3, se listan todas las herramientas que formaron parte de la experimentación, declarando la versión utilizada y en la columna de descripción se indica el rol del aplicativo en la arquitectura.

TABLA 2.3
Versiones de software, disponibles para el experimento

| Tecnología | Descripción | Versión |
|---|---|---|
| Javascript | Lenguaje de programación | Latest |
| Apollo Server | Servidor de Backend se lo utilizará como un GraphQL Provider, encargado de realizar consultas a la base de datos y visualizarla en formato de GraphQL | 2.19.2 |
| Dataloader | Herramienta utilizada para optimizar consultas de GraphQL a la base de datos | 2.0.0 |
| React | Framework de Frontend de JavaScript | 17.0.1 |
| Go | Lenguaje de programación | 1.14 |
| 99designs/gqlgen | Herramienta inspirada en Apollo Server que se utilizó para crear el servidor de GraphQL en el lenguaje de Go | 0.13.0 |
| vektah/dataloaden | Herramienta utilizada para optimizar consultas de GraphQL a la base de datos | 0.3.0 |
| machinebox/graphql | Herramienta utilizada como cliente de Graphql de Go | 0.2.2 |
| Python | Lenguaje de programación | 3.6.9 |
| Graphene | Librería utilizada para crear APIs GraphQL en Python | 2.1.8 |
| GQL | Librería cliente de Python que se utiliza para realizar consultas a API de GraphQL. | 2.0.0 |
| Django | Framework de Python el cual se utilizará para crear API como backend utilizando la librería Graphene | 3.1.5 |
| Postgres | Base de datos | 13.1 |

Nota: Elaboración propia

### 2.2.5 Diseño del Laboratorio Experimental.

En el laboratorio experimental se planeó evaluar 9 arquitecturas distintas, que resultan de la combinación de las herramientas seleccionadas y presentadas en la sección anterior. Se configuraron 3 arquitecturas en la cuales el *backend* y el *frontend* son bajo el mismo lenguaje y los 6 restantes son combinaciones, es decir el *backend* desarrollado en un lenguaje y el *frontend* en otro, partiendo de la premisa que todas las herramientas seleccionadas son compatibles con el lenguaje de consulta GraphQL, ya

que la finalidad del laboratorio experimental es reconocer cual combinación es más eficiente ante la demanda de servicio de este lenguaje. El mismo se diseñó en dos fases, la primera consiste en accionar consultas, a través de las 9 arquitecturas diseñadas, considerando 3 casos de uso (CU-01, CU-02 y CU-03), con este escenario se realizó cada corrida tres veces con 5 números diferentes de registros en las tablas de la base de datos, lo que generó 405 registros de tiempo para la variable: Tiempo medio de respuesta de arquitecturas orientadas a microservicios. Para la segunda fase se empleó un cuarto caso de uso (CU-04), con este escenario se realizó cada corrida tres veces con 4 números diferentes de registros en las tablas de la base de datos y se generaron 108 registros para la variable: Tiempo medio de rendimiento de arquitecturas orientadas a microservicios. En total, se obtienen 513 registros para su posterior análisis e interpretación, aplicando técnicas y métodos estadísticos.

Para la ejecución del laboratorio experimental, se diseñaron 9 arquitecturas, todas tienen en común tres elementos: primeramente, la base de datos, en segundo que son compatibles con el lenguaje GraphQL y por último el esquema de medición de calidad; sin embargo, la diferencia está en el *backend* y *frontend*. A continuación, se detallan estas arquitecturas (o *stack*):

***Stack 1*:** Está conformada por Apollo Server-JavaScript en el *backend* y React-JavaScript en el *frontend*, véase la figura 8.


Figura 8: Diseño de la Arquitectura a evaluar *Stack* 1
Fuente: Elaboración propia.

***Stack 2*:** Está conformada por Apollo Server-JavaScript en el *backend* y Django/GQL-Python en el *frontend*, véase la figura 9.


Figura 9: Diseño de la Arquitectura a evaluar *Stack* 2.
Fuente: Elaboración propia.

***Stack 3*:** Está conformada por Apollo Server-JavaScript en el *backend* y machinebox/graphql-Go en el *frontend*, véase la figura 10.

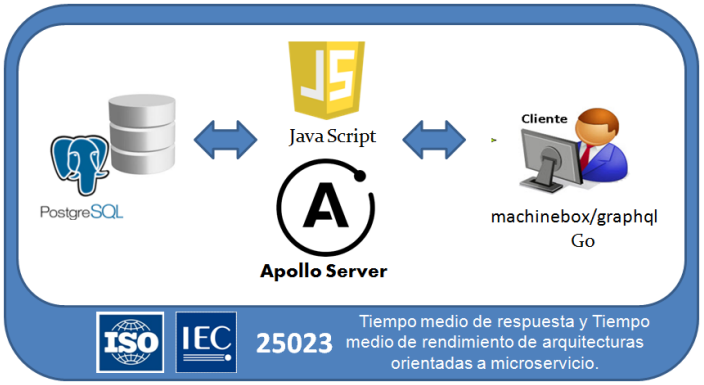


Figura 10: Diseño de la Arquitectura a evaluar *Stack*3.
Fuente: Elaboración propia.

***Stack 4*:** Está conformada por Django/Graphene-Python en el *backend* y React-JavaScript en el *frontend*, véase la figura 11.


Figura 11: Diseño de la Arquitectura a evaluar *Stack*4.
Fuente: Elaboración propia.

***Stack 5*:** Está conformada por Django/Graphene-Python en el *backend* y machinebox/graphql-Go en el *frontend*, véase la figura 12.


Figura 12: Diseño de la Arquitectura a evaluar *Stack*5.
Fuente: Elaboración propia.

***Stack 6*:** Está conformada por Django/Graphene-Python en el *backend* y Django/GQL-Python en el *frontend*, véase la figura 13.


Figura 13: Diseño de la Arquitectura a evaluar *Stack*6.
Fuente: Elaboración propia.

***Stack7*:** Está conformada por 99desings/gqlgen-Go en el *backend* y React-JavaScript en el *frontend*, véase la figura 14.


Figura 14: Diseño de la Arquitectura a evaluar *Stack*7.
Fuente: Elaboración propia.

***Stack8*:** Está conformada por 99desings/gqlgen-Go en el *backend* y Django/GQL-Python en el *frontend*, véase la figura 15.


Figura 15: Diseño de la Arquitectura a evaluar *Stack*8.
Fuente: Elaboración propia.

***Stack9*:** Está conformada por 99desings/gqlgen-Go en el *backend* y machinebox/graphql-Go en el *frontend*, véase la figura 16.


Figura 16: Diseño de la Arquitectura a evaluar *Stack*9.
Fuente: Elaboración propia.

En relación con los de casos de uso, estos se diseñaron en función de las posibles consultas a la base de datos, es decir, un primer caso de uso acceder a una sola tabla, luego un segundo caso para acceder a dos tablas, un tercer caso en el que acceden a las tres tablas de la base de datos; adicionalmente, se midió el rendimiento para el caso de incluir un registro en el base de datos.

Para el diseño de los casos de uso, se definió un único actor denominado: INVESTIGADOR, quien interactuó con el sistema al realizar las consultas, para obtener los tiempos de respuestas de las diversas arquitecturas.

A continuación, se detallan los 4 casos de uso empleados en el laboratorio

experimental.

**Caso de uso 1 (CU-01):** Consulta a una sola tabla.

Seleccionando este caso de uso, el INVESTIGADOR tomará el tiempo en que la arquitectura realice una consulta a una sola tabla de la base de datos, los detalles se indican en la tabla 2.4.

TABLA 2.4
Caso de uso CU-01

| **CU-01** | **Consulta de usuarios** |
|---|---|
| Autor | Paul Rocha |
| Fecha | 15/03/2021 |
| Descripción | Se realiza una consulta a la tabla "users", contenida en la base de datos. |
| Actores | INVESTIGADOR. |
| Precondiciones | Existe una base de datos configurada con tres tablas de datos, con registros pre-cargados. |
| Flujo normal | 1.- El investigador selecciona el número de usuarios a consultar.<br>2.- El INVESTIGADOR selecciona el tipo de caso: 1<br>3.- El INVESTIGADOR acciona la consulta.<br>3.- El sistema comprueba la validez de la consulta.<br>4.- El sistema hará la consulta del número de usuarios seleccionados, en la tabla "*users*" de la base de datos.<br>5.- El sistema publica el tiempo que tomó la consulta. |
| Flujo alternativo | 4.A.- Si los datos no son correctos, el sistema imprime un mensaje al INVESTIGADOR para informar el hallazgo y que la consulta no se pudo realizar, permitiéndole que repita la prueba. |

Nota: Elaboración propia

**Caso de uso 2 (CU-02)**: Consulta de dos tablas de la base de datos.

Seleccionando este caso de uso, el INVESTIGADOR tomará el tiempo en que la arquitectura realice una consulta a dos tablas de la base de datos, los detalles se indican en la tabla 2.5.

TABLA 2.5
Caso de uso CU-02

| CU-02 | Consulta de usuarios |
|---|---|
| Autor | PAUL ROCHA |
| Fecha | 15/03/2021 |
| Descripción | Se realiza una consulta a las tablas "*users*" y "*post*", contenidas en la base de datos. |
| Actores | INVESTIGADOR. |
| Precondiciones | Existe una base de datos configurada con tres tablas de datos, con registros pre-cargados. |
| Flujo normal | 1.- El investigador selecciona el número de usuarios a consultar.<br>2.- El INVESTIGADOR selecciona el tipo de caso: 2<br>3.- El INVESTIGADOR acciona la consulta.<br>3.- El sistema comprueba la validez de la consulta.<br>4.- El sistema hará la consulta del número de usuarios seleccionados, en las tablas "*users*" y *"post"* de la base de datos.<br>5.- El sistema publica el tiempo que tomó la consulta. |
| Flujo alternativo | 4.A.- Si los datos no son correctos, el sistema imprime un mensaje al INVESTIGADOR para informar el hallazgo y que la consulta no se pudo realizar, permitiéndole que repita la prueba. |
| Postcondiciones | Se realiza el registro del Tiempo de respuesta. |

Nota: Elaboración propia

**Caso de uso 3(CU-03):** Consulta de tres tablas de la base de datos.

Seleccionando este caso de uso, el INVESTIGADOR tomará el tiempo en que la arquitectura realice una consulta a tres tablas de la base de datos, los detalles se indican en la tabla 2.6.

TABLA 2.6
Caso de uso CU-03

| CU-03 | Consulta de usuarios |
|---|---|
| Autor | Paul Rocha |
| Fecha | 15/03/2021 |
| Descripción | Se realiza una consulta a las tablas "*users*", "*post*" y "*comments*", contenidas en la base de datos. |
| Actores | INVESTIGADOR. |
| Precondiciones | Existe una base de datos configurada con tres tablas de datos, con registros pre-cargados. |
| Flujo normal | 1.- El investigador selecciona el número de usuarios a consultar.<br>2.- El INVESTIGADOR selecciona el tipo de caso: 3<br>3.- El INVESTIGADOR acciona la consulta.<br>3.- El sistema comprueba la validez de la consulta.<br>4.- El sistema hará la consulta del número de usuarios seleccionados, en las tablas "users", "post" y "comments" de la base de datos.<br>5.- El sistema publica el tiempo que tomó la consulta. |
| Flujo alternativo | 4.A.- Si los datos no son correctos, el sistema imprime un mensaje al INVESTIGADOR para informar el hallazgo y que la consulta no se pudo realizar, permitiéndole que repita la prueba. |
| Postcondiciones | Se realiza el registro del Tiempo de respuesta. |

Nota: Elaboración propia

**Caso de uso 4(CU-04):** Inclusión de usuarios en la base de datos.

Seleccionando este caso de uso, el INVESTIGADOR tomará el tiempo que le tomará a la arquitectura hacer la inclusión de usuarios en la tabla "*users*", los detalles se indican en la tabla 2.7.

TABLA 2.7
Caso de uso CU-04

| **CU-04** | **Consulta de usuarios** |
|---|---|
| Autor | Paul Rocha |
| Fecha | 15/03/2021 |
| Descripción | Se realiza la inclusión de registros en la tabla "*users*", contenida en la base de datos. |
| Actores | INVESTIGADOR. |
| Precondiciones | Existe una base de datos configurada con tres tablas de datos, con registros pre-cargados. |
| Flujo normal | 1.- El investigador selecciona el número de usuarios a ingresar.<br>2.- El INVESTIGADOR acciona la inclusión.<br>3.- El sistema comprueba la validez de la inclusión.<br>4.- El sistema incluirá el número de usuarios seleccionados, en la tabla “users".<br>5.- El sistema publica el tiempo que tomó la consulta. |
| Flujo alternativo | 4.A.- Si los datos no son correctos, el sistema imprime un mensaje al INVESTIGADOR para informar el hallazgo y que la inclusión no se pudo realizar, permitiéndole que repita la prueba. |
| Postcondiciones | El número de registros de la tabla "*users*" incrementa. Se realiza el registro del tiempo de respuesta. |

Nota: Elaboración propia

### 2.2.6 Instrumentación.

Como se mencionó anteriormente, las dos métricas de calidad seleccionadas para evaluar las arquitecturas diseñadas derivan de la característica Eficiencia en el Desempeño de la norma ISO/IEC 25023, específicamente de la sub-característica comportamiento del tiempo. La primera métrica es Tiempo de Respuesta, que responde la interrogante: ¿Cuál es el tiempo que trascurre desde que se envía una instrucción, para que inicie un trabajo, hasta que lo completa?; para el monitoreo y registro de esta variable, se utilizaron librerías disponibles en cada uno de los aplicativos utilizados, que permiten tomar el tiempo de inicio y fin de las consultas. La segunda métrica el Rendimiento, da respuesta a la pregunta: ¿Cuántas tareas pueden ser procesadas por unidad de tiempo?; para lograr el cálculo, se toma la ecuación descrita en tabla 2.8, que

define a la métrica como el cociente de la cantidad de tareas ejecutadas por unidad de tiempo. Esta métrica, se calcula en la ejecución del cuarto caso de uso, en el cual se considera que la actividad a contabilizar es el ingreso de registro en el base de datos, cantidad que se divide por el tiempo total que dura la inclusión de todos los definidos para la corrida.

TABLA 2.8
Métricas para la característica de calidad Eficiencia en el Desempeño

| Sub característica | Métrica | Fase del ciclo de vida de calidad del producto | Propósito de la métrica de calidad | Método de aplicación | Fórmula | Valor deseado | Tipo de medida | Recursos utilizados |
|---|---|---|---|---|---|---|---|---|
| Comportamiento del tiempo | Tiempo de respuesta | Interna/Externa | ¿Cuál es el tiempo desde que se envía una instrucción, para que inicie un trabajo, hasta que lo completa? | Tomar el tiempo cuando se inicia un trabajo (A) y el tiempo en completar el trabajo (B) | X = B – A, A= Valor del Tiempo cuando se inicia un trabajo y B = Valor del Tiempo que transcurre en completar el trabajo | 0<=X<=1 El más cercano a 0 es el mejor. Donde el peor caso es >=15t. | X=Tiempo /Tiempo A=Tiempo B=Tiempo | Especificación de requerimientos, Código fuente, Desarrollador, Tester |
| | Rendimiento | | ¿Cuántas tareas pueden ser procesadas por unidad de tiempo? | Contar el número de tareas completadas (A) en un intervalo de tiempo (T) | X = A/T, A= Número de tareas completadas T = Intervalo de tiempo Dónde: T > 0 | X = A/T El más lejano a 0/t es el mejor. Donde el mejor caso es >=10/t | X= Contable/ Tiempo A=Contable, T= Tiempo | Especificación de requerimientos, Código fuente, Desarrollador, Tester |

Nota: Elaboración propia (Adaptado de ISO 2016)

### 2.2.7 Validación de la Evaluación

La validez de la evaluación se garantizó durante la realización del laboratorio experimental, se mantuvo bajo control y monitoreo las amenazas identificadas. Uno de los principales aspectos que favoreció a la validez, fue la existencia de un ambiente de baja complejidad en su arquitectura computacional; además, se aseguró que las versiones de los aplicativos fuesen compatibles entre sí, por ello fue posible hacer seguimiento a cada corrida de la prueba.

En cuanto al proceso de observación, lo llevo a cabo una sola persona el INVESTIGADOR, quien mantuvo los mismos criterios durante todo el laboratorio experimental. Para la medición de las métricas de calidad, se empleó el reloj de la arquitectura computacional, dado que ambas dependían del tiempo, lo que aseguró que la fuente de datos fuese la misma. Los datos generados se calcularon y se almacenaron bajo el mismo esquema, ya que se utilizaron librerías preexistentes y variables

predefinidas, ambos elementos fueron comunes para cada arquitectura evaluada; también es de considerar que cada una de las 513 corridas planificadas, se realizaron una a la vez, lo que permitió el control y el seguimiento con rigurosidad. Adicionalmente, se contó con un repositorio de eventos por cada corrida, en el cual se podía apreciar la secuencia de las tareas ejecutadas, así como también quedaba registrada cualquier eventualidad no deseada, habilitando la toma de decisiones de manera oportuna y lográndose validar los datos obtenidos luego de cada corrida culminada.

## 2.3 Operación.

En esta sección se describió cómo se llevó a cabo el laboratorio experimental, es decir se documentó la secuencia de acciones acometidas, antes, durante y posteriormente a la ejecución de las pruebas.

### 2.3.1 Preparación.

Culminada la planificación, se inició con la preparación y desarrollo del laboratorio experimental, lo que contempló las siguientes actividades:

- Descarga, instalación y configuración de cada aplicativo, según la versión seleccionada.
- Diseño, programación y configuración de los *frontend*, uno por cada lenguaje de programación seleccionado (Java script, Python y Go), lo que incluyó la generación de la interfaz para la ejecución de las pruebas. Se diseñaron dos pantallas, por lenguaje para este fin, la primera para ejecutar las corridas asociadas a los casos de usos 1,2 y 3. Y una segunda para el caso de uso número 4.
- Se programaron las rutinas para obtener los tiempos de ejecución de cada corrida, esto empleando librerías de los aplicativos. Se aseguró, que todos los tiempos fuesen registrados en la misma medida de tiempo, segundos (s).

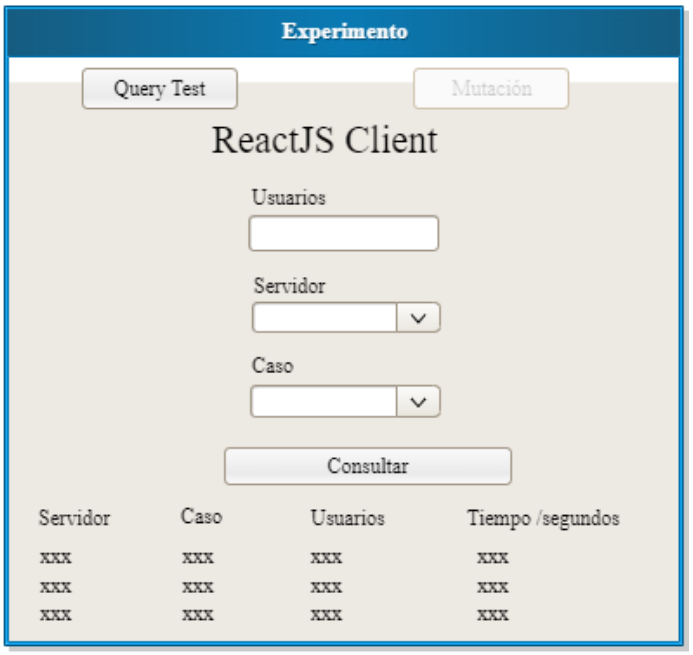


Figura 17: Diseño de la Pantalla para las consultas a la base de datos.
Fuente: Elaboración propia.


Figura 18:Diseño de la Pantalla para la inserción de registros en la base de datos.
Fuente: Elaboración propia.

### 2.3.2 Ejecución.

Para la ejecución se definieron dos fases. En la primera fase, se realizaron las corridas asociadas a los casos de uso: CU-01, CU-02 y CU-03, cuya variable de interés es: Tiempo de respuesta de la arquitectura. Para esta fase, se ejecutaron 405 corridas y se registraron 135 tiempos de respuesta, 45 por cada caso de uso, que son el promedio de 3 corridas por arquitectura y por caso de uso, en la figura 19 se muestra el esquema que se empleó para la realización de las corridas contempladas en esta primera etapa del laboratorio experimental.

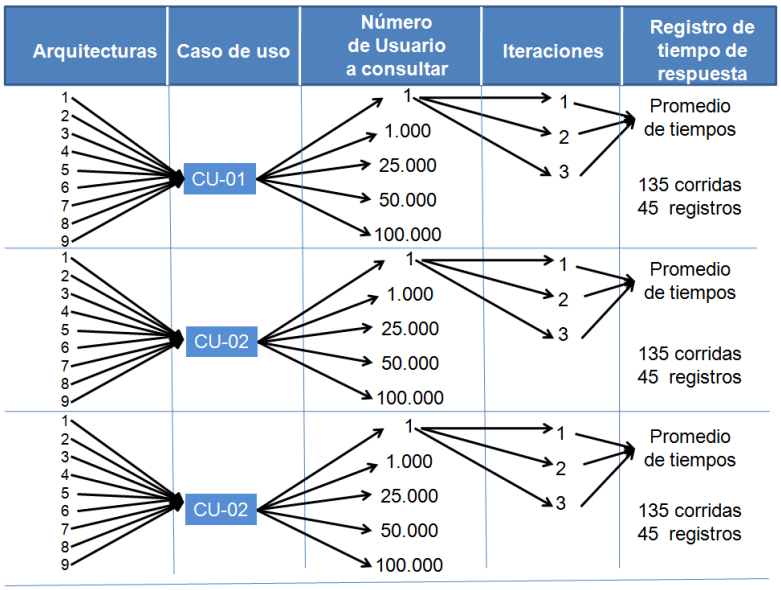


Figura 19:Diseño de la ejecución del laboratorio experimental caso de uso CU-01, CU-02 y CU-03.

La segunda fase, fue desarrollada en función del cuarto caso de uso: CU-04, cuya variable de interés es: Rendimiento de la arquitectura. Para esta fase, se ejecutaron 108 corridas y se registraron 36 tiempos de rendimiento, que resultan del cociente del número de usuarios a ingresar en la base de datos entre el tiempo que dura la inserción, en la figura 20 se muestra el esquema que se empleó para realizarlas corridas contempladas en esta segunda etapa del experimento.

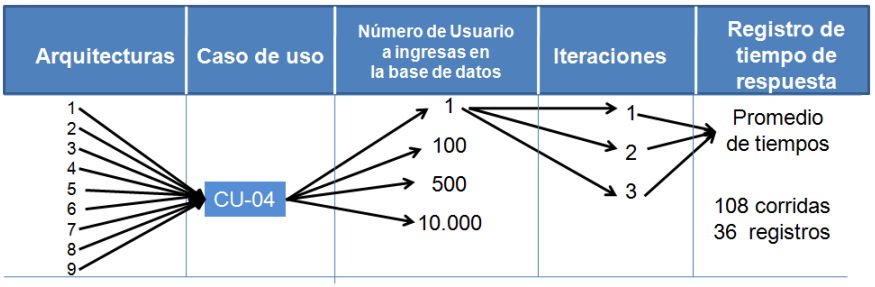


Figura 20:Diseño de la ejecución del experimento caso de uso CU-04.

### 2.3.3 Validación de Data.

La validación de la data se realizó internamente, al terminar cada corrida de la prueba, se revisó el archivo de eventos y otras variables configuradas del entorno de las pruebas, para constatar que cada corrida se ejecutó de manera correcta, sin ningún inconveniente, y así poder considerar el resultado como aceptable. En el caso que existiera alguna evidencia que la corrida estuvo afectada por una eventualidad, el valor de esta fue descartado, y se procedía a repetirla.

## 2.4 Análisis e interpretación.

En función del objetivo principal del estudio, el cual está enfocado en la selección de una arquitectura orientada a microservicios eficiente utilizando el lenguaje de consultas GraphQL, como se ha mencionado se ejecutaron varios experimentos, en que se midieron la eficiencia de las arquitecturas a evaluar desde diferentes puntos de vistas o criterios, por lo cual se requirió emplear una herramienta que pudiese combinar dichos criterios, de tal manera que permitiera escoger una arquitectura más óptima (entre las 9 combinaciones de arquitecturas estudiadas), por ello se empleó la matriz de puntos como metodología para establecer jerarquías o prioridades de los resultados del experimentos. Por lo tanto, su uso facilitó ordenar las arquitecturas, dependiendo de su desempeño en cada una las pruebas realizadas.

La matriz de puntos consistió en asignar una puntación a cada elemento que se está comparando, tratando de establecer un orden entre ellos, el mayor puntaje se le coloca al que demuestre ser más favorable en función del objetivo del análisis, la segunda puntuación más alta al elemento que tenga un buen desempeño, pero esté por debajo del primero y así hasta valorar con una puntuación secuencial a todos los elementos del estudio (ILPES, 2014).

Debido a que los datos obtenidos se obtuvieron en un contexto controlado, se disminuyó considerablemente la subjetividad del experimento, por lo tanto, la matriz de punto

se aplicó sin considerar ponderaciones o correlaciones entre las mediciones, sólo se jerarquizaron las arquitecturas en cada prueba (4 en total) y luego se sumaron las puntuaciones individuales para obtener la total. Con este resultado, que es un resumen de todos los experimentos, se seleccionó la arquitectura que obtuvo la puntuación total más alta, considerando esta como la más eficiente. En la figura 21, se pueden apreciar las características de la matriz de punto utilizada, incluyendo la ecuación para determinar la puntuación total.

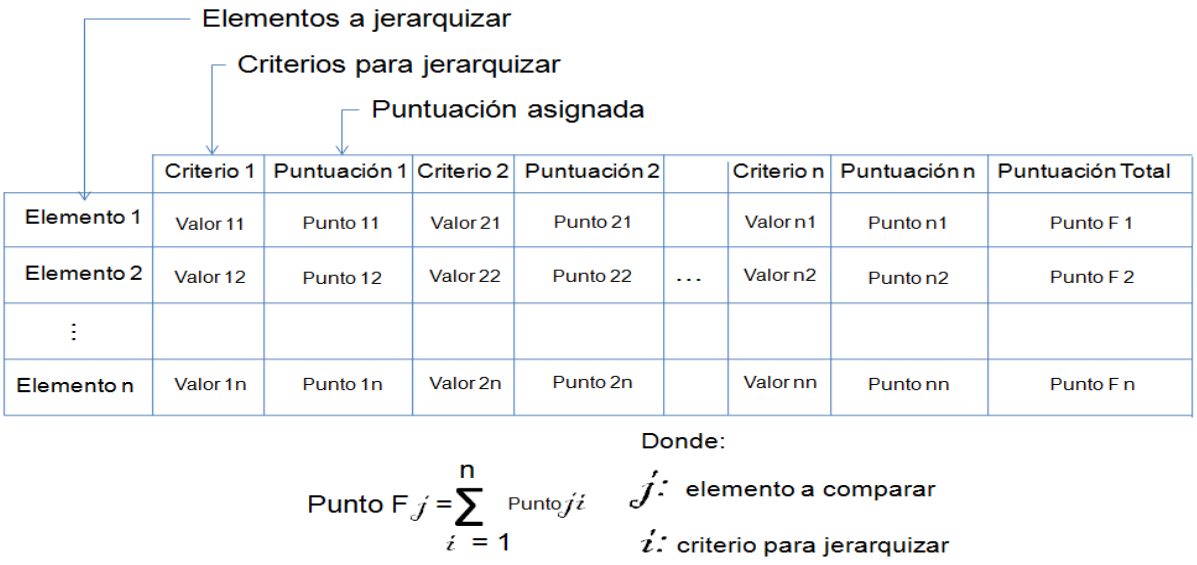


Figura 21:Diseño de la ejecución del experimento caso de uso CU-04.
Fuente: Elaboración propia.

Con la referencia de la matriz de puntos presentada, se procedió a jerarquizar las arquitecturas (elementos a comparar), bajo los criterios de tiempo de respuesta (casos de uso CU-01, CU-02 y CU-03) y el rendimiento en la inserción de registros en la base de datos (caso de uso CU-04), lo cual permitió darle una puntuación que calificara la actuación de cada arquitectura con respecto a las demás, para ello, se le asignó 9 puntos a la arquitectura con mejor actuación, luego 8 puntos a la segunda y así sucesivamente hasta llegar a la que tuvo el registró menos favorable, a la cual se le asignó 1 punto, esto se repitió para cada uno de los casos de uso. Luego, se sumaron las cuatro puntuaciones asignadas y se volvieron a jerarquizar las arquitecturas, obteniendo como resultado que la arquitectura con mejor eficiencia es la constituida por “GqlGen & ReactJs”, con un total de 34 puntos. En la figura 22 se muestra la matriz de puntos resultante del procedimiento aplicado.

Arquitecturas

| Apollo Server & ReactJs | Apollo Server & Django | Apollo Server & Go | Graphene & ReactJs | Graphene & Django | Graphene & Go | GqlGen & ReactJs | GqlGen & Django | GqlGen & Go |
|---|---|---|---|---|---|---|---|---|
| Stack 1 | Stack 2 | Stack 3 | Stack 4 | Stack 5 | Stack 6 | Stack 7 | Stack 8 | Stack 9 |

Resultados de las pruebas

| | CU-01 | | CU-02 | | CU-03 | | CU-04 | | |
|---|---|---|---|---|---|---|---|---|---|
| ARQUTECTURA | TIEMPO DE RESPUESTA | PUNTUACIÓN | TIEMPO DE RESPUESTA | PUNTUACIÓN | TIEMPO DE RESPUESTA | PUNTUACIÓN | RENDIMIENTO | PUNTUACIÓN | PUNTUACIÓN TOTAL |
| Stack 1 | 0,642 | 6 | 3,37 | 8 | 7,3625 | 8 | 140,463226 | 4 | 26 |
| Stack 2 | 0,826 | 4 | 3,794 | 4 | 8,8825 | 4 | 146,8954496 | 5 | 17 |
| Stack 3 | 0,684 | 5 | 3,652 | 5 | 8,5525 | 5 | 236,9160978 | 8 | 23 |
| Stack 4 | 2,97 | 3 | 17,62 | 3 | 39,8375 | 3 | 79,65135792 | 2 | 11 |
| Stack 5 | 3,258 | 1 | 19,166 | 1 | 43,7225 | 1 | 77,69904104 | 1 | 4 |
| Stack 6 | 3,242 | 2 | 19,004 | 2 | 43,5975 | 2 | 80,28986048 | 3 | 9 |
| Stack 7 | 0,468 | 9 | 3,16 | 9 | 6,6925 | 9 | 229,5119684 | 7 | 34 |
| Stack 8 | 0,494 | 8 | 3,47 | 6 | 7,485 | 6 | 217,2070689 | 6 | 26 |
| Stack 9 | 0,522 | 7 | 3,454 | 7 | 7,375 | 7 | 238,5619435 | 9 | 30 |

Arquitectura seleccionada Stack 7, compuesta por GqlGen & ReactJs

Figura 22: Matriz de Punto para jerarquizar las arquitecturas.
Fuente: Elaboración propia.

### 2.4.1 Validar el funcionamiento de la arquitectura más eficiente.

La arquitectura que resultó ser más eficiente, como ya se mencionó fue la constituida por “GqlGen & ReactJs”, cuyo *backend* fue desarrollado en Go y el *frontend* en Java Script; es de resaltar que las arquitecturas que tenían el aplicativo GqlGen obtuvieron rendimientos jerarquizados entre los primeros, como se puede apreciar en la tabla 2.9 donde se muestra la jerarquización final de las pruebas, también es evidente que las arquitecturas cuyo *backend* fue Graphene se posicionaron de últimas.

TABLA 2.9
Jerarquización final de las arquitecturas

| ARQUITECTURA | | PUNTUACIÓN TOTAL | POSICIÓN EN LA JERARQUIZACIÓN FINAL |
|---|---|---|---|
| GqlGen & ReactJs | Stack 7 | 34 | 1 |
| GqlGen & Django | Stack 9 | 30 | 2 |
| Apollo Server & ReactJs | Stack 1 | 26 | 3 |
| GqlGen & Go | Stack 8 | 26 | 4 |
| Apollo Server & Django | Stack 3 | 23 | 5 |
| Apollo Server & Go | Stack 2 | 17 | 6 |
| Graphene & ReactJs | Stack 4 | 11 | 7 |
| Graphene & Go | Stack 6 | 9 | 8 |
| Graphene & Django | Stack 5 | 4 | 9 |

Nota: Elaboración propia.

## 2.5. Desarrollo de una prueba de concepto.

El desarrollo de la prueba de concepto se realizó para comprobar el uso y funcionalidad de la arquitectura más eficiente encontrada en el presente estudio. En esta sección se desglosa las historias de usuarios que plasman los requisitos de software, las principales consideraciones de diseño, el rol definido para la ejecución, se muestra el diseño de pantalla y la estimación del tiempo en la ejecución de las pruebas.

### *2.5.1. Requisitos (historias de usuarios).*

El desarrollo de las historias de usuarios fue en función de las fases de las pruebas, por lo tanto, se construyeron dos historias, una para la primera fase, cuya finalidad fue hacer consultas a la base de datos; para ello se definió la mostrada en la tabla 2.10.

TABLA 2.10
Historia de usuario HU-01 primera fase de las pruebas

| **Historia de Usuario: HU -01** | | | |
|---|---|---|---|
| **Numero:** | 1 | **Nombre:** | Prueba TIEMPO DE RESPUESTA |
| **Usuario:** | INVESTIGADOR | **Iteración:** | 1 |
| **Versión de la Historia:** | 1.0 | **Fecha:** | 20/03/2021 |
| **Prioridad:** | ALTA | **Riesgo:** | BAJO |
| **Descripción:** | | | |
| El INVESTIGADOR tendrá la posibilidad de probar una arquitectura al hacer una consulta a unas bases de datos, para poder registrar el TIEMPO DE RESPUESTA de dicha consulta. Para ello el INVESTIGADOR debe: 1) Seleccionar un FRONTEND; 2) Presionar la opción QUERY TEST; 3) Seleccionar la cantidad de usuarios para la prueba 1, 1.000, 25.000, 50.000 o 100.000; 4) Seleccionar un BACKEND para ejecutar la consulta entre los tres disponibles Apollo Server, Graphene o GqlGen; 5) Seleccionar un CASO DE USO (CU-01, CU-02 o CU-03) que tiene asociado un tipo de consulta a la base de datos; 6) Presionar la opción CONSULTAR. Como resultado el INVESTIGADOR obtendrá el tiempo que duró la consulta. | | | |
| **Criterio de aceptación:** | | | |
| Para considerar la prueba como válida, no debe haber evidencia alguna de que hubo algún error mientras se ejecutaba la consulta, por esto al culminar cada prueba, el INVESTIGADOR debe validar la secuencia de eventos generados. | | | |
| **Observaciones:** | | | |
| Esta HISTORIA DE USUARIO aplica para los tres FRONTEND disponibles, cada uno programado en un lenguaje diferente (JavaScript, Go y Python). | | | |

Nota: Elaboración propia.

Para la segunda fase de las pruebas, cuya finalidad fue hacer inserción de registros en la base de datos, se definió la historia de usuario mostrada en la tabla 2.11.

TABLA 2.11
Historia de usuario HU-02segunda fase de las pruebas

| Historia de Usuario:HU-02 | | | |
|---|---|---|---|
| **Numero:** | 2 | **Nombre:** | Prueba TIEMPO DE RESPUESTA |
| **Usuario:** | INVESTIGADOR | **Iteración:** | 1 |
| **Versión de la Historia:** | 1.0 | **Fecha:** | 20/03/2021 |
| **Prioridad:** | ALTA | **Riesgo:** | BAJO |

**Descripción:**
El INVESTIGADOR tendrá la posibilidad de medir el RENDIMIENTO de una arquitectura al realizar la inserción de registros a una base de datos, para poder registrar el tiempo que toma dicha inclusión. Para ello el INVESTIGADOR debe: 1) Seleccionar un FRONTEND; 2) Presionar la opción MUTACION; 3) Seleccionar la cantidad de usuarios para la prueba 1, 100, 500 o 10.000; 4) Seleccionar un BACKEND para ejecutar la consulta entre los tres disponibles Apollo Server, Graphene o GqlGen; 5) Presionar la opción INGRESAR. Como resultado el INVESTIGADOR obtendrá el tiempo que duró la inserción de los usuarios, con lo que podrá calcular el rendimiento de la arquitectura.

**Criterio de aceptación:**
Para considerar la prueba como válida, no debe haber evidencia alguna de que hubo algún error mientras se ejecutaba la inserción de los registros, por esto al culminar cada prueba, el INVESTIGADOR debe validar la secuencia de eventos generados.

**Observaciones:**
Esta HISTORIA DE USUARIO aplica para los tres FRONTEND disponibles, cada uno programado en un lenguaje diferente (JavaScript, Go y Python).

Nota: Elaboración propia.

### *2.5.2. Diseño, desarrollo (roles, tiempo, pantallas).*

El diseño de las historias de usuarios se desarrolló considerando los siguientes aspectos:

**Roles:** Para las pruebas de conceptos se definió un solo rol, denominado para efecto de las historias de usuarios como el INVESTIGADOR, cuyo objetivo principal es la ejecución de las pruebas de las arquitecturas.

**Pantallas:** En total se desarrollaron 6 pantallas, 3 de ellas asociadas a la primera fase de las pruebas, en las que se hacen consultas a la base de datos, empleando los casos de uso: CU-01, CU-02 y CU-03. En estas pantallas el INVESTIGADOR puede: seleccionar:

- La cantidad de usuarios para la prueba establecida previamente en: 1, 1.000, 25.000, 50.000 o 100.000;
- El *backend* para ejecutar la consulta entre los tres disponibles: Apollo Server, Graphene o GqlGen.

- Un caso de uso que tiene asociado un tipo de consulta a la base de datos.

La pantalla de igual forma le permite iniciar la prueba al presionar la opción CONSULTAR. Como resultado el INVESTIGADOR, obtendrá el tiempo que duró la consulta. En la figura 23, se detalla la pantalla para la primera fase de la prueba.

Las 3 pantallas restantes asociadas a la segunda fase de las pruebas permiten hacer la inserción de registros a la base de datos, empleando el caso de uso: CU-04, igual que las primeras, cada una ellas están desarrolladas en un lenguaje en particular, esto con la fiabilidad de emplear el *frontend* correspondiente al diseño estipulado de las arquitecturas para las pruebas. Estas pantallas las activa el INVESTIGADOR al presionar la opción MUTACION, luego puede seleccionar:

- La cantidad de usuarios para la prueba 1, 100, 500 o 10.000.
- Un BACKEND para ejecutar la inserción de registros, entre los tres disponibles Apollo Server, Graphene o GqlGen.

Para iniciar la corrida debe ser presionado la opción INGRESAR. Como resultado, el INVESTIGADOR obtendrá el tiempo que duró la inserción de los usuarios, con lo que podrá calcular el rendimiento de la arquitectura. En la figura 24, se detalla la pantalla para la segunda fase de la prueba.


Figura 23. Pantalla para la primera fase de la prueba.
Fuente: Elaboración propia.

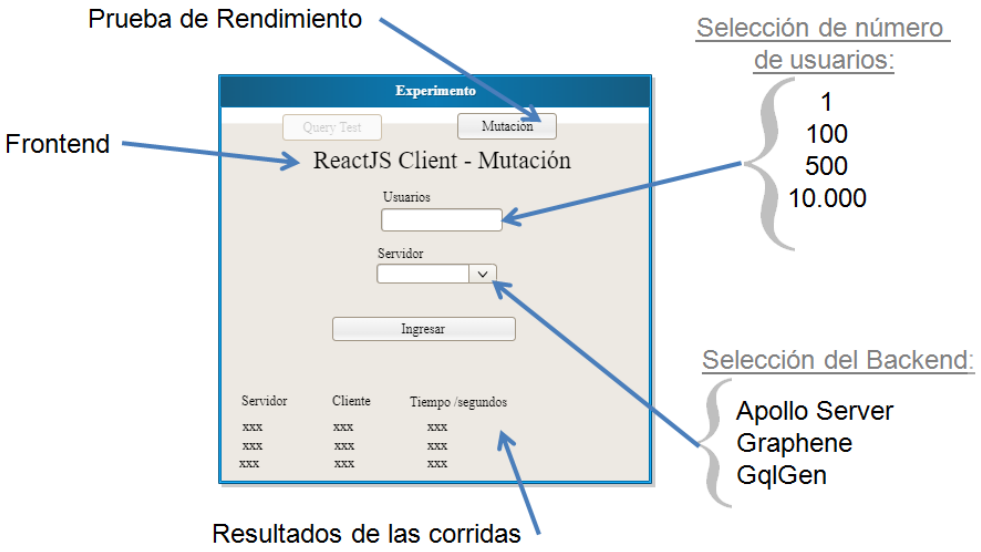


Figura 24. Pantalla para la segunda fase de la prueba.
Fuente: Elaboración propia.

**Tiempo:** el lapso estimado para la ejecución de la prueba estuvo sujeto a la realización de las 513 corridas, en tres etapas: configuración por medio de las pantallas, ejecución de la corrida y la validación de esta, alcanzando un tiempo total invertido de 28 horas, aproximadamente 2,5 minutos por corridas.

### *2.5.3. Pruebas de aceptación (criterios de aceptación de las HU).*

Para considerar la prueba como válida, no debe haber evidencia alguna de que hubo algún error mientras se ejecutaba, en ambas fases de la prueba, al terminar cada corrida de consulta o inserción de registros en la base de datos, el INVESTIGADOR validaba la secuencia de eventos generados, si no se registraba ningún evento que fuese evidencia de algún error durante la prueba, la misma sea aceptada como válida, en el caso contrario sea descartaba y se ejecute nuevamente. Ver tabla 2.12.

TABLA 2.12
Pruebas de Aceptación

| Nro. | Historia de Usuario | Prueba de aceptación | Estado |
|---|---|---|---|
| 1 | HU1 | Obtener esperado establecidos en el Caso de Uso 1 | Exitoso |
| 2 | HU1 | Mostró errores de ejecución. | No Exitoso |
| 3 | HU2 | Obtener esperado establecidos en el Caso de Uso 2 | Exitoso |
| 4 | HU2 | Mostró errores de ejecución. | No Exitoso |

Nota: Elaboración propia.

### *2.5.4. Conclusión del desarrollo.*

Con el desarrollo de las pruebas, se logró determinar cuál de las 9 arquitecturas resultó ser la más eficiente considerando las variables de tiempo de respuesta y rendimiento, en un ambiente de pruebas controlado y con las herramientas necesarias para la validación de las mediciones.

### 2.5.5 Conclusión de la validación.

En las corridas ejecutadas y consideradas para los cálculos, no hubo evidencias de eventos no deseados, por ello se consideraron como válidas. Con la ejecución de los laboratorios experimentales se puede apreciar que es una buena alternativa combinar el lenguaje de consulta GraphQL con las arquitecturas en las que se acoplen los aplicativos GqlGen (*backend* desarrollado en Go) y ReactJs (*frontend* desarrollado en Java Script).

# CAPÍTULO 3

## Análisis de Resultados

En este capítulo se presenta y analiza los datos obtenidos en los laboratorios experimentales ejecutados. En esencia, se tomaron registros de dos variables de calidad ajustadas a la norma ISO/IEC 25023, ya que son atributos de calidad de la característica de eficiencia en el desempeño, por ello fueron base para la selección de la arquitectura más recomendable para interactuar con el lenguaje de consulta GraphQL.

### 3.1 Resultados.

Siguiendo el esquema definido, se analizó la eficiencia del desempeño partiendo del comportamiento de dos de las variables. La primera de ellas es el tiempo de respuesta para ejecutar consultas a la base de datos y para evaluarla se diseñaron tres casos de uso. La segunda variable, rendimiento medido por el número de tareas ejecutadas en una unidad de tiempo. Conjugar ambas mediciones, permite jerarquizar las arquitecturas desde la más eficiente hasta la menos eficiente.

#### 3.1.1. Tiempo de Respuesta.

La variable tiempo de respuesta, según la norma ISO/IEC 25023, se mide desde que se inicia la instrucción hasta que se obtienen los resultados esperados. El valor esperado de la medición sería lo más cercano a cero, es decir la arquitectura que registre el menor tiempo es más eficiente.

##### *3.1.1.1. Caso de Uso CU-01.*

Este caso de uso se diseñó para hacer una consulta a la tabla *users* de la base de datos. Esta acción se ejecutó con diferentes cantidades de registros por cada una de las arquitecturas a evaluar. Cada corrida, se repitió en tres oportunidades, se contabilizó el tiempo que tardó la consulta y se promediaron, de lo que resultaron 45 tiempos, mostrados de forma ordenada en la tabla 3.1, se puede apreciar el tiempo en promedio que tardó cada corrida por arquitectura y cantidad de registro a consultar.

TABLA 3.1
Caso de uso CU-01

| CU-01 | Stack 1 | Stack 2 | Stack 3 | Stack 4 | Stack 5 | Stack 6 | Stack 7 | Stack 8 | Stack 9 |
|---|---|---|---|---|---|---|---|---|---|
| # Usuarios | Apollo Server & ReactJs | Apollo Server & Django | Apollo Server & Go | Graphene & ReactJs | Graphene & Django | Graphene & Go | GqlGen & ReactJs | GqlGen & Django | GqlGen & Go |
| 1 | 0.03 | 0.51 | 0.04 | 0.02 | 0.06 | 0.02 | 0.02 | 0.01 | 0 |
| 1000 | 0.10 | 0.10 | 0.07 | 0.10 | 0.11 | 0.10 | 0.04 | 0.04 | 0.04 |
| 25000 | 0.49 | 0.73 | 0.50 | 2.10 | 2.31 | 2.33 | 0.35 | 0.37 | 0.40 |
| 50000 | 0.87 | 0.94 | 0.99 | 4.18 | 4.56 | 4.63 | 0.65 | 0.69 | 0.73 |
| 100000 | 1.72 | 1.85 | 1.82 | 8.45 | 9.25 | 9.13 | 1.28 | 1.36 | 1.44 |
| Promedio | 0.642 | 0.826 | 0.684 | 2.970 | 3.258 | 3.242 | 0.468 | 0.494 | 0.522 |

Nota: Los datos tabulados corresponde al tiempo de respuesta (expresados en segundos) de las arquitecturas. Elaboración propia.

Obtenidos los 45 tiempos, se procedió a promediarlos resultados por arquitectura lo que permitió la comparación. Para este primer caso de uso, la arquitectura que obtuvo el menor tiempo de respuesta es la conformada por GqlGen & Reactjs, con un valor de 0,468 segundos. Sin embargo, no se puede perder de vista, que las opciones que alcanzaron a registrar tiempos muy cercanos fueron GqlGen & Django con 0,494 segundos y GqlGen & Go con 0,522 segundos, estas tres arquitecturas tienen en común el aplicativo GqlGen.

El comportamiento del tiempo de respuesta estuvo relacionado con el *backend* de la arquitectura, como se puede apreciar en la figura 25, las corridas que se ejecutaron con el aplicativo GqlGen, resultaron ser las más rápidas, por lo tanto, con mayor eficiencia. En cambio, en el caso de las arquitecturas cuyo *backend* es Graphene, los tiempos de repuesta son los más altos con una brecha significativa, lo que se interpreta que son las menos eficientes. Es de acotar que el grupo de arquitectura, cuyo *backend* es Apollo Server, se considera el segundo mejor grupo según los tiempos censados.

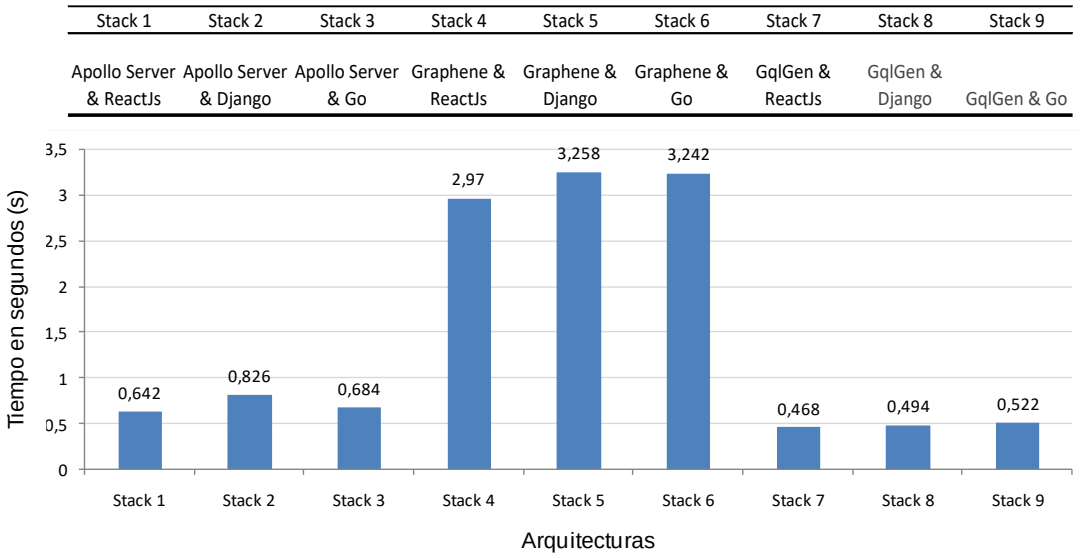


Figura 25. Tiempo de respuesta promedio por arquitectura, CU-01.
Fuente: Elaboración propia.

### *3.1.1.2. Caso de Uso CU-02.*

Al ejecutar las corridas de este laboratorio experimental, se realizó la consulta de dos tablas de la base de datos reconocidas como *users* y *post;* esta acción se ejecutó con diferentes cantidades de registros por cada una de las arquitecturas a evaluar. Cada corrida se repitió en tres oportunidades, se contabilizó el tiempo que tardó la consulta y se promediaron, de lo que resultaron 45 tiempos, mostrados de forma ordenada en la tabla 3.2, donde se puede apreciar el tiempo que se tarda en promedio la corrida por arquitectura y cantidad de registros a consultar. En líneas generales, las corridas han tomado más tiempo, que el anterior, sólo con incluir una segunda tabla en la consulta.

TABLA 3.2
Caso de uso CU-02

| CU-02 | Stack 1 | Stack 2 | Stack 3 | Stack 4 | Stack 5 | Stack 6 | Stack 7 | Stack 8 | Stack 9 |
|---|---|---|---|---|---|---|---|---|---|
| Usuarios | Apollo Server & ReactJs | Apollo Server & Django | Apollo Server & Go | Graphene & ReactJs | Graphene & Django | Graphene & Go | GqlGen & ReactJs | GqlGen & Django | GqlGen & Go |
| 1 | 0.11 | 0.13 | 0.03 | 0.03 | 0.03 | 0.02 | 0.02 | 0.02 | 0.01 |
| 1000 | 0.21 | 0.21 | 0.20 | 0.49 | 0.59 | 0.54 | 0.1 | 0.12 | 0.11 |
| 25000 | 2.43 | 2.60 | 2.53 | 12.32 | 13.41 | 13.35 | 2.8 | 2.95 | 3.06 |
| 50000 | 4.65 | 5.08 | 5.04 | 25.21 | 27.63 | 26.86 | 4.53 | 5.13 | 5.13 |
| 100000 | 9.45 | 10.95 | 10.46 | 50.05 | 54.17 | 54.25 | 8.35 | 9.13 | 8.96 |
| Promedio | 3.37 | 3.794 | 3.652 | 17.620 | 19.166 | 19.004 | 3.160 | 3.470 | 3.454 |

Nota: Los datos tabulados corresponden al tiempo de respuesta (expresados en segundos) de las arquitecturas. Elaboración propia.

En este caso se trató de la misma forma que el anterior, obtenidos los 45 tiempos, se procedió a promediar los resultados por arquitectura lo que permite la

comparación. Nuevamente, la arquitectura que obtuvo el menor tiempo de respuesta es la conformada por GqlGen & Reactjs, con un valor de 3,16 segundos, según la definición de esta variable, esta combinación de aplicativos tiene un buen desempeño y en la jerarquización realizada obtuvo la mayor puntuación, seguida de arquitectura definida por Apollo Server & ReactJs con un tiempo de 3,37 segundos y luego por GqlGen & Django que alcanzó un tiempo de 3,47 segundos. Es de resaltar, que estas dos últimas arquitecturas, comparten aplicativos con la que obtuvo el mejor tiempo de respuesta.

El tiempo de respuesta, tuvo el mismo comportamiento que el caso de uso anterior de esta fase del experimento, su tendencia estuvo relacionada con el *backend* de la arquitectura, como lo que muestra la figura 26. Las corridas que se ejecutaron con el aplicativo GqlGen resultaron ser las más rápidas, por lo tanto, con mayor eficiencia. Las arquitecturas en que se empleó Apollo Server, mostraron un comportamiento muy cercano a las citadas anteriormente, de igual forma se pueden considerar aceptables. En cambio, en el caso de las arquitecturas cuyo *backend* es Graphene, los tiempos de respuesta son los más altos, con una diferencia sustancial en comparación a los dos grupos de arquitecturas referidas inicialmente, por lo tanto, estas serían las menos eficientes.

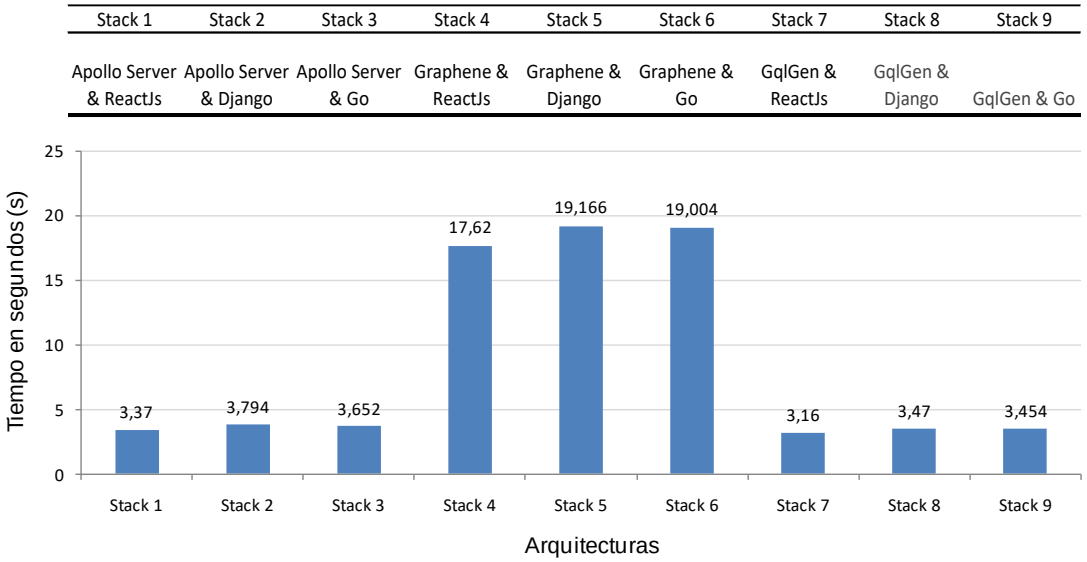


Figura 26. Tiempo de respuesta promedio por arquitectura, CU-02.
Fuente: Elaboración propia

### 3.1.1.3. Caso de Uso CU-03.

Este tercer caso de uso se hace para consultar a tres tablas de la base de datos reconocidas como users, post y comment, esta acción se ejecutó con diferentes cantidades de registros por cada una de las arquitecturas a evaluar. Para cada corrida se aplicó el mismo procedimiento de los casos anteriores, de lo que resultaron 45 tiempos, mostrados de forma ordenada en la tabla 3.3 en la que se puede apreciar el tiempo que se tarda en promedio la corrida por arquitectura y cantidad de registro a consultar. Con este caso de uso, se culmina la primera fase del laboratorio experimental, en que se evalúa el tiempo de respuesta de las arquitecturas.

TABLA 3.3
Caso de uso CU-03

| CU-03 | Stack 1 | Stack 2 | Stack 3 | Stack 4 | Stack 5 | Stack 6 | Stack 7 | Stack 8 | Stack 9 |
|---|---|---|---|---|---|---|---|---|---|
| Usuarios | Apollo Server & ReactJs | Apollo Server & Django | Apollo Server & Go | Graphene & ReactJs | Graphene & Django | Graphene & Go | GqlGen & ReactJs | GqlGen & Django | GqlGen & Go |
| 1 | 0.07 | 0.09 | 0.02 | 0.05 | 0.06 | 0.05 | 0.03 | 0.02 | 0.02 |
| 1000 | 0.44 | 0.50 | 0.47 | 2.90 | 2.71 | 2.91 | 0.28 | 0.30 | 0.30 |
| 25000 | 9.43 | 10.71 | 10.31 | 51.77 | 56.61 | 56.98 | 9.44 | 10.29 | 10.15 |
| 50000 | 19.51 | 24.23 | 23.41 | 104.63 | 115.51 | 114.45 | 17.02 | 19.33 | 19.03 |
| 100000 | 44.45 | 51.42 | 48.49 | 216.98 | 237.14 | 233.17 | 33.33 | 37.46 | 37.82 |
| Promedio | 7.3625 | 8.8825 | 8.5525 | 39.8375 | 43.7225 | 43.5975 | 6.6925 | 7.4850 | 7.3750 |

Nota: Los datos tabulados corresponden al tiempo de respuesta (expresados en segundos) de las arquitecturas.
Elaboración propia

Para este ensayo, se esperaban los mayores tiempos de respuesta, ya que implicaba la consulta de las tres tablas de la base de datos, tal como resultó. El tiempo medido incrementó de forma exponencial, en relación con las pruebas previas. De igual forma que los anteriores casos se trataron los resultados, obtenidos los 45 tiempos se procedió a promediar los resultados por arquitectura lo que permite la comparación. La arquitectura que obtuvo el menor tiempo de respuesta es la conformada por GqlGen & Reactjs, con un valor de 6,6925 segundos, seguida de arquitectura definida por Apollo Server & ReactJs con un tiempo de 7,3625 segundos y luego estuvo GqlGen & Go alcanzando un tiempo de 7,375 segundos; es de apreciar que existen aplicativos en común entre estas tres arquitecturas que fueron jerarquizadas como las más eficientes en este laboratorio experimental.

En esta oportunidad las corridas que se ejecutaron con el aplicativo GqlGen resultaron ser las más rápidas, ratificando los resultados de los casos anteriores. En

cambio, en el caso de las arquitecturas cuyo *backend* es Graphene los tiempos de repuesta son los más altos, por lo tanto, las menos eficientes. Este tercer caso de uso ratifica que el tiempo de repuesta de las arquitecturas depende del *backend* de la arquitectura. La comparación del tiempo de repuesta está reflejada en la figura 27.

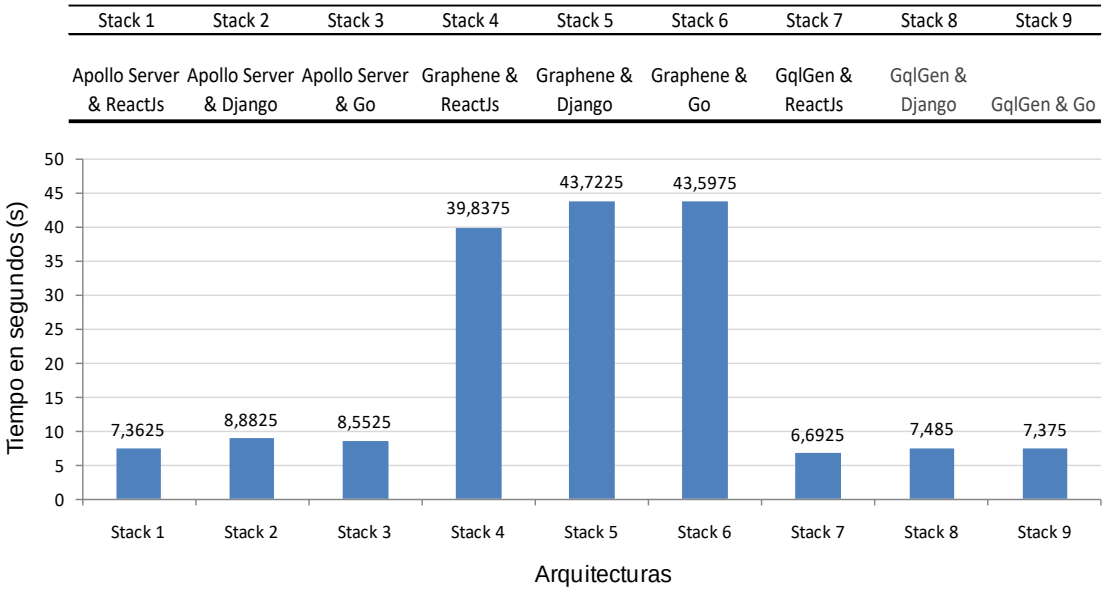


Figura 27. Tiempo de respuesta promedio por arquitectura, CU-03.
Fuente: Elaboración propia

### 3.1.2 Rendimiento.

La variable rendimiento, según la norma ISO/IEC 25023, se calcula considerando una cantidad de tareas ejecutadas, por unidad de tiempo. Para efecto de esta investigación, se cuenta una tarea por cada inserción de un registro en la base de datos, acciones que son contabilizadas y dividas por el tiempo en que se ejecutaron estas tareas.

#### 3.1.2.1 Caso de Uso CU-04.

En la ejecución de este caso de uso en cada corrida se insertaron en la base de datos una cantidad predeterminada de datos (usuarios) y se midió el tiempo en que esto se ejecutó, y luego se calculó el cociente entre ambas variables. Con este esquema se obtuvieron 36 valores del rendimiento, que se presentan en la tabla 3.4

.

TABLA 3.4
caso de uso CU-04

| CU-04 | Stack 1 | Stack 2 | Stack 3 | Stack 4 | Stack 5 | Stack 6 | Stack 7 | Stack 8 | Stack 9 |
|---|---|---|---|---|---|---|---|---|---|
| Usuarios | Apollo Server & ReactJs | Apollo Server & Django | Apollo Server & Go | Graphene & ReactJs | Graphene & Django | Graphene & Go | GqlGen & ReactJs | GqlGen & Django | GqlGen & Go |
| 1 | 20 | 12.5 | 20 | 20 | 20 | 10 | 25 | 33.3333 | 10 |
| 100 | 45.6621005 | 52.0833333 | 76.9230769 | 25.3807107 | 24.7524752 | 26.5251989 | 74.0740741 | 51.2820513 | 82.6446281 |
| 500 | 42.8816467 | 52.6315789 | 82.1018062 | 24.9625562 | 24.4977952 | 25.1635632 | 67.6589986 | 79.491256 | 84.6023689 |
| 10000 | 453.309157 | 470.366886 | 768.639508 | 248.262165 | 241.545894 | 259.47068 | 751.314801 | 704.721635 | 777.000777 |
| | 140.4632 | 146.8954 | 236.9161 | 79.6514 | 77.6990 | 80.2899 | 229.5120 | 217.2071 | 238.5619 |

Nota: Los datos tabulados corresponden al rendimiento (expresados en tareas/segundo) de las arquitecturas.
Elaboración propia.

Con la finalidad de establecer la comparación entre las arquitecturas se determinó el rendimiento promedio de cada una de ellas, de lo que resultó que la combinación fue GqlGen & Go demostró un mejor rendimiento con la ejecución de 238,56 tareas/segundo, en este contexto de pruebas. La segunda arquitectura en demostrar un buen rendimiento está definida por Apollo Server & Go con un rendimiento de 236,916tareas/segundos, y luego estuvo GqlGen & ReactJs alcanzando un rendimiento de 229,511 tareas/segundos, es de apreciar que existen aplicativos en común entre estas tres arquitecturas que fueron jerarquizadas, como las más eficientes en este experimento.

Las corridas que se ejecutaron con el aplicativo GqlGen como *backend*, obtuvieron el mejor rendimiento. En cambio, en el caso de las arquitecturas cuyo *backend* Graphene registraron el menor rendimiento. En la figura 28 se muestran los rendimientos promedios calculados y el comportamiento del rendimiento por arquitectura para este cuarto caso de uso, el cual estuvo relacionado con el *backend* de la arquitectura.

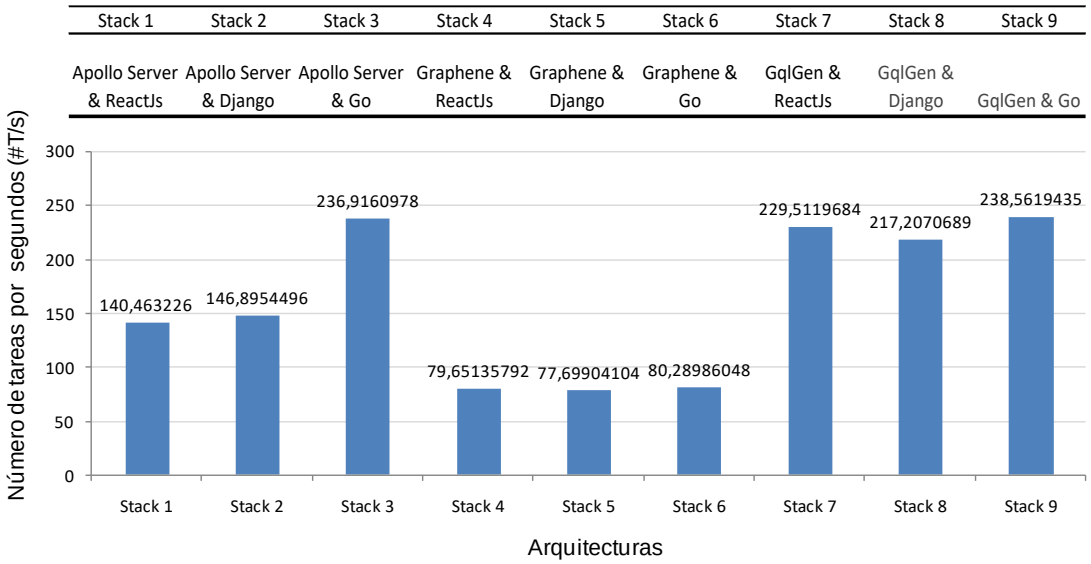


Figura 28.Rendimiento promedio por arquitectura, CU-04.
Fuente: Elaboración propia.

## 3.2 Eficiencia del desempeño.

Para efecto de la investigación bajo el marco de la norma ISO/IEC 25023, la eficiencia del desempeño de las arquitecturas sometidas a evaluación se estima a través de las variables analizadas en la sección 3.1. Por ello se empleó una matriz de punto explicada en la sección 2.4, logrando conjugar ambas variables (tiempo de respuesta y rendimiento).

Luego de completar la matriz de punto, se obtienen la jerarquización de las arquitecturas según la eficiencia, el orden resultante se muestra en la tabla 3.5.

TABLA 3.5
Jerarquización de las arquitecturas según la eficiencia

| POSICIÓN EN LA JERARQUIZACIÓN FINAL | ARQUITECTURA | |
|---|---|---|
| 1 | Stack 7 | GqlGen & ReactJs |
| 2 | Stack 9 | GqlGen & Django |
| 3 | Stack 1 | Apollo Server & ReactJs |
| 4 | Stack 8 | GqlGen & Go |
| 5 | Stack 3 | Apollo Server & Django |
| 6 | Stack 2 | Apollo Server & Go |
| 7 | Stack 4 | Graphene & ReactJs |
| 8 | Stack 6 | Graphene & Go |
| 9 | Stack 5 | Graphene & Django |

Nota: Elaboración propia.

### 3.2.1 Análisis de resultados de *Backend.*

Los datos obtenidos como resultado de la jerarquización de las arquitecturas en cuanto al *backend* empleado son de interés, ya que es evidente que el aplicativo que cumpla con esta función marca significativamente la eficiencia de la arquitectura. Con la ayuda de la figura 29, se puede identificar la aplicación GqlGen como el *backend* que más favorece a la eficiencia de desempeño de las arquitecturas. En segundo lugar, está Apollo Server, y por último se tiene a Graphene; es de resaltar que existe una brecha significativa entre el desempeño del último aplicativo y los dos primeros, por lo tanto, es el menos recomendado.

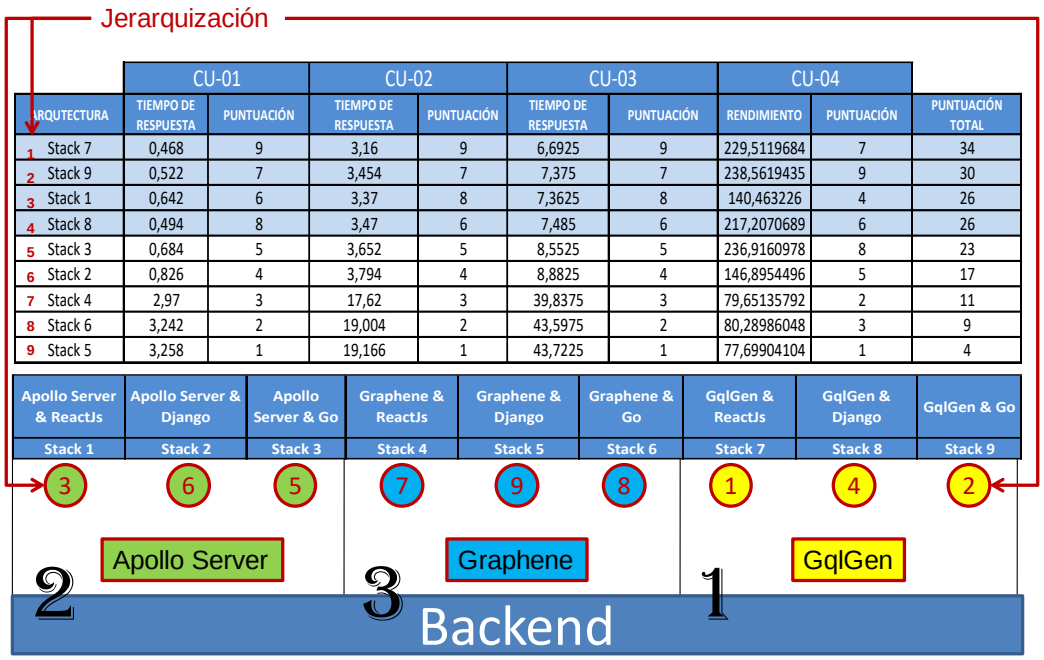


Figura 29. Jerarquización de arquitecturas *backend*.
Fuente: Elaboración propia.

### 3.2.2 Análisis de resultados de *Frontend.*

En cuanto al *frontend*, en líneas generales el laboratorio experimental no arrojó evidencia que indique la existencia de una influencia significativa entre el tipo de aplicativo usado y la eficiencia en el desempeño de las arquitecturas. Sin embargo, en la jerarquización final de las arquitecturas se puede observar que el empleo de ReactJs se posicionó en los primeros lugares, en especial con la combinación de Apollo Server y GqlGen, por ende, es un aplicativo que aporta a la eficiencia como *frontend*, pero depende del *backend* con que se haga acompañar.

### 3.3 Pruebas de hipótesis.

En este estudio, para la prueba de hipótesis se tomó los promedios calculados en los casos de uso del laboratorio experimental, con la finalidad de compararlos para evaluar la viabilidad de cada uno de los planteamientos (NOTA: no se aplicó pruebas estadísticas).

Para Hipótesis Nula ($H_0$):

- Todas las arquitecturas orientadas a microservicios, utilizando el lenguaje de consultas GraphQL, tienen la misma eficiencia.

Se considera válida esta hipótesis si todos los promedios de las variables de eficiencia son iguales, es decir, $H_0$: $\mu_{ij} = \mu_{ij}$; donde i es el caso de uso y j la arquitectura evaluada. Considerando, los resultados expuestos en cada caso de uso, que se resumen en la figura 32, es evidente que no se cumple lo planteado en la hipótesis, de hecho, el hallazgo es que $\mu_{ij} \neq \mu_{ij}$, para toda i y j. Por lo tanto, se rechaza la hipótesis $H_0$.

Para Hipótesis Alternativa ($H_1$):

- Es posible desarrollar una arquitectura orientada a microservicios, utilizando el lenguaje de consultas GraphQL, que sea más eficiente que otras.

Se considera válida esta hipótesis si existen un promedio que indique que una de las arquitecturas es más eficiente que el resto de las evaluadas, $H_1$: $\mu_{ij} > \mu_{ij}$; donde i es el caso de uso y j la arquitectura evaluada. Considerando los resultados expuestos en cada caso de uso, que se resumen en la figura 30, es posible identificar cual arquitectura fue más eficiente ante la prueba aplicada, esto comparando los promedios obtenidos. Por lo tanto, se acepta la hipótesis $H_1$.

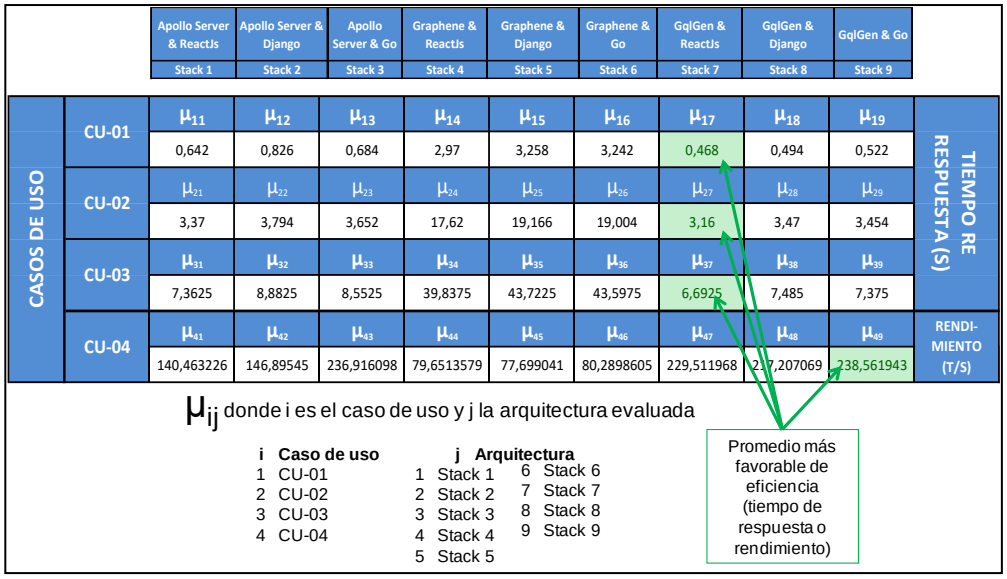


Figura 30. Esquema resumen de los resultados de la experimentación.
Fuente: Elaboración propia.

# Conclusiones

- Mediante el marco teórico establecido, se logró realizar el estudio que pudo comprobar la arquitectura más eficiente usando GraphQL, mediante un laboratorio experimental controlado, que siguió un marco referencial para la prueba de software (Wohilin et al. 2012) y estableciendo las métricas de calidad de eficiencia descrita en la normativa internacional ISO / IEC 25023: 2016.
- Con el laboratorio experimental diseñado en este trabajo y la metodología de la matriz de puntos se logró establecer, luego de comprar varios aplicativos, qué la arquitectura de software es la más eficiente, cuando a nivel de Backend el lenguaje de programación es GO utilizando la librería GQLGen para la gestión de consulta de datos con GraphQL y en lo que se refiere al Frontend, está presente, el lenguaje de programación Javascript utilizando el framework ReactJs. La combinación de ambas herramientas alcanzó la mayor puntuación, de la evaluación luego de ejecutar 4 casos de uso, alcanzando 34 puntos de un tope de 36, según la matriz de punto diseñada.
- De las arquitecturas concebidas para realizar la fase experimental de esta investigación, resultó ser la más eficiente la que tiene en el *backend* a GQLGen y en el *frontend* a ReactJS, ya que de los tiempos de respuestas medidos, este diseño registró los menores en los tres casos de uso ejecutados, así como también obtuvo el tercer lugar en el caso de uso en que se evaluó el rendimiento, al combinar los experimentos y comparar las nueves arquitecturas, la combinación GQLGen–ReacJS, fue la que mejor desempeño registro.
- Con los resultados obtenidos en los casos de uso en que se evaluó el tiempo de respuesta, se concluye que esta variable está relacionada principalmente con el aplicativo que se configure en el *backend*, dependiendo de este la arquitectura será más o menos eficiente, en términos de rapidez de respuesta, para efectos de los experimentos ejecutados, el backend con mejor desempeño fue GQLGen.
- En el caso de los *frontend* utilizados en las pruebas, no hubo evidencia que indique que influyen en el desempeño de la arquitectura de manera significativa, sin embargo, en el ranqueo final de las arquitecturas se puede observar que el empleo de ReactJs se posicionó en los primeros lugares, en especial con la combinación de Apollo Server y GqlGen, por ende, es un aplicativo que aporta a la eficiencia como *frontend*, pero

depende del *backend* con que se haga acompañar.

- Con el desarrollo de un aplicativo (prueba de concepto) se logró establecer y validar la implementación de la arquitectura más eficiente obtenida en el experimento, la cual se realizó mediante 2 historias de usuario y 4 pruebas de aceptación.

# Recomendaciones

- Considerando el alcance de la presente investigación es recomendable para futuros trabajos, evaluar los demás criterios de calidad enmarcados en la ISO/IEC 25023, que no son medidos en esta oportunidad, de tal manera de que sean complementariosen sus análisis.
- Se recomienda usar la librería ‘Dataloader’ para optimizar las búsquedas por niveles de datos, cuando se utilice el lenguaje de consultas de GraphQLen cada servicio de *backend*, el cual optimiza las búsquedas a la mitad de tiempo, optimizando consultas SQL y dando un mejor manejo de los datos que se están buscando.
- En este estudio se configuraron 9 arquitecturas con aplicativos asociados a 3 lenguajes que son compatibles con la herramienta GraphQL y que registran mayor demanda de uso por las comunidades de programadores, estos lenguajes son: GO, Python y JavaScript, por lo tanto, se recomienda realizar ensayos con otras opciones, manteniendo un ambiente de pruebas de controlado.
- Dada la metodología de prueba de software utilizada, uno de los aspectos que permita la validación de los experimentos es tener un ambiente controlado, y esto se caracteriza por el hecho de que permita registrar los eventos que se presenta durante la ejecución de las pruebas para su posterior análisis, por ello se recomienda, el esquema presentado por Wohilin et al. (2012), para la ejecución y control de las fases de experimentales de los aplicativos.

# Referencias


Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in practice.* Addison-Wesley Professional.

Bottero, M., Datola, G., & Monaco, R. (2017). Exploring the Resilience of Urban Systems Using Fuzzy Cognitive Maps. *Computational Science and Its Applications – ICCSA 2017*, (págs. 338-353).

Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., y otros. (6 de Octubre de 2014). *Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing.* Recuperado el 15 de Mayo de 2021, de USENIX : https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin

Byron, L. (14 de Septiembre de 2015). *GraphQL: A data query language.* Recuperado el 21 de Mayo de 2021, de Facebook Engineering : https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/

Chicaiza, D. (2020). *Diseño de un prototipo de una arquitectura basada en microservicios para la integración de aplicaciones Web altamente transaccionales. Caso: Entidades integración de aplicaciones Web altamente transaccionales. Caso: Entidades.* Tesis de Grado, Universidad Politécnica Salesiana, Quito, Ecuador.

Chulca, C., & Molina, R. (2020). *Migración hacia una arquitectura basada en microservicios del sistema de gestión centralizada de laboratorio de la DGIP.* Tesis de Grado, Escuela Politécnica Nacional, Quito, Ecuador.

Di Francesco, P., Lago, P., & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *Journal of Systems and Software , 150*, 77-97.

Fielding, R. (2000). *Architectural styles and the design of network -based software architectures.* Tesis de Doctorado, University of California, Oakland, U.S.A.

Flores, J. (2020). *Estudio de Una Arquitectura de Microservicios Mediante Spring Cloud Para El Desarrollo Del Módulo de Registro y Seguimiento Médico de Los Deportistas En La Federación Deportiva de Imbabura.* Universidad Técnica del Norte, Ibarra, Ecuador.

Freeman, A. (2019). *Understanding GraphQL.* Berkeley, , California, U.S.A: Apress.

Github. (01 de 03 de 2021). *GitHub, Inc*. Obtenido de GitHub, Inc: https://github.com/99designs/gqlgen

Github. (01 de 03 de 2021). *GitHub, Inc.* Obtenido de GitHub, Inc.: https://github.com/graphql-python/gql

Github. (01 de 03 de 2021). *Graphene-Python*. Obtenido de Graphene-Python: https://graphene-python.org/

GraphQL. (s.f.). *Code using GraphQl*. Recuperado el 20 de Mayo de 2021, de GraphQl: https://graphql.org/code/

Hartina, D., Lawi, A., & Enrico, B. (2018). Performance Analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University. *2nd East Indonesia Conference on Computer and Information Technology (EIConCIT).* Makassar, Indonesia .

Hinojosa, J. (2017). *Arquitectura de software basada en microservicios para desarrollo de aplicaciones web de la asamblea naciona.* Tesis de Msc., Universidad Técnica del Norte, Ibarra, Ecuador.

ILPES. (2014). *Manual metodológico de evaluación multicriterio para programas y proyectos.* Santiago de Chile: Publicaciones de las Naciones Unidas.

Keepcoding Tech School. (2018). *Lenguaje de programacion Go y sus características*. Recuperado el 23 de Mayo de 2021, de Keepcoding Tech School: https://keepcoding.io/blog/lenguaje-de-programacion-go-caracteristicas/

Khan, R., & Noor, A. (2020). Sustainable IoT Sensing Applications Development through GraphQL-Based Abstraction Layer. *Electronics , 9* (564), 1-23.

Lenarduzzi, V., Lomio, F., saarimaki, N., & Taibi, D. (2020). Does migrating a monolithic system to microservices decrease the technical debt? *Journal of Systems and Software , 169*.

Madsen, M., Lhotak, O., & Tip, F. (2020). A Semantics for the Essence of React. En R. Hirschfeld, & T. Pape (Ed.), *34th European Conference on Object-Oriented Programming (ECOOP 2020).*, (págs. 1-26). Alemania.

Maldonado, L. (8 de Abril de 2019). *Why GraphQL is the future of API´s.* Recuperado el 20 de Mayo de 2021, de FreeCodeCamp: https://medium.com/free-code-camp/why-graphql-is-the-future-of-apis-6a900fb0bc81

Malhotra, R. (2019). *Rapid Java Persistence and Microservices.* Faridabad, Haryana, India: Apress.

Organización Internacional de Normalización. (2016). *System and Software Engineering-*

*Systems and Software Qaulity Requirements and Evaluation (SQuaRE)- Measurement of system and software product quality*.

PostgreSQL. (20 de Mayo de 2021). *PostgreSQL*. Recuperado el 23 de Mayo de 2021, de PostgreSQL: https://www.postgresql.org/about/

Red Hat, Inc. (01 de 04 de 2021). *https://www.redhat.com/*. Obtenido de https://www.redhat.com/: https://www.redhat.com/es/topics/containers/what-is-docker

Rodríguez, K. (2020). *Desarrollo de un envoltorio del API -REST de Menfeley con GraphQL.* Tesis de Grado, Universiad Técnica del Norte, Ibarra, Ecaudor.

Rozanski, N., & Woods, E. (2012). *Software Systems Architecture* (Segunda edicción ed.). Westford, Massachusetts, U.S.A: Pearson Education. Inc.

Veritone, Inc. (01 de 03 de 2021). *Veritone, Inc.* Obtenido de Veritone, Inc.: https://machinebox.io/

Wiesbauer, M. (Junio de 2019). *The benefits of GraphQL for more effcient APIs: A better version of REST.* Recuperado el 20 de Mayo de 2021, de MWIESBAU: https://mwiesbau.github.io/online-cv/assets/4_whitepaper.pdf

Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., & Wesslen, A. (2012). *Experimentation in Software Engineering.* Verlag Berlin , Heidelberg, Alemania: Springer.

Yaguachi, L. (2017). *Aplicación de un modelo para evaluar el rendimiento en el proceso de migración de una aplicación monolítica hacia una orient.* Tesis de Grado, Universidad Técnica Particular de Loja, Loja, Ecuador.